IBM System Blue Gene Solution: Blue Gene/P Application Development

Understand the Blue Gene/P programming environment

Learn how to run and debug MPI programs

Learn about Bridge and Real-time APIs
Second Edition (September 2008)

This edition applies to Version 1, Release 2, Modification 0 of IBM System Blue Gene/P Solution (product number 5733-BGP).

Note: Before using this information and the product it supports, read the information in “Notices” on page ix.
Contents

Notices ................................................................................................................ ix
Trademarks ........................................................................................................ x

Preface ................................................................................................................. xi
The team that wrote this book ........................................................................... xi
Become a published author .............................................................................. xiii
Comments welcome .......................................................................................... xiii

Summary of changes ......................................................................................... xv
IBM System Blue Gene Solution: Blue Gene/P Application Development, Second Edition. xv

Part 1. Blue Gene/P: System and environment overview ......................................... 1

Chapter 1. Hardware overview ........................................................................... 3
  1.1 System architecture overview ....................................................................... 4
    1.1.1 System buildup .................................................................................... 5
    1.1.2 Compute and I/O nodes ....................................................................... 5
    1.1.3 Blue Gene/P environment ..................................................................... 6
  1.2 New on Blue Gene/P .................................................................................... 7
  1.3 Microprocessor ............................................................................................. 8
  1.4 Compute nodes ............................................................................................. 9
  1.5 I/O nodes .................................................................................................... 10
  1.6 Networks ..................................................................................................... 10
  1.7 Blue Gene/P programs ................................................................................ 11
  1.8 Blue Gene specifications ............................................................................. 12
  1.9 Host system ................................................................................................ 13
    1.9.1 Service node ....................................................................................... 13
    1.9.2 Front end nodes .................................................................................. 13
    1.9.3 Storage nodes ..................................................................................... 13
  1.10 Host system software ................................................................................ 14

Chapter 2. Software overview ............................................................................ 15
  2.1 Blue Gene/P software at a glance ................................................................ 16
  2.2 Compute node kernel ................................................................................... 17
    2.2.1 High-performance computing and High-Throughput Computing modes. 18
    2.2.2 Threading support on Blue Gene/P ..................................................... 18
  2.3 Message Passing Interface on Blue Gene/P ............................................... 18
  2.4 Memory considerations ............................................................................. 18
    2.4.1 Memory leaks ..................................................................................... 22
    2.4.2 Memory management ......................................................................... 22
    2.4.3 Uninitialized pointers ........................................................................ 22
  2.5 Other considerations ................................................................................... 22
    2.5.1 Input/output ........................................................................................ 22
    2.5.2 Linking ............................................................................................... 23
  2.6 Compilers overview .................................................................................... 23
    2.6.1 Programming environment overview ............................................... 23
    2.6.2 GNU Compiler Collection .................................................................. 23
    2.6.3 IBM XL compilers .............................................................................. 24
  2.7 I/O node software ....................................................................................... 24

© Copyright IBM Corp. 2008. All rights reserved.
Personality of Blue Gene/P ......................................................... 298
Example of running personality on Blue Gene/P ...................... 298

Appendix D.  Porting applications .............................................. 301
Appendix E.  Mapping ............................................................... 303
Appendix F.  htcpartition ......................................................... 307

Appendix G.  Use of GNU profiling tool on Blue Gene/P .......... 309
Profiling with the GNU toolchain .............................................. 310
  Timer tick (machine instruction level) profiling .................... 310
  Procedure-level profiling with timer tick information .......... 310
  Full level of profiling ................................................. 310
Additional function in the Blue Gene/P gmon support .............. 310
Enabling/disabling profiling within your application ............. 311
Collecting the gmon data as a set of program counter values ... 311
Enabling profile data for threads in Blue Gene/P ................... 311
Enhancements to gprof in the Blue Gene/P toolchain .............. 311
  Using gprof to read gmon.sample.x files ......................... 311
  Using gprof to merge a large number of gmon.out.x files ..... 311

Appendix H.  Statement of completion .................................... 313

References .................................................................................. 315

Related publications ................................................................. 319
IBM Redbooks ......................................................................... 319
Other publications ................................................................... 319
Online resources ....................................................................... 321
How to get IBM Redbooks ...................................................... 322
Help from IBM ......................................................................... 322

Index ......................................................................................... 323
Notices

This information was developed for products and services offered in the U.S.A.

IBM might not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service might be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right might be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM might have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement might not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM might make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM might use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You might copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.
Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

AIX®
Blue Gene/L™
Blue Gene/P™
Blue Gene®
DB2 Universal Database™
DB2®
eServer™
General Parallel File System™
GPFS™
IBM®
LoadLeveler®
POWER™
POWER4™
POWER5™
POWER6™
PowerPC®
Redbooks®
Redbooks (logo) (®
System p™
Tivoli®

The following terms are trademarks of other companies:

Snapshot, and the NetApp logo are trademarks or registered trademarks of NetApp, Inc. in the U.S. and other countries.

SUSE, the Novell logo, and the N logo are registered trademarks of Novell, Inc. in the United States and other countries.

Intel, Intel logo, Intel Inside logo, and Intel Centrino logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names might be trademarks or service marks of others.
Preface

This IBM® Redbooks® publication is one in a series of IBM books written specifically for the IBM System Blue Gene/P Solution. The Blue Gene/P system is the second generation of a massively parallel supercomputer from IBM in the IBM System Blue Gene Solution series. This book provides an overview of the application development environment for the Blue Gene/P system. It is intended to help programmers understand the requirements to develop applications on this high-performance massively parallel supercomputer.

In this book, we explain instances where the Blue Gene/P system is unique in its programming environment. We also attempt to look at the differences between the IBM System Blue Gene/L Solution and the Blue Gene/P Solution. This book does not delve into great depth about the technologies that are commonly used in the supercomputing industry, such as Message Passing Interface (MPI) and Open Multi-Processing (OpenMP), nor do we try to teach parallel programming. References are provided in those instances for you to find more information if desired.

Prior to reading this book, you must have a strong background in high-performance computing (HPC) programming. The high-level programming languages that are used throughout this book are C/C++ and Fortran95. Previous experience using the Blue Gene/L system can help you understand better some concepts in this book that we do not extensively discuss. However, several IBM Redbooks publications about the Blue Gene/L system are available for you to obtain general information about the Blue Gene/L system. We recommend that you refer to “IBM Redbooks” on page 319 for a list of those publications.

The team that wrote this book

This book was produced in collaboration with the IBM Blue Gene developers at IBM Rochester, Minnesota, and IBM Blue Gene® developers at the IBM T. J. Watson Center in Yorktown Heights, N.Y. The information presented in this book is direct documentation of many of the Blue Gene/P hardware and software features. This information was published by the International Technical Support Organization, Rochester, MN.

Carlos Sosa is a Senior Technical Staff Member in the Blue Gene Development Group of IBM, where he has been the team lead of the Chemistry and Life Sciences high-performance effort since 2006. For the past 18 years, he has focused on scientific applications with emphasis in Life Sciences, parallel programming, benchmarking, and performance tuning. He received a Ph.D. degree in Physical Chemistry from Wayne State University and completed his post-doctoral work at the Pacific Northwest National Laboratory. His areas of interest are future IBM POWER™ architectures, Blue Gene, Cell Broadband, and cellular molecular biology.

We thank the following people and their teams for their contributions to this book:

- Tom Liebsch for being the lead source for hardware information
- Harold Rodakowski for software information
- Thomas M. Gooding for kernel information
- Michael Blokosome for parallel paradigms
- Michael T. Nelson and Lynn Boger for their help with the compiler
- Thomas A. Budnick for his assistance with APIs
- Brant L. Knudson and Paul Allen for their extensive contributions
We also thank the following people for their contributions to this project:

Gary Lakner
Gary Mullen-Schultz
ITSO, Rochester, MN

Dino Quintero
ITSO, Poughkeepsie, NY

Paul Allen
Mike Blocksome
Lynn Boger
Thomas A. Budnik
Ahmad Faraj
Thomas M. Gooding
Nicholas Goracke
Todd Ingle
Brant L. Knudson
Tom Liebsch
Mark Megerian
Sam Miller
Mike Mundy
Tom Musta
Mike Nelson
Jeff Parker
Joseph Ratterman
Richard Shok
Brian Smith
IBM Rochester

Philip Heidelberg
Sameer Kumar
Martin Ohmacht
James C. Sexton
Robert E. Walkup
Robert Wisniewski
IBM Watson Center

Mark Mendell
IBM Toronto

Ananthanaraya Sugavanam
Enci Zhong
IBM Poughkeepsie

Kirk Jordan
IBM Waltham

Jerrold Heyman
IBM Raleigh

Subba R. Bodda
IBM India
Become a published author

Join us for a two- to six-week residency program! Help write a book dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You will have the opportunity to team with IBM technical professionals, Business Partners, and Clients.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you will develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:  
ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks in one of the following ways:

- Use the online Contact us review Redbooks form found at:
  ibm.com/redbooks
- Send your comments in an e-mail to:
  redbooks@us.ibm.com
- Mail your comments to:
  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400
Summary of changes

This section describes the technical changes made in this edition of the book and in previous editions. This edition might also include minor corrections and editorial changes that are not identified.

Summary of Changes
for SG24-7287-01
for Blue Gene/P Application Development
as created or updated on September 2008

IBM System Blue Gene Solution: Blue Gene/P Application Development, Second Edition

This revision reflects the addition, deletion, or modification of new and changed information described in the following sections.

New information
- High Throughput Computing Chapter 12, “High-Throughput Computing (HTC) paradigm” on page 187)
- Documentation on htcpartition was added in Appendix F, “htcpartition” on page 307

Modified information
- The book was reorganized to included Part 4, “Job scheduler interfaces” on page 193. This section contains the Blue Gene/P™ APIs. Updates to the API chapters for HTC are included.
- Appendix E, “Mapping” on page 303 has been updated to reflect the predefined mapping for mpirun.
Part 1

Blue Gene/P: System and environment overview

Blue Gene/P is the next generation of massively parallel systems produced by IBM. It follows in the tradition established by the IBM Blue Gene/L™ Solution in challenging our thinking to take advantage of this innovative architecture. This next generation of supercomputers follows the winning formula provided as part of the Blue Gene/L Solution, that is, orders of magnitude in size and substantially more efficient in power consumption.

In this part, we present an overview of the two main topics of this book: hardware and software environment. This part includes the following chapters:

- Chapter 1, “Hardware overview” on page 3
- Chapter 2, “Software overview” on page 15
Hardware overview

In this chapter, we provide a brief overview of hardware. This chapter is intended for programmers who are interested in learning about the Blue Gene/P system. This chapter is also an overview for programmers who are already familiar with the Blue Gene/L system and want to understand the differences between the Blue Gene/L and Blue Gene/P systems.

It is important to understand where the Blue Gene/P system fits within the multiple systems that are currently available in the market. To gain a historical perspective as well as a perspective from an applications point of view, we recommend that you read the first chapter of the book *Unfolding the IBM eServer Blue Gene Solution*, SG24-6686. Although this book is written for the Blue Gene/L system, these concepts apply to the Blue Gene/P system.

In this chapter, we describe the Blue Gene/P architecture. We also provide an overview of the machine with a brief description of some of the components. Specifically we address the following topics:

- System architecture overview
- New on Blue Gene/P
- Microprocessor
- Compute nodes
- I/O nodes
- Networks
- Blue Gene/P programs
- Blue Gene specifications
- Host system
- Host system software
1.1 System architecture overview

The IBM System Blue Gene Solution is a revolutionary and important milestone for IBM in the high-performance computing arena. The Blue Gene/L system has been the fastest supercomputer in the last few years as noted by the TOP500 organization. Now IBM has introduced the Blue Gene/P system as the next-generation of massively parallel supercomputers, based on the same successful architecture in the Blue Gene/L system.

The Blue Gene/P system includes the following key features and attributes among others:

- Dense number of cores per rack: 4096 cores per rack
- PowerPC®, Book E compliant, 32-bit microprocessor, 850 MHz
- Double-precision, dual pipe floating-point acceleration on each core
- 24-inch/42U server rack air cooled
- Low power per flop ratio on Blue Gene/P compute application-specific integrated circuit (ASIC), 1.8 watts per GFlop/sec. per SOC
- Includes memory controllers, caches, network controllers, and high-speed input/output (I/O)
- Linux® kernel running on I/O nodes
- Message Passing Interface (MPI)² support between nodes via MPI library support
- Open Multi-Processing (OpenMP)³ application programming interface (API)
- Scalable control system based on external service node and front end node
- Standard IBM XL family of compilers support with XL C/C++, XLF, and GNU Compiler Collection⁵
- Software support for LoadLeveler®, General Parallel File System™ (GPFS™), and Engineering and Scientific Subroutine Library (ESSL)⁶

Figure 1-1 on page 5 illustrates the Blue Gene/P system architecture. It provides an overview of the multiple system components, from the microprocessor to the full system.

The system contains the following components:

**Chip**

The Blue Gene/P base component is a quad-core chip (also referred throughout this book as a compute node or node). The frequency of a single core is 850 MHz.

**Compute card**

One chip is soldered to a small processor card, one per card, together with memory (DRAM), to create a compute card (one node). The amount of DRAM per card is 2 GB.

**Compute node card**

The compute cards are plugged on a node card. These are two rows of sixteen compute cards on the card (planar). From zero to two I/O nodes per compute node card can be added to the compute node card.

**Rack**

A rack holds a total of 32 compute node cards.

**System**

A full petaflop system consists of 72 racks.
1.1.1 System buildup

Similar to the Blue Gene/L system, the number of cores in a system can be computed as follows:

Number of cores = (number of racks) x (number of node cards per rack) x (number of compute cards per node card) x (number of cores per compute card)

This equation corresponds to cores and memory. However, I/O is carried out through the I/O node that is connected externally via a 10 Gigabit Ethernet network. This network corresponds to the functional network. I/O nodes are not considered in the previous equation.

Finally, the compute and I/O nodes are connected externally (to the outside world) via the following peripherals:

- One service node
- One or more front end nodes
- Global file system

1.1.2 Compute and I/O nodes

Nodes are made of one quad-core with 2 GB of memory. These nodes do not have a local file system. Therefore, they must route I/O operations to an external device. To reach this external device (outside the environment), a compute node sends data to an I/O node, which in turn, carries out the I/O requests.

The hardware for both types of nodes is virtually identical. They only differ in the way that they are used. For example, extra RAM might be on the I/O nodes, and the physical connectors are different. A compute node runs a light, UNIX®-like proprietary kernel, referred as compute node kernel. The compute node kernel ships all network bound requests to the I/O node.
The I/O node is connected to the external device through an Ethernet port to the 10 gigabit functional network and can perform file I/O operations.

In the next section, we provide an overview of the Blue Gene environment, including all the components that fully populate the system.

1.1.3 Blue Gene/P environment

The Blue Gene/P environment consists of all the components that form part of the full system. Figure 1-2 illustrates the multiple components that form the Blue Gene/P environment.

The Blue Gene/P system consists of the following key components:

- **Service node**: This node provides control of the Blue Gene/P system.
- **Front end node**: This node provides access to the users to submit, compile, and build applications.
- **Compute node**: This node runs applications. Users cannot log on to this node.
- **I/O node**: This node provides access to external devices, and I/O requests are all routed through this node.
- **Functional network**: This network is used by all components of the Blue Gene/P system except the compute node.
- **Control network**: This network is the service network for specific system control functions between the service node and the I/O node.

In the remainder of this chapter, we describe these key components.

![Figure 1-2 Blue Gene/P environment](image-url)
1.2 New on Blue Gene/P

The Blue Gene/P Solution is a highly scalable multi-node supercomputer. Table 1-1 on page 8 shows key differences between the Blue Gene/L and Blue Gene/P systems. Each node consists of a single ASIC and forty 512 MB SDRAM-DDR2 memory chips. The nodes are interconnected through six networks, one of which connects the nearest neighbors into a three-dimensional (3D) torus or mesh. A system with 72 racks has a (x, y, z) 72 x 32 x 32 3D torus. The ASIC that is powering the nodes is in IBM CU-08 (CMOS9SF) system-on-a-chip technology and incorporates all of the compute and communication functionality needed by the core Blue Gene/P system. It contains 8 MiB of high-bandwidth embedded DRAM that can be accessed by the four cores in approximately 20 cycles for most L1 cache misses.

\[
\text{MiB: } 1 \text{ MiB} = 2^{20} \text{ bytes} = 1,048,576 \text{ bytes} = 1,024 \text{ kibibytes (a contraction of kilo binary byte)}
\]

The scalable unit of Blue Gene/P packaging consists of 512 compute nodes on a doubled-sided board, called a midplane, with dimensions of approximately 20 inches x 25 inches x 34 inches.

**Note:** This is the smallest unit that supports the full 3D torus.

Each node operates at Voltage Drain Drain (VDD) = 1.1v, 1.2v, or 1.3v, Temp. junction <70C, and a frequency of 850 MHz. Using an IBM PowerPC 450 processor and a single-instruction, multiple-data (SIMD), double-precision floating-point multiply add unit (double floating-point multiply add (FMA)), it can deliver four floating-point operations per cycle, or a theoretical maximum of 7.12 teraflops/sec. at peak performance for a single midplane. Two midplanes are contained within a single cabinet.

A midplane set of processing nodes, from a minimum of 16 to a maximum of 128, can be attached to a dedicated quad-processor I/O node for handling I/O communications to and from the compute nodes. The I/O node is assembled using the same ASIC as a compute node. Each Compute Node has a separate lightweight kernel, the Compute Node Kernel, which is designed for high-performance scientific and engineering code. With help from the I/O Node Kernel, the Compute Node Kernel provides Linux-like functionality to user applications. The I/O nodes run an embedded Linux operating system that is extended to contain additional system software functionality to handle communication with the external world and other services.

The I/O nodes of the Blue Gene/P system are connected to an external 10 Gigabit Ethernet switch, as previously mentioned, which provides I/O connectivity to file servers of a cluster-wide file system as illustrated in Figure 1-2 on page 6. The 10 gigabit Ethernet switch connects the Blue Gene/P system to the front end node and other computing resources. The front end node supports interactive logons, compiling, and overall system management.
Table 1-1 compares selected features between the Blue Gene/L and Blue Gene/P systems.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Blue Gene/L</th>
<th>Blue Gene/P</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Node</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cores per node</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Core clock speed</td>
<td>700 MHz</td>
<td>850 MHz</td>
</tr>
<tr>
<td>Cache coherency</td>
<td>Software managed</td>
<td>SMP</td>
</tr>
<tr>
<td>Private L1 cache</td>
<td>32 KB per core</td>
<td>32 KB per core</td>
</tr>
<tr>
<td>Private L2 cache</td>
<td>14 stream prefetching</td>
<td>14 stream prefetching</td>
</tr>
<tr>
<td>Shared L3 cache</td>
<td>4 MB</td>
<td>8 MB</td>
</tr>
<tr>
<td>Physical memory per node</td>
<td>512 MB - 1 GB</td>
<td>2 GB</td>
</tr>
<tr>
<td>Main memory bandwidth</td>
<td>5.6 GBps</td>
<td>13.6 GBps</td>
</tr>
<tr>
<td>Peak performance</td>
<td>5.6 GFlop/sec. per node</td>
<td>13.6 GFlop/sec. per node</td>
</tr>
<tr>
<td><strong>Network topologies</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Torus</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bandwidth</td>
<td>2.1 GBps</td>
<td>5.1 GBps</td>
</tr>
<tr>
<td>Hardware latency (nearest neighbor)</td>
<td>200 ns (32-byte packet) and 1.6 μs (256-byte packet)</td>
<td>100 ns (32-byte packet) and 800 ns (256-byte packet)</td>
</tr>
<tr>
<td><strong>Tree</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bandwidth</td>
<td>700 MBps</td>
<td>1.7 GBps</td>
</tr>
<tr>
<td>Hardware latency (round trip worst case)</td>
<td>5.0 μs</td>
<td>3.0 μs</td>
</tr>
<tr>
<td><strong>Full system</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Peak performance</td>
<td>410 TFlop/sec. (72 racks)</td>
<td>1 PFlop/Sec. (72 racks)</td>
</tr>
<tr>
<td>Power</td>
<td>1.7 MW (72 racks)</td>
<td>2.1 MW (72 racks)</td>
</tr>
</tbody>
</table>

Appendix A, “Blue Gene/P hardware-naming conventions” on page 287 provides an overview of how the Blue Gene/P hardware locations are assigned. Names are used consistently throughout both the hardware and software chapters. Understanding the naming convention is particularly useful when running applications on the Blue Gene/P system.

### 1.3 Microprocessor

The microprocessor is a PowerPC 450, Book E compliant, 32-bit microprocessor with a clock speed of 850 MHz. The PowerPC 450 microprocessor, with double-precision floating-point multiply add unit (double FMA), can deliver four floating-point operations per cycle with 3.4 GFlop/sec. per core.
1.4 Compute nodes

The compute node contains four PowerPC 450 processors with 2 GB of shared RAM and run a lightweight kernel to execute user-mode applications only. Typically all four cores are used for computation either in dual node mode, virtual node mode, or symmetrical multiprocessing. (Chapter 4, “Execution process modes” on page 391 covers these different modes.) Data is moved to and from the I/O nodes over the global collective network. Figure 1-3 illustrates the components of a compute node.

Compute nodes consist of the following components:

- Four 850 MHz PowerPC 450 cores
- 2 GB RAM per node
- Six connections to the torus network at 3.4 Gbps per link
- Three connections to the global collective network at 6.8 Gbps per link
- Four connections to the global interrupt network
- One connection to the control network (JTAG)

![Figure 1-3 Blue Gene/P ASIC](image-url)
1.5 I/O nodes

I/O nodes run an embedded Linux kernel with minimal packages required to support a Network File System (NFS) client and Ethernet network connections. They act as a gateway for the compute nodes in their respective rack to the external world (see Figure 1-4). The I/O nodes present a subset of standard Linux operating interfaces to the user. The 10 gigabit Ethernet interface of the I/O nodes is connected to the core Ethernet switch.

The node cards have the following components among others:

- 850 MHz PowerPC 450 cores
- 2 GB DDR2 SDRAM
- One 10 gigabit Ethernet adapter connected to the 10 gigabit Ethernet network
- Three connections to the global collective network at 6.8 Gbps per link
- Four connections to the global interrupt network
- One connection to the control network (JTAG)

![Node Card](image)

**Figure 1-4 Blue Gene/P I/O node card**

1.6 Networks

Five networks are used for various tasks on the Blue Gene/P system:

- Three-dimensional torus: point-to-point

  The torus network is used for general-purpose, point-to-point message passing and multicast operations to a selected “class” of nodes. The topology is a three-dimensional torus constructed with point-to-point, serial links between routers embedded within the Blue Gene/P ASICs. Therefore, each ASIC has six nearest-neighbor connections, some of which can traverse relatively long cables. The target hardware bandwidth for each torus link is 425 MBps in each direction of the link for a total of 5.1 GBps bidirectional bandwidth per node. The three-dimensional torus network supports the following features:

  - Interconnection of all compute nodes (73,728 for a 72-rack system)
  - Virtual cut-through hardware routing
  - 3.4 Gbps on all 12 node links (5.1 GBps per node)
  - Communications backbone for computations
  - 1.7/3.8 TBps bisection bandwidth, 67 TBps total bandwidth
Global collective: global operations

The global collective network is a high-bandwidth, one-to-all network used for collective communication operations, such as broadcast and reductions, and to move process and application data from the I/O nodes to the compute nodes. Each Compute and I/O node has three links to the global collective network at 850 MBps per direction for a total of 5.1 GBps bidirectional bandwidth per node. Latency on the global collective network is less than 2 µs from the bottom to top of the collective, with an additional 2 µs latency to broadcast to all. The global collective network supports the following features:

- One-to-all broadcast functionality
- Reduction operations functionality
- 6.8 Gbps of bandwidth per link; latency of network traversal 2 µs
- 62 TBps total binary network bandwidth
- Interconnects all compute and I/O nodes (1088)

Global interrupt: low latency barriers and interrupts

The global interrupt network is a separate set of wires based on asynchronous logic, which forms another network that enables fast signaling of global interrupts and barriers (global AND or OR). Round-trip latency to perform a global barrier over this network for a 72 K node partition is approximately 1.3 microseconds.

10 gigabit Ethernet: file I/O and host interface

The 10 gigabit Ethernet (optical) network consists of all I/O nodes and discrete nodes that are connected to a standard 10 gigabit Ethernet switch. A Cisco switch is typically used because it can be configured as a non-blocking switch and is scalable from a one-frame to many-frame configuration without requiring additional switches. The compute nodes are not directly connected to this network. All traffic is passed from the compute node over the global collective network to the I/O node and then onto the 10 gigabit Ethernet network.

Control: boot, monitoring, and diagnostics

The control network consists of a JTAG interface to a 1 gigabit Ethernet interface with direct access to shared SRAM in every Compute and I/O node. The control network is used for system boot, debug, and monitoring. It allows the service node to provide run-time non-invasive reliability, availability, and serviceability (RAS) support as well as non-invasive access to performance counters.

1.7 Blue Gene/P programs

The Blue Gene/P software for the Blue Gene/P core rack includes the following programs:

- Compute node kernel
  MPI support for hardware implementation and abstract device interface, control system, and system diagnostics.
- Compute node kernel and services
  Provides an environment for execution of user processes. The services that are provided are process creation and management, memory management, process debugging and RAS management.
- I/O node kernel and services
  Provides file system access and sockets communication to applications executing in the compute node.
- GNU Compiler Collection Toolchain Patches (Blue Gene/P changes to support GNU Compiler Collection).
The system software that is provided with each Blue Gene/P core rack or racks includes the following programs:

- DB2® Universal Database™ Enterprise Server Edition: System administration and management
- Compilers: XL C/C++ Advanced Edition for Linux with OpenMP support and XLF (Fortran) Advanced Edition for Linux

### 1.8 Blue Gene specifications

Table 1-2 lists the features of the Blue Gene/P compute nodes and I/O nodes.

<table>
<thead>
<tr>
<th>Node properties</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Node processors (Compute and I/O)</td>
<td>Quad 450 PowerPC</td>
</tr>
<tr>
<td>Processor frequency</td>
<td>850 MHz</td>
</tr>
<tr>
<td>Coherency</td>
<td>Symmetrical multiprocessing</td>
</tr>
<tr>
<td>L1 Cache (private)</td>
<td>32 KB per core</td>
</tr>
<tr>
<td>L2 Cache (private)</td>
<td>14 stream prefetching</td>
</tr>
<tr>
<td>L3 Cache size (shared)</td>
<td>8 MB</td>
</tr>
<tr>
<td>Main store memory/node</td>
<td>2 GB</td>
</tr>
<tr>
<td>Main store memory bandwidth</td>
<td>16 GBps</td>
</tr>
<tr>
<td>Peak performance</td>
<td>13.6 GFlop/sec. (per node)</td>
</tr>
</tbody>
</table>

**Torus network**

<table>
<thead>
<tr>
<th>Bandwidth</th>
<th>6 GBps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware latency (nearest neighbor)</td>
<td>64 ns (32-byte packet), 512 ns (256-byte packet)</td>
</tr>
<tr>
<td>Hardware latency (worst case)</td>
<td>3 μs (64 hops)</td>
</tr>
</tbody>
</table>

**Global collective network**

<table>
<thead>
<tr>
<th>Bandwidth</th>
<th>2 GBps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware latency (round-trip worst case)</td>
<td>2.5 μs</td>
</tr>
</tbody>
</table>

**System properties (for 73,728 compute nodes)**

<table>
<thead>
<tr>
<th>Peak performance</th>
<th>1 PFlop/sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average/peak total power</td>
<td>1.8 MW/2.5 MW (25 kW/34 kW per rack)</td>
</tr>
</tbody>
</table>
1.9 Host system

In addition to the Blue Gene/P core racks, the host system shown in Figure 1-5 is required for a complete Blue Gene/P system. There is generally one host rack for the core Ethernet switch, service node, and front end node. It might also house the Hardware Management Console (HMC) control node, monitor, keyboard, KVM switch, terminal server, and Ethernet modules.

1.9.1 Service node

The service node performs many functions for the operation of the Blue Gene/P system, including system boot, machine partitioning, system performance measurements, and system health monitoring. The service node uses DB2 as the data repository for system and state information.

1.9.2 Front end nodes

The front end node provides interfaces for users to log on, compile their applications, and submit their jobs to run from these nodes. They have direct connections to both the Blue Gene/P internal VLAN and the public Ethernet networks.

1.9.3 Storage nodes

The Storage Nodes provide mass storage for the Blue Gene/P system. We recommend that the Storage Nodes run GPFS locally to provide a single unified file system name space to the Blue Gene/P system. However the I/O nodes access the GPFS file system over standard NFS mounts.

The storage rack generally contains the terminal server, Storage Nodes with RAM, Gigabit Ethernet adapters connected to the core Ethernet switch, and adapters connected to a hard disk drive (HDD).
1.10 Host system software

The operating system requires installation of SUSE® Linux Enterprise Server 10 (SLES10, 64 bit) on the service node and front end node.

The following software applications for high-performance computing are optionally available for the Blue Gene/P system:

- Cluster System Management V1.5
- File system: GPFS for Linux Server with NFS Client
- Job Scheduler: LoadLeveler for Blue Gene/P
- Engineering and Scientific Subroutine Library
- Application development tools for Blue Gene/P, which include debugging environments, application performance monitoring and tuning tools, and compilers.
Chapter 2. Software overview

In this chapter, we provide an overview of the software that runs on the Blue Gene/P system. As shown in Chapter 1, “Hardware overview” on page 3, the Blue Gene/P environment consists of Compute and I/O nodes. It also has an external set of systems where users can perform system administration and management, partition and job management, application development, and debugging. In this heterogeneous environment, software must be able to interact.

Specifically, we cover the following topics:
- Blue Gene/P software at a glance
- Compute node kernel
- Message Passing Interface on Blue Gene/P
- Memory considerations
- Other considerations
- Compilers overview
- I/O node software
- Management software
2.1 Blue Gene/P software at a glance

Blue Gene/P software includes the following key attributes among others:

- Full Linux kernel running on I/O nodes
- Proprietary kernel dedicated for the compute nodes
- Message Passing Interface (MPI)\(^9\) support between nodes via MPI library support
- Open Multi-Processing (OpenMP)\(^10\) application programming interface (API)
- Scalable control system based on an external service node and front end node
- Standard IBM XL family of compilers\(^{11}\) support with XLC/C++, XLF, and GNU Compiler Collection\(^{12}\)
- Software support that includes LoadLeveler\(^{13}\), GPFS\(^{14}\), and Engineering and Scientific Subroutine Library (ESSL)\(^{15}\)

From a software point of view, the Blue Gene/P system is comprised of the following components:

- Compute node
- I/O nodes
- Front end nodes where users compile and submit jobs
- Control management network
- Service node, which provides capabilities to manage jobs running in the racks
- Hardware in the racks

The front end node consists of the interactive resources on which users log on to access the Blue Gene/P system. Users edit and compile applications, create job control files, launch jobs on the Blue Gene/P system, post-process output, and perform other interactive activities.

An Ethernet switch is the main communication path for applications that run on the compute node to the external devices. This switch provides high-speed connectivity to the file system, which is the main disk storage for the Blue Gene/P system. This switch also gives other resources access to the files on the file system.

A Control and Management Network provides system administrators with a separate command and control path to the Blue Gene/P system. This private network is not available to unprivileged users.

The software for the Blue Gene/P system consists of the following integrated software subsystems:

- System administration and management
- Partition and job management
- Application development and debugging tools
- Compute node kernel and services
- I/O node kernel and services

The five software subsystems are required in three hardware subsystems:

- Host complex (including front end node and service node)
- I/O node
- Compute node
Figure 2-1 illustrates these components.

The software environment illustrated in Figure 2-1 relies on a series of header files and libraries. A selected set is listed in Appendix B, “Header files and libraries” on page 293.

### 2.2 Compute node kernel

The compute node kernel provides an environment for execution of user processes. Compute node kernel services include:

- Process creation and management
- Memory management
- Process debugging
- Reliability, availability and serviceability (RAS) management
- File I/O
- Network

The compute nodes on Blue Gene/P are implemented as quad cores on a single chip with 2 GB of dedicated physical memory in which applications run.

A process is executed on Blue Gene/P nodes in three main modes:

- Symmetrical Multiprocessing (SMP) Node Mode
- virtual node mode (VN)
- Dual Node Mode (DUAL)
Application programmers see the compute node kernel software as a Linux-like operating system. This type of operating system is accomplished on the Blue Gene/P software stack by providing a standard set of run-time libraries for C, C++, and Fortran95. To the extent that is possible, the supported functions maintain open standard POSIX-compliant interfaces. We discuss the compute node kernel further in Part 2, “Kernel overview” on page 29. Applications can access system calls that provide hardware or system features, as illustrated by the examples in Appendix C, “Files on architectural features” on page 297.

2.2.1 High-performance computing and High-Throughput Computing modes

On Blue Gene/P we refer to the parallel paradigms that rely on the network for communication, mainly via the Message-Passing Interface (MPI) as high-performance computing (HPC). This topic is discussed in Chapter 7, “Parallel paradigms” on page 67. Blue Gene/P also offers a paradigm where applications do not require communication between tasks and each node is running a different instance of the application. We referred to this paradigm as High-Throughput Computing (HTC). This topic is discussed in Chapter 12, “High-Throughput Computing (HTC) paradigm” on page 187.

2.2.2 Threading support on Blue Gene/P

The threading implementation on the Blue Gene/P system supports OpenMP. The XL OpenMP implementation provides a futex-compatible syscall interface, so that the Native POSIX Thread Library (NPTL) pthreads implementation in glibc runs without modification. These syscalls allow only a total of four threads, limited support for mmap, and testing only with usage behavior of OpenMP. The compute node kernel provides a special thread function for I/O handling in MPI.

Important: The compute node kernel supports the execution of one quad-threaded process, where each of the four cores in the Blue Gene/P node is assigned hard affinity to each of a maximum of four threads. The compute node kernel also supports the execution of four single-threaded processes per core on a node or two single-threaded process per two cores on a node.

2.3 Message Passing Interface on Blue Gene/P

The implementation of MPI on the Blue Gene/P system is the MPICH2 standard that was developed by Argonne National Labs. For more information about MPICH2, see the Message Passing Interface (MPI) standard Web site at:

http://www-unix.mcs.anl.gov/mpi/

A function of the MPI-2 standard that is not supported by Blue Gene/P is Dynamic Process Management (creating new MPI processes). However, the various thread modes are supported.

2.4 Memory considerations

On the Blue Gene/P system, the entire physical memory of a compute node is 2 GB. Of that space, some is allocated for the compute node kernel itself. In addition, shared memory space is also allocated to the user process at the time at which the process is created.
Chapter 2. Software overview

The compute node kernel keeps track of collisions of stack and heap as the heap is expanded via a `brk` syscall. The Blue Gene/P system includes stack guard pages.

The compute node kernel and its private data are protected from read/write by the user process or threads. The code space of the process is protected from writing by the process or threads. Code and read-only data are shared between the processes in virtual node mode unlike in the Blue Gene/L system.

In general, give careful consideration to memory when writing applications for the Blue Gene/P system. At the time at which this book was written, each node has 2 GB of physical memory, unlike the Blue Gene/L system.

As previously mentioned, memory addressing is an important topic in regard to the Blue Gene/P system. As previously mentioned, memory addressing is an important topic in regard to the Blue Gene/P system. When an application stores data in memory it can be classified as follows:

- **data**: Initialized static and common variables
- **bss**: Uninitialized static and common variables
- **heap**: Controlled allocatable arrays
- **stack**: Controlled automatic arrays and variables

You can use the Linux `size` command to gain an idea of the memory size of the program. However, the `size` command does not provide any information about the run-time memory usage of the application nor on the classification of the types of data. Figure 2-2 illustrates memory addressing in HPC based on the different node modes that are available on the Blue Gene/P system.

**Important**: In C, C++ and Fortran, the `malloc` routine returns a NULL pointer when users request more memory than the physical memory available. We recommend you always check `malloc` return values for validity.

The hardware issues a segment violation (SEGV) interrupt and terminates the application on all nodes in the partition when referencing data using a NULL pointer.
Table 2-1 compares the three modes based on the memory addressing program introduced in *Unfolding the IBM eServer Blue Gene Solution*, SG24-6686. This program is shown in Example 2-1 with the memory parameters used here.

### Table 2-1 Memory addressing as a function of the three Blue Gene/P node modes

<table>
<thead>
<tr>
<th>Parameters</th>
<th>SMP</th>
<th>DUAL</th>
<th>VN</th>
</tr>
</thead>
<tbody>
<tr>
<td>heapsize function address</td>
<td>10012c0</td>
<td>10012c0</td>
<td>10012c0</td>
</tr>
<tr>
<td>printf function address</td>
<td>1002434</td>
<td>1002434</td>
<td>1002434</td>
</tr>
<tr>
<td>end of code address</td>
<td>1067b38</td>
<td>1067b38</td>
<td>1067b38</td>
</tr>
<tr>
<td>variable initialized address</td>
<td>1101c98</td>
<td>1101c98</td>
<td>1101c98</td>
</tr>
<tr>
<td>end of data address</td>
<td>11023a8</td>
<td>11023a8</td>
<td>11023a8</td>
</tr>
<tr>
<td>start of bss address</td>
<td>11023a8</td>
<td>11023a8</td>
<td>11023a8</td>
</tr>
<tr>
<td>variable uninitialized address</td>
<td>11023a8</td>
<td>11023a8</td>
<td>11023a8</td>
</tr>
<tr>
<td>end of bss address</td>
<td>1103cf0</td>
<td>1103cf0</td>
<td>1103cf0</td>
</tr>
<tr>
<td>start of heap address</td>
<td>1600010</td>
<td>1600010</td>
<td>1600010</td>
</tr>
<tr>
<td>end of heap_array0 address</td>
<td>7e60000c</td>
<td>3fc0000c</td>
<td>2020000c</td>
</tr>
<tr>
<td>start of heap_array address</td>
<td>7e601010</td>
<td>3fc01010</td>
<td>20201010</td>
</tr>
<tr>
<td>end of heap_array address</td>
<td>7e70100c</td>
<td>3fd0100c</td>
<td>2030100c</td>
</tr>
<tr>
<td>end of heap address</td>
<td>1127000</td>
<td>1127000</td>
<td>1127000</td>
</tr>
<tr>
<td>heap size</td>
<td>144144 23310</td>
<td>144144 23310</td>
<td>144144 23310</td>
</tr>
<tr>
<td>start of stack address</td>
<td>7fffd31c</td>
<td>403fd31c</td>
<td>209fd31c</td>
</tr>
<tr>
<td>end of stack address</td>
<td>2fd320</td>
<td>1fd320</td>
<td>1fd320</td>
</tr>
</tbody>
</table>

The text section starts at address 0. The heap section begins from the bottom, after the data and bss sections. The stack section starts from the top, at address 7fffd31c in SMP Node Mode (approximately 2 GB) and at address 403fd31c in Dual Node Mode (approximately 1 GB) and 209fd31c in virtual node mode (approximately 512 MB).

### Example 2-1 Program for memory addressing

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>             // for 'brk ()' and 'sbrk ()'

extern int  _etext;             // end of code area
extern int  _edata;             // end of data area
extern int  __bss_start;        // start of bss array
extern int  _end;               // end of bss area
```

20 Blue Gene/P Application Development
#define SIZE 1024*256           // 1 MB of long
//#define SZ 2000                 // SMP node mode
//#define SZ 998                   // DUAL node mode
#define SZ 492                     // VN node mode

unsigned long heapsize ()
{
    return (unsigned long) sbrk (0) - (unsigned long) & _end;
}

void * gotostack ()
{
    long st[SZ * SIZE];
    st[0] = 123456;
    printf ("\nstart of stack address         %9lx
", &st[SZ * SIZE - 1]);
    printf ("end of stack address           %9lx\n\n", st);
}

int initialized = 123;          // goes to data area
int uninitialized;              // goes to bss
int main (int argc, char * argv [] )
{
    int loop;
    long long_integer;
    long * heap_array;
    long * heap_array0;

    errno = 0;
    if ((heap_array0 = (long *) malloc (SZ * SIZE * sizeof (long_integer))) == NULL)
        printf("error, could not allocate\n");
    if (errno != 0){
        printf ("malloc errno : %d\n", errno); errno = 0; }
    if ((heap_array = (long *) malloc (SIZE * sizeof(long_integer))) == NULL)
        printf("error, could not allocate\n");
    if (errno != 0){
        printf ("malloc errno : %d\n", errno); errno = 0; }

    printf ("Memory mapping\n\n");
    printf ("heapsize function address      %9lx\n", heapsize);
    printf ("printf function address        %9lx\n", printf);
    printf ("end of code address            %9lx\n", &_etext);
    printf ("variable initialized address   %9lx\n", &initialized);
    printf ("end of data address            %9lx\n", &_edata);
    printf ("start of bss address          %9lx\n", &__bss_start);
    printf ("variable uninitialized address %9lx\n", &uninitialized);
    printf ("end of bss address            %9lx\n", &_end);
    printf ("start of heap address         %9lx\n", heap_array0);
    printf ("end of heap_array0 address     %9lx\n", &heap_array0[SZ * SIZE - 1]);
    printf ("start of heap_array address    %9lx\n", heap_array);
    printf ("end of heap_array address      %9lx\n", &heap_array[SIZE - 1]);
    printf ("start of heap address         %9lx\n", sbrk (0));
    long_integer = heapsize();
    printf ("\nHeap size %lu %9lx\n", long_integer, long_integer);
    gotostack();
}
2.4.1 Memory leaks

Given that no virtual paging exists on the Blue Gene/P system, any memory leaks in your application can quickly consume available memory. When writing applications for the Blue Gene/P system, you must be especially diligent that you release all memory that you allocate. This is true on any machine. Most of the time, having an application running on multiple architectures helps identify this type of problem.

2.4.2 Memory management

The Blue Gene/P computer implements a 32-bit memory model. It does not support a 64-bit memory model, but provides large file support and 64-bit integers.

In the case of the Blue Gene/P system, if the memory requirement per MPI task is greater than 512 MB in virtual node mode or greater than 1 GB in Dual Node Mode, then the application will not run on the Blue Gene/P system. However, in SMP Node Mode, 2 GB of memory is available. The application works only if you take steps to reduce the memory footprint.

In some cases, you can reduce the memory requirement by distributing data that was replicated in the original code. In this case, additional communication might be needed. It might also be possible to reduce the memory footprint by being more careful about memory management in the application, such as by not defining arrays for the index that corresponds to the number of nodes.

2.4.3 Uninitialized pointers

Blue Gene/P applications run in the same address space as the compute node kernel and the communications buffers. You can create a pointer that does not reference your own application’s data, but rather references the area used for communications. The compute node kernel itself is well protected from rogue pointers.

2.5 Other considerations

It is important to understand that the operating system present on the compute node, the compute node kernel, is not a full-fledged version of Linux. Because of this, you must use care in some areas, as explained in the following sections, when writing applications for the Blue Gene/P system. For a full list of supported system calls, see Chapter 6, “System calls” on page 53.

2.5.1 Input/output

I/O is an area where you must pay special attention in your application. The compute node kernel does not perform I/O. This is carried out by the I/O node.

File I/O
A limited set of file I/O is supported. Do not attempt to use asynchronous file I/O because it results in run-time errors.

Standard input
Standard input (stdin) is supported on the Blue Gene/P system.
Sockets calls
Sockets are supported on the Blue Gene/P system. For additional information, see Chapter 6, “System calls” on page 53.

2.5.2 Linking
Dynamic linking is not supported on the Blue Gene/L system. However, it is supported on the Blue Gene/P system. You can now statically link all code into your application or use dynamic linking.

2.6 Compilers overview
Read-only sections are supported in the Blue Gene/P system. However, this is not true of read-only sections within dynamically located modules.

2.6.1 Programming environment overview
The diagram in Figure 2-3 provides a quick view into the software stack that supports the execution of Blue Gene/P applications.

![Software stack diagram]

**Figure 2-3**  Software stack supporting the execution of Blue Gene/P applications

2.6.2 GNU Compiler Collection
The standard GNU Compiler Collection V4.1.1 for C, C++, and Fortran is supported on the Blue Gene/P system. The current versions are as follows:

- gcc 4.1.2.
- binutils 2.17.
- glibc 2.4.

You can find the GNU Compiler Collection in the /bgsys/drivers/ppcfloor/gnu-linux/bin directory. For more information, see Chapter 8, “Developing applications with IBM XL compilers” on page 95.
2.6.3 IBM XL compilers

The following IBM XL compilers are supported for developing Blue Gene/P applications:

- XL C/C++ Advanced Edition V9.0 for Blue Gene/P
- XL Fortran Advanced Edition V11.1 for Blue Gene/P

See Chapter 8, “Developing applications with IBM XL compilers” on page 95, for more compiler-related information.

2.7 I/O node software

The Blue Gene/P system is a massively parallel system with a large number of nodes. Compute nodes are reserved for computations, and I/O is carried out via the I/O nodes. These nodes serve as links between the compute nodes and external devices. For instance, applications running on compute nodes can access file servers and communicate with processes in other machines.

The I/O node software and the service node software communicate to exchange various data relating to machine configuration and workload. Communications use a key-based authentication mechanism with keys using at least 256 bits.

The I/O node kernel is a standard Linux kernel and provides file system access and sockets communication to applications that execute on the compute nodes.

2.7.1 I/O node kernel boot considerations

The I/O node kernel is designed to be booted as infrequently as possible due to the numerous possible failures of mounting remote file systems. The bootstrap process involves loading a ramdisk image and booting the Linux kernel. The ramdisk image is extracted to provide the initial file system, which contains minimal commands to mount the file system on the service node via the Network File System (NFS). The boot continues by running startup scripts from the NFS and running customer-supplied startup scripts to perform site-specific actions such as logging configuration and mounting high-performance file systems.

The Blue Gene/P system has considerably more content over the Blue Gene/L system in the ramdisk image to reduce the load on the service node exported by the NFS as the I/O node boot. Toolchain shared libraries and all of the basic Linux text and shell utilities are local to the ramdisk. Packages, such as GPFS, and customer-provided scripts are NFS mounted for administrative convenience.

2.7.2 I/O node file system services

The I/O node kernel supports an NFS client or GPFS client, which provides a file system service to application processes that execute on its associated compute node. The NFSv3 and GPFS file systems supported as part of the Blue Gene/L system continue with the Blue Gene/P system. As with the Blue Gene/L system, customers can still add their own parallel file systems by modifying Linux on the I/O node as needed.
2.7.3 Socket services for the compute node kernel

The I/O node includes a complete Internet Protocol (IP) stack, with TCP and UDP services. A subset of these services is available to user processes running on the compute node that is associated with an I/O node. Application processes communicate with processes running on other systems using client-side sockets via standard socket permissions and network connectivity. In addition, server-side sockets are available.

Note that the I/O node implements the sockets so that all the compute nodes within a processor set (pset) behave as though the compute tasks are executing on the I/O node. In particular, this means that the socket port number is a single address space within the pset and they share the IP address of the I/O node.

2.7.4 I/O node daemons

The I/O node includes the following daemons:

- Control and I/O daemon
- File system client daemons
- Syslog
- sshd
- ntpd on at least one I/O node

2.7.5 Control system

The control system retains the high-level components from the Blue Gene/L system with a considerable change in low-level components to accommodate the updated control hardware in the Blue Gene/P system as well as to increase performance for the monitoring system. The MMCS server and mcserver are now the processes that make up the control system on the Blue Gene/P system.

- The Midplane Management Control System (MMCS; console and server) is similar to the Blue Gene/L system in the way it handles commands, interacts with DB2, boots blocks, and runs jobs.
- mcServer is the process through which MMCS makes contact with the hardware (replacing idoproxy of the Blue Gene/L system). mcServer handles all direct interaction with the hardware. Low-level boot operations are now part of this process and not part of MMCS.
- The standard mpirun command to launch jobs can be used from any front end node or the service node. This command is often invoked automatically from a higher level scheduler.

The components that reside on the service node contain the following functions:

- Bridge APIs
  A scheduler that dynamically creates and controls blocks typically uses Bridge APIs. A range of scheduler options includes ignoring these APIs and using mpirun on statically created blocks to full dynamic creation of blocks with pass-through midplanes. The Blue Gene/P system includes a new set of APIs that notifies the caller of any changes in real time. Callers can register for various entities (entities are jobs, blocks, node cards, midplanes, and switches) and only see the changes. Callers can also set filters so that notifications occur for only specific jobs or blocks.
- ciodb
  ciodb is now integrated as part of the MMCS server for the Blue Gene/P system. This is different from the Blue Gene/L system. ciodb is responsible for launching jobs on already
booted blocks. Communication to ciodb occurs via the database and can be initiated by either \texttt{mpirun} or the Bridge APIs.

- **MMCS**
  The MMCS daemon is responsible for configuring and booting blocks. It can be controlled either via a special console interface (similar to the Blue Gene/L system) or via the Bridge APIs. The mmcs daemon also is responsible for relaying RAS information into the RAS database.

- **mcServer**
  The mcServer daemon has low-level control of the system, which includes a parallel efficient environmental monitoring capability as well as a parallel efficient reset and code load capability for configuring and booting blocks on the system. The diagnostics for the Blue Gene/P system directly leverage this daemon for greatly improved diagnostic performance over that of the Blue Gene/L system.

- **bgpmaster**
  The bgpmaster daemon monitors the other daemons and restarts any failed components automatically.

- **Service actions**
  Service actions are a suite of administrative shell commands that are used to service hardware. They are divided into device-specific actions with a “begin” and “end” action. Typically the “begin” action powers down hardware so it can be removed from the system, and the “end” action powers up the replacement hardware. The databases are updated with these operations, and they coordinate automatically with the scheduling system as well as the diagnostic system.

- **The submit server daemon**
  The submit daemon is the central resource manager for High-Throughput Computing (HTC) partitions. When MMCS boots a partition in HTC mode, each partition registers itself with the submit server before going to initialized state. This registration process includes several pieces of information, such as partition mode (SMP, DUAL, VN), partition size, user who booted the partition, and the list of users who can run on the partition. This information is maintained in a container and is used to match resource requests from submit commands based on their job requirements.

- **mpirund**
  The mpirund is a daemon process running on the service node whose purpose is to handle connections from frontend mpirun processes, and fork backend mpirun processes.

- **Real-time server actions**
  The Real-time Notification APIs are designed to eliminate the need for a resource management system to constantly have to read in all of the machine state to detect changes. The APIs allow the caller to be notified in real-time of state changes to jobs, blocks, and hardware, such as base partitions, switches, and node cards. After a resource management application has obtained an initial snapshot of the machine state using the Bridge APIs, the Bridge APIs can then determine to be notified only of changes, and the Real-time Notification APIs provides that mechanism.
2.8 Management software

The Blue Gene/P management software is based on a set of databases that run on the service node. The database software is DB2.

2.8.1 Midplane Management Control System

Both Blue Gene/P hardware and software are controlled and managed by the MMCS. The service node, front end node, and the file servers are not under the control of the MMCS. The MMCS currently consists of several functions that interact with a DB2 database running on the service node.
Kernel overview

The kernel provides the glue that makes all components in Blue Gene/P work together. In this part, we provide only an overview of the kernel functionality for applications developers. This part is for those who require information about system-related calls and interaction with the kernel.

This part contains the following chapters:
- Chapter 3, “Kernel functionality” on page 31
- Chapter 4, “Execution process modes” on page 39
- Chapter 5, “Memory” on page 45
- Chapter 6, “System calls” on page 53
Kernel functionality

In this chapter, we provide an overview of the functionality implemented as part of the compute node kernel and I/O node Kernel. We discuss the following topics:

- System software overview
- Compute node kernel
- I/O node kernel
3.1 System software overview

In general, the function of the kernel is to enable applications to run on a particular hardware system. This enablement consists of providing such services as applications execution, file I/O, memory allocation, and many others. In the case of the Blue Gene/P system, the system software provides two kernels:

- Compute node kernel
- I/O node kernel

3.2 Compute node kernel

The kernel that runs on the compute node is called the compute node kernel and is IBM proprietary. It has a subset of the Linux system calls. The compute node kernel is a flexible, lightweight kernel for Blue Gene/P compute nodes that are capable of supporting both diagnostic modes and user applications.

The Compute Node Kernel is intended to be a Linux-like operating system, from the application point of view, supporting a large subset of Linux compatible system calls. This subset is taken from the subset used successfully on the Blue Gene/L system, which demonstrates good compatibility and portability with Linux.

Now, as part of the Blue Gene/P system, the compute node kernel supports threads and dynamic linking for further compatibility with Linux. The compute node kernel has been tuned for the capabilities and performance of the Blue Gene/P System. In Figure 3-1, you see the interaction between the application space and the kernel space.

![Figure 3-1  Compute node kernel overview](image)
When running a user application, the compute node kernel connects to the I/O node via the collective network. This connection communicates to a process running on the Linux I/O node called the control and I/O daemon (CIOD). All function-shipped system calls are forwarded to the CIOD process and executed on the I/O node.

At the user-application level, the compute node kernel supports the following APIs among others:

- Message Passing Interface (MPI)\(^{17}\) support between nodes via MPI library support
- Open Multi-Processing (OpenMP)\(^{18}\) API
- Standard IBM XL family of compilers support with XLC/C++, XLF, and GNU Compiler Collection\(^{19}\)
- Highly optimized mathematical libraries such as IBM Engineering and Scientific Subroutine Library (ESSL)\(^{20}\)
- GNU Compiler Collection (GCC) C Library, or glibc, which is the C standard library and interface of GCC for a provider library plugging into an other library (system programming interfaces (SPIs))

The following services are some of those that are provided by the compute node kernel:

- Torus direct memory access (DMA)\(^{21}\), which provides memory access for reading, writing, or doing both independently of the processing unit
- Shared-memory access on a local node
- Hardware configuration
- Memory management
- MPI topology
- File I/O
- Sockets connection
- Signals
- Thread management
- Transport layer via collective network

### 3.2.1 Boot sequence of a compute node

The Blue Gene/P hardware is a stateless system. When power is initially applied, the hardware must be externally initialized. Given the architectural and reliability improvements in the Blue Gene/P design, reset of the compute nodes should be an infrequent event.

The following procedure explains how to boot a compute node as part of the main partition. Independent reset of a single compute node and independent reset of a single I/O node are different procedures.

The compute node kernel must be loaded into memory after every reset of the compute node. To accomplish this task, several steps must occur to prepare a compute node kernel for running an application:

1. The control system loads a small bootloader into SRAM.
2. The control system loads the personality into SRAM. The personality is a data structure that contains node-specific information, such as the X, Y, Z coordinates of the node.
3. The control system releases the compute node from reset.
4. The bootloader starts executing and initializes the hardware.
5. The bootloader communicates with the control system over the mailbox to load Common Node Services and compute node kernel images.
6. The bootloader then transfers control to the Common Node Services.
7. Common Node Services performs its setup and then transfers control to the compute node kernel.
8. The compute node kernel performs its setup and communicates to the CIOD.

At this point, the compute node kernel submits the job to the CIOD.

### 3.2.2 Common node services

Common node services provide low-level services that are both specific to the Blue Gene/P system and common to the Linux and the compute node kernel. As such, these services provide a consistent implementation across node types while insulating the kernels from the details of the control system.

The common node services provide the same low-level hardware initialization and setup interfaces to both Linux and the compute node kernel.

The common node services provide the following services:
- Access to the SRAM mailbox for performing I/O operations over the service network to the console
- Initialization of hardware for various networks
- Access to the personality
- Low-level services for RAS, including both event reporting and recovery handling
- Access to the Blue Gene interrupt controller

### 3.3 I/O node kernel

The kernel of the I/O node (see Figure 3-2 on page 35) is referred to as the **Mini-Control Program (MCP)**. It is a port of the Linux Kernel, which means it is GPL/LGPL licensed. It is similar to the Blue Gene/L I/O node. The I/O node kernel on the Blue Gene/P system has the following characteristics:
- **Embedded Linux Kernel**
  - Linux Version 2.6.16
  - Four-way symmetrical multiprocessing (SMP)
  - Paging disabled (no swapping available)
- **Ethernet**
  - 10 Gigabit Ethernet driver
  - Large Maximum Transmission Unit (MTU) support, which allows Ethernet frames to be increased from the default value of 1500 bytes to 9000 bytes
– TCP checksum offload engine support
– Availability of /proc files for configuring and gathering status

File systems supported
– Network File System (NFS)
– Parallel Virtual File System (PVFS)
– General Parallel File System (GPFS)
– Lustre File System

CIOD
– Lightweight proxy between compute nodes and the outside world
– Debugger access into the compute nodes
– SMP support

Figure 3-2  I/O node kernel overview

The I/O service is provided to the compute nodes from the compute node I/O proxy, which is started by the initialization script during the boot procedure of the MCP. The compute node I/O proxy is a user-level process that controls and services applications in the compute node and interacts with the Midplane Management and Control System (MMCS).

3.3.1 Control and I/O daemon

The CIOD serves the following roles:

● Interface to and from the control system
● Proxy for the compute node
● Proxy for the debug server

Note: To access this functionality, use Telnet to connect to the I/O node on port 9000 and type help for a list of commands.
The CIOD for the Blue Gene/P system includes the following major changes:

- Single process to many processes:
  - Takes advantage of four-way SMP
  - Cleans up tracking of compute node
- CIOMStream (control messages from the control system)
- DataStream (stdout/stderr/stdin messages)
- CIOP protocol (interface to compute node kernel)
- Support for tool daemons
CIOD threading architecture

CIOD reads from the collective network and places the message into the shared memory dedicated to the sending node I/O proxy. Figure 3-4 shows the threading architecture of the CIOD.

Figure 3-4  CIOD threading architecture
Execution process modes

The compute nodes on Blue Gene/P are implemented as quad-cores on a single chip with 2 GB of dedicated physical memory in which applications run. The three main modes in which a process is executed on Blue Gene/P nodes are as follows:

- Symmetrical Multiprocessing (SMP) node mode
- virtual node mode (VN)
- Dual node mode (DUAL)

In this chapter, we explore these modes in detail. The Blue Gene/L system has only two node modes, which are the coprocessor node mode and virtual node mode.
4.1 Symmetrical Multiprocessing Node Mode

In the default mode of operation of the Blue Gene/P system, the SMP node mode, each physical compute node executes a single task (Message Passing Interface (MPI) task) per node with a maximum of four threads. The Blue Gene/P system software treats those four cores in a compute node symmetrically. This mode is referred as *SMP mode (1X4)*, where 1 corresponds to one task and 4 corresponds to four threads.

In Figure 4-1, you see the interaction of this mode between the application space and the kernel space. The task or process can have up to four threads. pthreads and OpenMP are supported in this mode. In this mode, each thread is pinned to a processor.

![Figure 4-1 SMP node mode](image)

4.2 virtual node mode

The compute node kernel in the compute nodes also supports a virtual node mode of operation for the machine. In this mode, the kernel runs four separate processes on each compute node. Node resources (primarily the memory and the torus network) are shared by all processes. This mode is referred as *VN mode (4X1)*, where 4 corresponds to four tasks and 1 corresponds to one thread. In Figure 4-2 on page 41, virtual node mode is illustrated with four tasks per node and one thread per process. Shared memory is available between processes.

*Note*: Shared memory is available only in HPC mode and not in HTC mode.

In virtual node mode, an MPI application can use any of the cores in a node simply by quadrupling its number of MPI tasks. The now distinct MPI tasks running on four cores of a compute node have to communicate with each other. This communication is done...
transparently via direct memory access (DMA) on the node. DMA puts data destined for a physically different node on the torus, while it locally copies data when it is destined for the same physical node.

In virtual node mode, the four cores of a compute node act as different processes. Each has its own rank in the message layer. The message layer supports virtual node mode by providing a correct torus to rank mapping and first in, first out (FIFO) pinning in this mode. The hardware FIFOs are shared equally between the processes. Torus coordinates are expressed by quadruplets instead of triplets. In virtual node mode, communication between the four threads in a compute node is done via DMA local copies.

Each virtual node executes one compute process. Processes allocated in the same compute node share memory, which can be reserved at job launch. An application that wants to run with four tasks per node can dedicate a large portion for shared memory if the tasks need to share global data. This data can be read/write, and data coherency is handled in hardware.

The Blue Gene/P MPI implementation supports virtual node mode operations by sharing the systems communications resources of a physical compute node between the four compute processes that execute on that physical node. The low-level communications library of the Blue Gene/P system, that is the message layer, virtualizes these communications resources into logical units that each process can use independently.
4.3 Dual node mode

A new mode in the Blue Gene/P system is the dual node mode (DUAL). In this mode, each physical compute node executes two tasks per node with a maximum of four threads (two per task). Each task in dual node mode gets half the memory and cores, so that it can run two threads per task. This mode is referred as DUAL mode (2X2), where the first 2 corresponds to two tasks and the second 2 corresponds to two threads.

In Figure 4-3, you see two processes per node. Each process can have up to two threads. OpenMP and pthreads are supported. Shared memory is available between processes. Threads are pinned to a processor.

4.4 Shared memory support

Shared memory is supported in dual node mode and virtual node mode process models. Shared-memory usage in the SMP node mode is excluded because each processor already has access to all of the node’s memory.

Shared memory is allocated via standard Linux methods (`shm_open` and `mmap`). However, because the compute node kernel does not have virtual pages, the physical memory that backs the shared memory must come out of a memory region that is dedicated for shared memory. This memory region has its size fixed at job launch.

**BG_SHAREDMEMPOOLSIZE:** The BG_SHAREDMEMPOOLSIZE environmental variable specifies in MB the amount of memory to be allocated. This can be done via the `mpirun -env` flag, for example, `BG_SHAREDMEMPOOLSIZE=8`. This allocates 8 MB of shared memory storage.
The user can change the amount of memory to be set aside for this memory region at job launch. Figure 4-4 illustrates shared-memory allocation.

```c
fd = shm_open( SHM_FILE, O_RDWR, 0600 );
ftruncate( fds[0], MAX_SHARED_SIZE );
shmptr1 = mmap( NULL, MAX_SHARED_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

Figure 4-4  Shared-memory allocation

Figure 4-5 illustrates shared-memory deallocation.

```c
munmap(shmptr1, MAX_SHARED_SIZE);
close(fd);
shm_unlink(SHM_FILE);
```

Figure 4-5  shared-memory deallocation

The `shm_open()` and `shm_unlink()` routines access a pseudo-device, `/dev/shm/filename`, which the kernel interprets. Because multiple processes can access or close the shared-memory file, allocation and deallocation are tracked by a simple reference count. As such, the processes do not need to coordinate deallocation of the shared memory region.

### 4.5 Deciding which mode to use

The choice of the node mode largely depends on the type of application and the parallel paradigm that has been implemented for a particular application. The obvious case involves applications where a hybrid paradigm between MPI and OpenMP has been implemented. In this case, it is beneficial to use the SMP node mode. If you are writing single-threaded applications, you should consider virtual node mode.

I/O-intensive tasks that require a relatively large amount of data interchange between compute nodes benefit more by using virtual node mode. Those applications that are primarily CPU bound and do not have large working memory requirements (the application gets only half of the node memory), run more quickly in virtual node mode.

Finally, the HTC mode offers the possibility of running multiple instances on an application that does not require communication between nodes. This mode also allows running applications on single nodes and permits other users to share single nodes in the same partition. See Chapter 12, “High-Throughput Computing (HTC) paradigm” on page 187 for more information.

### 4.6 Specifying a mode

The default mode for `mpirun` is the SMP. To specify virtual node mode and dual node mode, you use the following commands:

```
mpirun ... -mode VN ...
mpirun ... -mode DUAL ...
```

See Chapter 11, “mpirun” on page 167 for more information about the `mpirun` command.

In the case of the `submit` command, you use the following commands:

```
submit ... -mode VN ...
submit ... -mode DUAL ...
```
Memory

In this chapter, we provide an overview of the memory subsystem and explain how it relates to the compute node kernel. This chapter includes the following topics:

- Memory overview
- Memory management
- Memory protection
- Persistent memory
5.1 Memory overview

Similar to the Blue Gene/L system, Blue Gene/P nodes support virtual memory. Memory is laid out as a single, flat, fixed-size virtual address space shared between the operating system kernel and the application program.

The Blue Gene/P system is a distributed-memory supercomputer, which includes an on-chip cache hierarchy, and memory is off-chip. It contains optimized on-chip symmetrical multiprocessing (SMP) support for locking and communication between the four ASIC processors.

The aggregate memory of the total machine is distributed in the style of a multi-computer, with no hardware sharing between nodes. The total physical memory amount supported is 2 GB per compute node.

The first level (L1) cache is contained within the PowerPC 450 core (see Figure 1-3 on page 9). The PowerPC 450 L1 cache is 64-way set associative.

The second level (L2R and L2W) caches, one dedicated per core, are 2 KB in size. They are fully associative and coherent. They act as prefetch and write-back buffers for L1 data. The L2 cache line is 128 bytes in size. Each L2 cache has one connection toward the L1 instruction cache running at full processor frequency. Each L2 cache also has two connections toward the L1 data cache, one for the writes and one for the loads, each running at full processor frequency. Read and write are 16 bytes wide.

The third level (L3) cache is 8-way set associative, 8 MB in size, with 128-byte lines. Both banks can be accessed by all processor cores. The L3 cache has three write queues and three read queues: one for each processor core and one for the 10 gigabit network. Ethernet and direct memory access (DMA) share the L3 ports. Only one unit can use the port at a time. The compute nodes use DMA, and the I/O nodes use Ethernet. The last one is used on the I/O node and for torus network DMA on the Compute Networks. All the write queues go across a four-line write buffer to access the eDRAM bank. Each of the two L3 banks implements thirty 128-byte-wide write combining buffers, for a total of sixty 128-byte-wide write combining buffers per chip.

Table 5-1 provides an overview of some of the features of different memory components.

<table>
<thead>
<tr>
<th>Cache</th>
<th>Total per node</th>
<th>Size</th>
<th>Replacement policy</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 instruction</td>
<td>4</td>
<td>32 KB</td>
<td>Round-Robin</td>
<td>➤ 64-way set-associative</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>➤ 16 sets</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>➤ 32-byte line size</td>
</tr>
<tr>
<td>L1 data</td>
<td>4</td>
<td>32 KB</td>
<td>Round-Robin</td>
<td>➤ 64-way set-associative</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>➤ 16 sets</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>➤ 32-byte line size</td>
</tr>
<tr>
<td>L2 prefetch</td>
<td>4</td>
<td>14 x 256 bytes</td>
<td>Round-Robin</td>
<td>➤ Fully associative (15-way)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>➤ 128-byte line size</td>
</tr>
<tr>
<td>L3</td>
<td>2</td>
<td>2 x 4 MB</td>
<td>Least recently used</td>
<td>➤ 8-way associative</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>➤ 2 bank interleaved</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>➤ 128-byte line size</td>
</tr>
<tr>
<td>Double data RAM (DDR)</td>
<td>2</td>
<td>➤ Minimum 2 x 512 MB ➤ Maximum 4 GB</td>
<td>N/A</td>
<td>➤ 128-byte line size</td>
</tr>
</tbody>
</table>
5.2 Memory management

You must give careful consideration to managing memory on the Blue Gene/P system. This is particularly true in order to achieve optimal performance. The memory subsystem of Blue Gene/P nodes has specific characteristics and limitations that the programmer should know about.

5.2.1 L1 cache

On the Blue Gene/P system, the PowerPC 450 internal L1 cache does not have automatic prefetching. Explicit cache touch instructions are supported. Although the L1 instruction cache was designed with support for prefetches, it was disabled for efficiency reasons.

Figure 1-3 on page 9 shows the L1 caches in the PowerPC 450 architecture. The size of the L1 cache line is 32 bytes. The L1 cache has two buses toward the L2 cache: one for the stores and one for the loads. The buses are 128 bits in width and run at full processor frequency. The theoretical limit is 16 bytes per cycle. However, 4.6 bytes is achieved on L1 load misses, and 5.6 bytes is achieved on all stores (write through). This value of 5.6 bytes is achieved for the stores but not for the loads. The L1 cache has only a three-line fetch buffer. Therefore, there are only three outstanding L1 cache line requests. The fourth one waits for the first one to complete before it can be sent.

The L1 hit latency is four cycles for floating point and three cycles for integer. The L2 hit latency is at about 12 cycles for floating point and 11 cycles for integer. The 4.6-byte throughput limitation is a result of the limited number of line fill buffers, L2 hit latency, the policy when a line fill buffer commits its data to L1, and the penalty of delayed load confirmation when running fully recoverable.

Because only three outstanding L1 cache line load requests can occur at the same time, at most three cache lines can be obtained every 18 cycles. The maximum memory bandwidth is three times 32 bytes divided by 18 cycles, which yields 5.3 bytes per cycle, which written as an equation looks like this:

\[
\frac{(3 \times 32 \text{ bytes})}{18 \text{ cycles}} = 5.3 \text{ bytes per cycle}
\]

Important:

- Avoid instructions when prefetching data in the L1 cache on the Blue Gene/P system. Using the processor, you can concurrently fill in three L1 cache lines. Therefore, it is mandatory to reduce the number of prefetching streams to three or less.

  To optimize the floating-point units (FPUs) and feed the floating-point registers, you can use the XL compiler directives or assembler instructions (dcbt) to prefetch data in the L1 data cache. The applications that are specially tuned for IBM POWER4™ or POWER5™ processors that take advantage of four or eight prefetching engines will choke the memory subsystem of the Blue Gene/P processor.

- To take advantage of the single-instruction, multiple-data (SIMD) instructions, it is essential to keep the data in the L1 cache as much as possible. Without an intensive reuse of data from the L1 cache and the registers, because of the number of registers, the memory subsystem is unable to feed the double FPU and provide two multiply-addition operations per cycle.
In the worst case, SIMD instructions can hurt the global performance of the application. For that reason, we advise that you disable the SIMD instructions in the porting phase by compiling with `-qarch=450`. Then recompile the code with `-qarch=450d` and analyze the performance impact of the SIMD instructions. Perform the analysis with a data set and a number of processors that is realistic in terms of memory usage.

**Optimization tips:**
- The optimization of the applications must be based on the 32 KB of the L1 cache.
- The benefits of the SIMD instructions can be canceled out if data does not fit in the L1 cache.

### 5.2.2 L2 cache

The L2 cache is the hardware layer that provides the link between the embedded cores and the Blue Gene/P devices, such as the 8 MB L3-eDRAM and the 32 KB SRAM. The 2 KB L2 cache line is 128 bytes in size. Each L2 cache is connected to one processor core.

The L2 design and architecture were created to provide optimal support for the PowerPC 450 cores for scientific applications. Thus, a logic for automatic sequential stream detection and prefetching to the L2 added on the PowerPC 440 is still available on PowerPC 450. The logic is optimized to perform best on sequential streams with increasing addresses. The L2 boosts overall performance for almost any application and does not require any special software provisions. It autonomously detects streams, issues the prefetch requests, and keeps the prefetched data coherent.

You can achieve latency and bandwidth results close to the theoretical limits (4.6 bytes per cycle) dictated by the PowerPC 450 core by doing careful programming. The L2 accelerates memory accesses for one to seven sequential streams.

### 5.2.3 L3 cache

The L3 cache is 8 MB in size. The line size is 128 bytes. Both banks are directly accessed by all processor cores and the 10 Gb network, only on the I/O node, and are used in compute nodes for torus DMA. There are three write queues and three read queues. The read queues directly access both banks.

Each L3 cache implements two sets of write buffer entries. Into each of the two sets, one 32-byte data line can deposit a set per cycle from any queue. In addition, one entry can be allocated for every cycle in each set. The write rate for random data is much higher in the Blue Gene/P system than in the Blue Gene/L system. The L3 cache can theoretically complete an aggregate of four write hits per chip every two cycles. However, banking conflicts reduce this number in most cases.

**Optimization tip:** Random access can divide the write sustained bandwidth of the L3 cache by a factor of three on compute nodes and more on I/O nodes.

### 5.2.4 Double data RAM

The theoretical memory bandwidth on a Blue Gene/P node to transfer a 128-byte line from the external DDR to the L3 cache is 16 cycles. Nevertheless, this bandwidth can only be sustained with sequential access. Random access can reduce bandwidth significantly.
Table 5-2 illustrates latency and bandwidth estimates for the Blue Gene/P system.

| Table 5-2  Latency and bandwidth estimates |
|-----------------|-----------------|
| Latency*        | Sustained bandwidth (bytes per cycle)**, ** |
| L1              | 3               | 8               |
| L2              | 11              | 4.6             |
| L3              | 50              | 4.6             |
| External DDR (single processor) | 104 | 40 |
| External DDR (dual processor) |             |                 |
| External DDR (triple processor) |             |                 |
| External DDR (quad processor) |             | 3.7             |

- This corresponds to integer load latency. Floating-point latency is one cycle higher.
- This is the maximum sustainable bandwidth for linear sequential access.
- Random access bandwidth is dependent on the access width and overlap access, respectively.

### 5.3 Memory protection

The PowerPC 450 processor has limited flexibility with regard to supported translation look-aside buffer (TLB) sizes and alignments. There is also a small number of TLB slots per processor. These limitations create situations where the dual goal of both static TLBs and memory protection is difficult to achieve with access to the entire memory space. This depends on the node’s memory configuration, process model, and size of the applications sections.

On the Blue Gene/P system, the compute node kernel reads only sections from the application. This prevents an application from accidentally corrupting its text (that is, its code) section due to an errant memory write. Additionally, the compute node kernel prevents an application from corrupting the compute node kernel text segments or any kernel data structures.

When a debugger is not attached to the running application, compute node kernel can protect the active thread’s stack using the data-address-compare debug registers in the PowerPC 450 processor. You can use this mechanism for stack protection without incurring TLB miss penalties. For the main thread, this protection is just above the maximum mmap() address. For a spawned thread, this protection is at the lower bound of the thread’s stack. This protection is not available when the debugger is being used because the debugger is managing those register settings.

The compute node kernel is strict in terms of TLB setup. For example, the compute node kernel does not create a 256 MB TLB that covers only 128 MB of real memory. By precisely creating the TLB map, any user-level page faults (also known as segfaults) are immediately caught.

In the default mode of operation of the Blue Gene/P system, which is SMP node mode, each physical compute node executes a single task per node with a maximum of four threads. The
Blue Gene/P system software treats those four core threads in a compute node symmetrically. Figure 5-1 illustrates how memory is accessed in SMP node mode. The user space is divided into user space “read, execute” and user space “read/write, execute”. The latter corresponds to global variables, stack, and heap. In this mode, the four threads have access to the global variables, stack, and heap.

Figure 5-1 Memory access protection in SMP node mode

Figure 5-2 shows how memory is accessed in virtual node mode. In this mode, the four core threads of a compute node act as different processes. The compute node kernel reads only sections of an application from local memory. No user access occurs between processes in the same node. User space is divided into user-space “read, execute” and user-space “read/write, execute”. The latter corresponds to global variables, stack, and heap. These two sections are designed to avoid data corruption.

Figure 5-2 Memory access protections in virtual node mode
Each task in dual node mode gets half the memory and cores so it can run two threads per task. Figure 5-3 shows that no user access occurs between the two processes. Although a layer of shared-memory per node and the user-space “read, execute” is common to the two tasks, the two user-spaces “read/write, execute” are local to each process.

Figure 5-3  Memory access protections in dual node mode

5.4 Persistent memory

Persistent memory is process memory that retains its contents from job to job. To allocate persistent memory, the environment variable BG_PERSISTMEMSIZE=X must be specified, where X is the number of 1024*1024 bytes to be allocated for use as persistent memory. In order for the persistent memory to be maintained across jobs, all job submissions must specify the same value for BG_PERSISTMEMSIZE. The contents of persistent memory can be re-initialized during job startup by either changing the value of BG_PERSISTMEMSIZE or by specifying the environment variable BG_PERSISTMEMRESET=1. The following new kernel function was added to support persistent memory:

persist_open()
System calls

System calls provide an interface between an application and the kernel. In this chapter, we provide information about the service points through which applications running on the compute node request services from the compute node kernel. This set of entry points into the compute node kernel is referred to as system calls (syscall). System calls on the Blue Gene/P system have substantially changed from system calls on the Blue Gene/L system. In this chapter, we describe system calls that are defined on the Blue Gene/P system.

We cover the following topics in this chapter:
- Compute node kernel
- Supported and unsupported system calls
- System programming interfaces
- Socket support
- Signal support

In general, the two types of system calls are:
- Local system calls
- Function-shipped system calls

Local system calls are handled by the compute node kernel only and provide Blue Gene/P-specific functionality. The following examples are of standard, local system calls:
- `brk()`
- `mmap()`
- `clone()`

Alternatively, function-shipped system calls are forwarded by the compute node kernel over the collective network to the control and I/O daemon (CIOD). The CIOD then executes those system calls on the I/O node and replies to the compute node kernel with the resultant data. Examples of function-shipped system calls are functions that manipulate files and socket calls.
6.1 Introduction to the compute node kernel

The role of the kernel on the compute node is to create an environment for the execution of a user process that is “Linux-like.” It is not a full Linux kernel implementation, but rather implements a subset of POSIX functionality.

The compute node kernel is a single-process operating system. It is designed to provide the services needed by applications that are expected to run on the Blue Gene/P system, but not services for all applications. The compute node kernel is not intended to run system administration functions from the compute node.

To achieve the best reliability, a small and simple kernel is a design goal. This enables a simpler checkpoint function. See Chapter 10, “Checkpoint and restart support for applications” on page 159.

Compute node application user: The compute node application never runs as the root user. In fact, it runs as the same user (uid) and group (gid) under which the job was submitted.

6.2 System calls

The compute node kernel system calls are divided into the following categories:

- File I/O
- Directory operations
- Time
- Process information
- Signals
- Miscellaneous
- Sockets
- Compute node kernel

6.2.1 Return codes

As is true for return codes on a standard Linux system, a return code of zero from a system call indicates success. A value of negative one (-1) indicates a failure. In this case, errno contains further information about exactly what caused the problem.
### 6.2.2 Supported system calls

Table 6-1 lists all the function prototypes for system calls by category that are supported on the Blue Gene/P system.

<table>
<thead>
<tr>
<th>Function prototype</th>
<th>Category</th>
<th>Header required</th>
<th>Description and type</th>
</tr>
</thead>
<tbody>
<tr>
<td>int access(const char *pathname, int mode);</td>
<td>File I/O</td>
<td>&lt;unistd.h&gt;</td>
<td>Determines the accessibility of a file; function-shipped to CIOD; mode: R_OK, X_OK, F_OK; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td>int chmod(const char *pathname, mode_t mode);</td>
<td>File I/O</td>
<td>&lt;sys/types.h&gt; &lt;sys/stat.h&gt;</td>
<td>Changes the access permissions on an already open file; function-shipped to CIOD; mode: S_ISUID, S_ISGID, S_ISVTX, S_IRWXU, S_IRUSR, S_IWUSR, S_IXUSR, S_IRWXG, S_IWGRP, S_IXGRP, S_IRWXO, S_IROTH, S_IWOTH, and S_IXOTH; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td>int chown(const char *pathname, uid_t owner, gid_t group);</td>
<td>File I/O</td>
<td>&lt;sys/types.h&gt; &lt;sys/stat.h&gt;</td>
<td>Changes the owner and group of a file; function-shipped to CIOD.</td>
</tr>
<tr>
<td>int close(int filedes);</td>
<td>File I/O</td>
<td>&lt;unistd.h&gt;</td>
<td>Closes a file descriptor; function-shipped to CIOD; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td>int dup(int filedes);</td>
<td>File I/O</td>
<td>&lt;unistd.h&gt;</td>
<td>Duplicates an open descriptor; function-shipped to CIOD; returns new file descriptor if OK or -1 on error.</td>
</tr>
<tr>
<td>int dup2(int filedes, int filedes2);</td>
<td>File I/O</td>
<td>&lt;unistd.h&gt;</td>
<td>Duplicates an open descriptor; function-shipped to CIOD; returns new file descriptor if OK or -1 on error.</td>
</tr>
<tr>
<td>int fchmod(int filedes, mode_t mode);</td>
<td>File I/O</td>
<td>&lt;sys/types.h&gt; &lt;sys/stat.h&gt;</td>
<td>Changes the mode of a file; function-shipped to CIOD; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td>int fchown(int filedes, uid_t owner, gid_t group);</td>
<td>File I/O</td>
<td>&lt;sys/types.h&gt; &lt;unistd.h&gt;</td>
<td>Changes the owner and group of a file; function-shipped to CIOD; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td>int fcntl(int filedes, int cmd, int arg);</td>
<td>File I/O</td>
<td>&lt;sys/types.h&gt; &lt;unistd.h&gt; &lt;fcntl.h&gt;</td>
<td>Performs the following operations on an open file; function-shipped to CIOD, mode: F_GETFL, F_DUPFD, F_GETLK, F_SETLK, F_SETLK64, F_SETLKW, F_SETLKW64. What is returned depends on the command if OK or NULL on error.</td>
</tr>
<tr>
<td>int fstat(int filedes, struct stat *buf);</td>
<td>File I/O</td>
<td>&lt;sys/types.h&gt; &lt;sys/stat.h&gt;</td>
<td>Gets the file status; function-shipped to CIOD; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td>int stat64(const char *path, struct stat64 *buf);</td>
<td>File I/O</td>
<td>&lt;sys/types.h&gt; &lt;sys/stat.h&gt;</td>
<td>Gets the file status.</td>
</tr>
<tr>
<td>int statfs(const char *path, struct statfs *buf);</td>
<td>File I/O</td>
<td>&lt;sys/vfs.h&gt;</td>
<td>Gets file system statistics.</td>
</tr>
<tr>
<td>Function prototype</td>
<td>Category</td>
<td>Header required</td>
<td>Description and type</td>
</tr>
<tr>
<td>-----------------------------------------------------------------------------------</td>
<td>----------</td>
<td>----------------</td>
<td>-------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>long fstatfs64 (unsigned int fd, size_t sz, struct statfs64 *buf);</td>
<td>File I/O</td>
<td>&lt;sys/vfs.h&gt;</td>
<td>Gets file system statistics.</td>
</tr>
<tr>
<td>int fsync(int filedes);</td>
<td>File I/O</td>
<td>&lt;unistd.h&gt;</td>
<td>Synchronizes changes to a file; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td>int ftruncate(int filedes, off_t length);</td>
<td>File I/O</td>
<td>&lt;sys/types.h&gt;</td>
<td>Truncates a file to a specified length; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td>int ftruncate64(int filedes, off64_t length);</td>
<td>File I/O</td>
<td>&lt;unistd.h&gt;</td>
<td>Truncates a file to a specified length for files larger than 2 GB; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td>int lchown(const char *pathname, uid_t owner, gid_t group);</td>
<td>File I/O</td>
<td>&lt;unistd.h&gt;</td>
<td>Changes the owner and group of a symbolic link; function-shipped to CIOD; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td>int link(const char *existingpath, const char *newpath);</td>
<td>File I/O</td>
<td>&lt;unistd.h&gt;</td>
<td>Links to a file; function-shipped to CIOD; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td>off_t lseek(int filedes, off_t offset, int whence);</td>
<td>File I/O</td>
<td>&lt;unistd.h&gt;</td>
<td>Moves the read/write file offset; function-shipped to CIOD; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td>int _llseek(unsigned int fd, unsigned long offset_high, unsigned long offset_low, loff_t *result, unsigned int whence);</td>
<td>File I/O</td>
<td>&lt;unistd.h&gt;</td>
<td>Moves the read/write file offset.</td>
</tr>
<tr>
<td>int lstat(const char *pathname, struct stat *buf);</td>
<td>File I/O</td>
<td>&lt;sys/types.h&gt;</td>
<td>Gets the symbolic link status; function-shipped to CIOD; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td>int lstat64(const char *pathname, struct stat64 *buf);</td>
<td>File I/O</td>
<td>&lt;sys/types.h&gt;</td>
<td>Gets the symbolic link status; determines the size of a file larger than 2 GB.</td>
</tr>
<tr>
<td>int open(const char *pathname, int oflag, mode_t mode);</td>
<td>File I/O</td>
<td>&lt;sys/types.h&gt;</td>
<td>Opens a file; function-shipped to CIOD; oflag: O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_CREAT, O_EXCL, O_TRUNC, O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_CREAT, O_EXCL, O_TRUNC, O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_CREAT, O_EXCL, O_TRUNC, mode: S_IRWXU, S_IRUSR, S_IWUSR, S_IRWXG, S_IRWXO, S_IROTH, S_IWOTH, and S_IROTH; returns file descriptor if OK or -1 on error.</td>
</tr>
<tr>
<td>ssize_t read(int filedes, void *buff, size_t nbytes);</td>
<td>File I/O</td>
<td>&lt;unistd.h&gt;</td>
<td>Reads from a file; function-shipped to CIOD; returns number of bytes; read if OK, 0 if end of file, or -1 on error.</td>
</tr>
<tr>
<td>int readlink(const char *pathname, char *buf, int bufsize);</td>
<td>File I/O</td>
<td>&lt;unistd.h&gt;</td>
<td>Reads the contents of a symbolic link; function-shipped to CIOD; returns number of bytes read if OK or -1 on error.</td>
</tr>
<tr>
<td>ssize_t readv(int filedes, const struct iovec iov[], int iovcnt)</td>
<td>File I/O</td>
<td>&lt;sys/uio.h&gt;</td>
<td>Reads a vector, function-shipped to CIOD; returns number of bytes read if OK or -1 on error.</td>
</tr>
<tr>
<td>Function prototype</td>
<td>Category</td>
<td>Header required</td>
<td>Description and type</td>
</tr>
<tr>
<td>-----------------------------------------------------------------------------------</td>
<td>------------</td>
<td>--------------------------------------</td>
<td>-----------------------------------------------------------</td>
</tr>
<tr>
<td>int rename(const char *oldname, const char *newname);</td>
<td>File I/O</td>
<td>&lt;stdio.h&gt;</td>
<td>Renames a file; function-shipped to CIOD; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td>int stat(const char *pathname, struct stat *buf);</td>
<td>File I/O</td>
<td>&lt;sys/types.h&gt; &lt;sys/stat.h&gt;</td>
<td>Gets the file status; function-shipped to CIOD; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td>int stat64(const char *pathname, struct stat64 *buf);</td>
<td>File I/O</td>
<td>&lt;sys/types.h&gt; &lt;sys/stat.h&gt;</td>
<td>Gets the file status.</td>
</tr>
<tr>
<td>int statfs (char *Path, struct statfs *StatusBuffer);</td>
<td>File I/O</td>
<td>&lt;sys/statfs.h&gt;</td>
<td>Gets file system statistics.</td>
</tr>
<tr>
<td>long statfs64 (const char *path, size_t sz, struct statfs64 *buf);</td>
<td>File I/O</td>
<td>&lt;sys/statfs.h&gt;</td>
<td>Gets file system statistics.</td>
</tr>
<tr>
<td>int symlink(const char *actualpath, const char *sympath);</td>
<td>File I/O</td>
<td>&lt;unistd.h&gt;</td>
<td>Makes a symbolic link to a file; function-shipped to CIOD; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td>int truncate(const char *pathname, off_t length);</td>
<td>File I/O</td>
<td>&lt;sys/types.h&gt; &lt;unistd.h&gt;</td>
<td>Truncates a file to a specified length; function-shipped to CIOD; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td>truncate64(const char *path, off_t length);</td>
<td>File I/O</td>
<td>&lt;unistd.h&gt; &lt;sys/types.h&gt;</td>
<td>Truncates a file to a specified length.</td>
</tr>
<tr>
<td>mode_t umask(mode_t cmask);</td>
<td>File I/O</td>
<td>&lt;sys/types.h&gt; &lt;sys/stat.h&gt;</td>
<td>Sets and gets the file mode creation mask; function-shipped to CIOD; returns the previous file mode creation mask.</td>
</tr>
<tr>
<td>int unlink(const char *pathname);</td>
<td>File I/O</td>
<td>&lt;unistd.h&gt;</td>
<td>Removes a directory entry; function-shipped to CIOD; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td>int utime(const char *pathname, const struct utimbuf *times);</td>
<td>File I/O</td>
<td>&lt;sys/types.h&gt; &lt;utime.h&gt;</td>
<td>Sets file access and modification times; function-shipped to CIOD; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td>ssize_t write(int filedes, const void *buff, size_t nbytes);</td>
<td>File I/O</td>
<td>&lt;unistd.h&gt;</td>
<td>Writes to a file; function-shipped to CIOD; returns the number of bytes written if OK or -1 on error.</td>
</tr>
<tr>
<td>ssize_t writev(int filedes, const struct iovec iov[], int iovcntl);</td>
<td>File I/O</td>
<td>&lt;sys/types.h&gt; &lt;sys/uio.h&gt;</td>
<td>Writes a vector; function-shipped to CIOD; returns the number of bytes written if OK or -1 on error.</td>
</tr>
<tr>
<td>int chdir(const char *pathname);</td>
<td>Directory</td>
<td>&lt;unistd.h&gt;</td>
<td>Changes the working directory; function-shipped to CIOD; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td>char *getcwd(char *buf, size_t size);</td>
<td>Directory</td>
<td>&lt;unistd.h&gt;</td>
<td>Gets the path name of the current working directory; function-shipped to CIOD; returns buf if OK or NULL on error.</td>
</tr>
<tr>
<td>int getdents(int filedes, char **buf, unsigned nbyte);</td>
<td>Directory</td>
<td>&lt;sys/types.h&gt;</td>
<td>Gets the directory entries in a file system; function-shipped to CIOD; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td>getdents64(unsigned int fd, struct dirent *dirp, unsigned int count);</td>
<td>Directory</td>
<td>&lt;sys/dirent.h&gt;</td>
<td>Gets the directory entries in a file system.</td>
</tr>
<tr>
<td>Function prototype</td>
<td>Category</td>
<td>Header required</td>
<td>Description and type</td>
</tr>
<tr>
<td>--------------------</td>
<td>----------------</td>
<td>--------------------------</td>
<td>--------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>int mkdir(const char *pathname, mode_t mode);</td>
<td>Directory</td>
<td><code>&lt;sys/types.h&gt;</code>&lt;sys/stat.h&gt;`</td>
<td>Makes a directory; function-shipped to CIOD; mode S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH, S_IWOTH, and S_IXOTH; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td>int rmdir(const char *pathname);</td>
<td>Directory</td>
<td><code>&lt;unistd.h&gt;</code></td>
<td>Removes a directory; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td>int getitimer(int which, struct itimerval *value);</td>
<td>Time</td>
<td><code>&lt;sys/time.h&gt;</code></td>
<td>Gets the value of the interval timer; local system call; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td>int gettimeofday(struct timeval *tp, void *restrict tzp);</td>
<td>Time</td>
<td><code>&lt;sys/time.h&gt;</code></td>
<td>Gets the date and time; local system call; returns 0 if OK or NULL on error.</td>
</tr>
<tr>
<td>int setitimer(int which, const struct itimerval *value, struct itimerval *ovalue);</td>
<td>Time</td>
<td><code>&lt;sys/time.h&gt;</code></td>
<td>Sets the value of an interval timer; only the following operations are supported: ITIMER_PROF, ITIMER_REAL. <strong>Note:</strong> An application can only set one active timer at a time.</td>
</tr>
<tr>
<td>time_t time(time_t *calptr);</td>
<td>Time</td>
<td><code>&lt;time.h&gt;</code></td>
<td>Gets the time; local system call; returns the value of time if OK or -1 on error.</td>
</tr>
<tr>
<td>gid_t getgid(void);</td>
<td>Process information</td>
<td><code>&lt;unistd.h&gt;</code></td>
<td>Gets the real group ID.</td>
</tr>
<tr>
<td>pid_t getpid(void);</td>
<td>Process information</td>
<td><code>&lt;unistd.h&gt;</code></td>
<td>Gets the process ID. The value is the MPI rank of the node, meaning that 0 is a valid value.</td>
</tr>
<tr>
<td>int getrusage(int who, struct rusage *r_usage);</td>
<td>Process information</td>
<td><code>&lt;sys/resource.h&gt;</code></td>
<td>Gets information about resource utilization. All time reported is attributed to the user application, so the reported system time is always zero.</td>
</tr>
<tr>
<td>uid_t getuid(void);</td>
<td>Process information</td>
<td><code>&lt;unistd.h&gt;</code></td>
<td>Gets the real user ID.</td>
</tr>
<tr>
<td>int setrlimit(int resource, const struct rlimit *rlp);</td>
<td>Process information</td>
<td><code>&lt;sys/resource.h&gt;</code></td>
<td>Sets resource limits. Only RLIMIT_CORE can be set.</td>
</tr>
<tr>
<td>clock_t times(struct tms *buf);</td>
<td>Process information</td>
<td><code>&lt;sys/times.h&gt;</code></td>
<td>Gets the process times. All time reported is attributed to the user application, so the reported system time is always zero.</td>
</tr>
<tr>
<td>int brk(void *end_data_segment);</td>
<td>Miscellaneous</td>
<td><code>&lt;unistd.h&gt;</code></td>
<td>Changes the data segment size.</td>
</tr>
<tr>
<td>void exit(int status);</td>
<td>Miscellaneous</td>
<td><code>&lt;stdlib.h&gt;</code></td>
<td>Terminates a process.</td>
</tr>
<tr>
<td>int uname(struct utsname *buf);</td>
<td>Miscellaneous</td>
<td><code>&lt;sys/utsname.h&gt;</code></td>
<td>Gets the name of the current system and other information, for example, version and release.</td>
</tr>
</tbody>
</table>
6.2.3 Other system calls

Although many system calls are unsupported, you must be aware of the following unsupported calls:

- The Blue Gene/P system does not support the use of the `system()` function. Therefore, for example, you cannot use something such as the `system('chmod -w file')` call. Although, `system()` is not a system call, it uses `fork()` and `exec()` via `glibc`. Both `fork()` and `exec()` are currently not implemented.

- The Blue Gene/P system does not provide the same support for `gethostname()` and `getlogin()` as Linux provides.

- Calls to `usleep()` are not supported.

See 6.6, “Unsupported system calls” on page 62, for a complete list of unsupported system calls.

6.3 System programming interfaces

Low-level access to Blue Gene/P-specific interfaces, such as direct memory access (DMA), is provided by the system programming interfaces (SPIs). These interfaces provide a consistent interface for Linux and compute node kernel-based applications to access the hardware.

The following Blue Gene/P-specific interfaces are included in the SPI:

- Collective network
- Torus network
- Direct memory access
- Global interrupts
- Performance counters
- Lockbox

The following items are not included in the SPI:

- L2
- Snoop
- L3
- DDR hardware initialization
- serdes
- Environmental monitor

This hardware is set up by either the bootloader or Common Node Services. The L1 interfaces, such as TLB miss handlers, are typically extremely operating system specific, and therefore an SPI is not defined. TOMAL and XEMAC are present in the Linux 10 Gb Ethernet device driver (and therefore open source), but there are no plans for an explicit SPI.

6.4 Socket support

The compute node kernel provides socket support via the standard Linux `socketcall()` system call. The `socketcall()` is a kernel entry point for the socket system calls. It determines which socket function to call and points to a block that contains the actual parameters, which are passed through to the appropriate call. The compute node kernel function-ships the `socketcall()` parameters to the CIOD, which then performs the requested operation. The CIOD is a user-level process that controls and services applications in the compute node and interacts with the Midplane Management Control System (MMCS).
This socket support allows the creation of both outbound and inbound socket connections using standard Linux APIs. For example, an outbound socket can be created by calling `socket()`, followed by `connect()`. An inbound socket can be created by calling `socket()` followed by `bind()`, `listen()`, and `accept()`.

Communication through the socket is provided via the glibc `send()`, `recv()`, and `select()` function calls. These function calls invoke the `socketcall()` system call with different parameters. Table 6-2 summarizes the list of Linux 2.4 socket system calls.

<table>
<thead>
<tr>
<th>Function prototype</th>
<th>Category</th>
<th>Header required</th>
<th>Description and type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);</code></td>
<td>Sockets</td>
<td>&lt;sys/types.h&gt;</td>
<td>Extracts the connection request on the queue of pending connections; creates a new connected socket; returns a file descriptor if OK or -1 on error.</td>
</tr>
<tr>
<td><code>int bind(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen);</code></td>
<td>Sockets</td>
<td>&lt;sys/types.h&gt;</td>
<td>Assigns a local address; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td><code>int connect(int socket, const struct sockaddr *address, socklen_t address_len);</code></td>
<td>Sockets</td>
<td>&lt;sys/types.h&gt;</td>
<td>Connects a socket; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td><code>int getpeername(int socket, struct sockaddr *restrict address, socklen_t *restrict address_len);</code></td>
<td>Sockets</td>
<td>&lt;sys/socket.h&gt;</td>
<td>Gets the name of the peer socket; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td><code>int getsockname(int socket, struct sockaddr *restrict address, socklen_t *restrict address_len);</code></td>
<td>Sockets</td>
<td>&lt;sys/socket.h&gt;</td>
<td>Retrieves the locally bound socket name; stores the address in sockaddr; and stores its length in the address_len argument; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td><code>int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen);</code></td>
<td>Sockets</td>
<td>&lt;sys/types.h&gt;</td>
<td>Manipulates options that are associated with a socket; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td><code>int listen(int sockfd, int backlog);</code></td>
<td>Sockets</td>
<td>&lt;sys/socket.h&gt;</td>
<td>Accepts connections; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td><code>ssize_t recv(int s, void *buf, size_t len, int flags);</code></td>
<td>Sockets</td>
<td>&lt;sys/types.h&gt;</td>
<td>Receives a message only from a connected socket; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td><code>ssize_t recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen);</code></td>
<td>Sockets</td>
<td>&lt;sys/types.h&gt;</td>
<td>Receives a message from a socket regardless of whether it is connected; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td><code>ssize_t recvmsg(int s, struct msghdr *msg, int flags);</code></td>
<td>Sockets</td>
<td>&lt;sys/types.h&gt;</td>
<td>Receives a message from a socket regardless of whether it is connected; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td><code>ssize_t send(int socket, const void *buffer, size_t length, int flags);</code></td>
<td>Sockets</td>
<td>&lt;sys/types.h&gt;</td>
<td>Sends a message only to a connected socket; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td><code>ssize_t sendto(int socket, const void *message, size_t length, int flags, const struct sockaddr *dest_addr, socklen_t dest_len);</code></td>
<td>Sockets</td>
<td>&lt;sys/types.h&gt;</td>
<td>Sends a message on a socket; returns 0 if OK or -1 on error.</td>
</tr>
</tbody>
</table>
6.5 Signal support

The compute node kernel provides ANSI-C signal support via the standard Linux system calls `signal()` and `kill()`. Additionally, signals can be delivered externally by using `mpirun` or for HTC using `submit`. Table 6-3 summarizes the supported signals.

<table>
<thead>
<tr>
<th>Function prototype</th>
<th>Category</th>
<th>Header required</th>
<th>Description and type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssize_t sendmsg(int s, const struct msghdr *msg, int flags);</td>
<td>Sockets</td>
<td>&lt;sys/types.h&gt; &lt;sys/socket.h&gt;</td>
<td>Sends a message on a socket; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td>int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen);</td>
<td>Sockets</td>
<td>&lt;sys/types.h&gt; &lt;sys/socket.h&gt;</td>
<td>Manipulates options that are associated with a socket; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td>int shutdown(int s, int how);</td>
<td>Sockets</td>
<td>&lt;sys/socket.h&gt;</td>
<td>Causes all or part of a connection on the socket to shut down; returns 0 if OK or -1 on error.</td>
</tr>
<tr>
<td>int socket(int domain, int type, int protocol);</td>
<td>Sockets</td>
<td>&lt;sys/types.h&gt; &lt;sys/socket.h&gt;</td>
<td>Opens a socket; returns a file descriptor if OK or -1 on error.</td>
</tr>
<tr>
<td>int socketpair(int d, int type, int protocol, int sv[2]);</td>
<td>Sockets</td>
<td>&lt;sys/types.h&gt; &lt;sys/socket.h&gt;</td>
<td>Creates an unnamed pair of connected sockets; returns 0 if OK or -1 on error.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function prototype</th>
<th>Category</th>
<th>Header required</th>
<th>Description and type</th>
</tr>
</thead>
<tbody>
<tr>
<td>int kill(pid_t pid, int sig);</td>
<td>Signals</td>
<td>&lt;sys/types.h&gt; &lt;signal.h&gt;</td>
<td>Sends a signal. A signal can be sent only to the same process.</td>
</tr>
<tr>
<td>int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);</td>
<td>Signals</td>
<td>&lt;signal.h&gt;</td>
<td>Manages signals. The only flags supported are SA_RESETHAND and SA_NODEFER.</td>
</tr>
<tr>
<td>typedef void (*sighandler_t)(int); sighandler_t signal(int signum, sighandler_t handler);</td>
<td>Signals</td>
<td>&lt;signal.h&gt;</td>
<td>Manages signals.</td>
</tr>
<tr>
<td>typedef void (*sighandler_t)(int); sighandler_t signal(int signum, sighandler_t handler);</td>
<td>Signals</td>
<td>&lt;signal.h&gt;</td>
<td>Returns from a signal handler.</td>
</tr>
</tbody>
</table>

Table 6-3   Supported signals
6.6 Unsupported system calls

The role of the kernel on the compute node is to create an environment for the execution of a user process that is “Linux like.” It is not a full Linux kernel implementation, but rather implements a subset of the POSIX functionality. The following list indicates the system calls that are not supported:

- acct
- adjtimex
- afs_syscall
- bdflush
- break
- capget
- capset
- chroot
- clock_getres
- clock_gettime
- clock_nanosleep
- clock_gettime
- create_module
- delete_module
- epoll_create
- epoll_ctl
- epoll_wait
- execve
- fadvise64
- fadvise64_64
- fchdir
- fdatasync
- fgetxattr
- flistxattr
- flock
- fork
- fremovexattr
- fsetxattr
- ftime
- get_kernel_syms
- iopl
- ipc
- kexec_load
- lgetxattr
- listxattr
- listxattr
- lock
- lookup_dcookie
- lremovexattr
- lsetxattr
- mincore
- mknod
- mpinxmq_getsetattr
- mq_notify
- mq_open
- mq_timedreceive
- mq_timedsend
- mq_unlink
- multiplexer
- nfsservctl
- nice
- oldfstat
- oldfstat
- oldoldname
- oldname
- oldstat
- oldstat
- pciconfig_iobase
- pciconfig_read
- removexattr
- rts
- rts_device_map
- rts_dma
- sched_get_priority_max
- sched_get_priority_min
- sched_getaffinity
- sched_getparam
- sched_getscheduler
- sched_rr_get_interval
- sched_setaffinity
- sched_setscheduler
- sched_setscheduler
- sched_setparam
- sched_yield
- select
- sendfile
- sendfile64
- setdomainname
- setgroups
- sethostname
- setpriority
- settimeofday
- settimeofday
- settimeofday
- stime
- stty
- swapcontext
- swapoff
- swapon
- sync
- sys_debug_setcontext
You can find additional information about these system calls on the syscalls(2) - Linux man page on the Web at:

http://linux.die.net/man/2/syscalls
Applications environment

In this part, we provide an overview of some of the software that forms part of the applications environment. Throughout this book, we consider the applications environment as the collection of programs that are required to develop applications.

This part includes the following chapters:

- Chapter 7, “Parallel paradigms” on page 67
- Chapter 8, “Developing applications with IBM XL compilers” on page 95
- Chapter 9, “Running and debugging applications” on page 137
- Chapter 10, “Checkpoint and restart support for applications” on page 159
- Chapter 11, “mpirun” on page 167
- Chapter 12, “High-Throughput Computing (HTC) paradigm” on page 187
Parallel paradigms

In this chapter, we discuss the parallel paradigms that are offered on the Blue Gene/P system. One such paradigm is the Message Passing Interface (MPI),\textsuperscript{22} for a distributed-memory architecture, and OpenMP,\textsuperscript{23} for shared-memory architectures. We refer to this paradigm as \textit{high-performance computing} (HPC). Blue Gene/P also offers a paradigm where applications do not require communication between tasks and each node is running a different instance of the application. We refer to this paradigm as \textit{High-Throughput Computing} (HTC). This topic is discussed in Chapter 14.

In this chapter, we address the following topics:

- Programming model
- Blue Gene/P MPI implementation
- MPI communications
- MPI functions
- Compiling MPI programs on Blue Gene/P
- MPI communications performance
- OpenMP
7.1 Programming model

The Blue Gene/P system has a distributed memory system and uses explicit message passing to communicate between tasks that are running on different nodes. It also has shared memory on each node; OpenMP and thread parallelism are supported as well.

MPI is the supported message-passing standard. It is the industry standard for message passing. For further information about MPI, refer to the Message Passing Interface Forum site on the Web at the following address:

http://www.mpi-forum.org/

If your code uses other message-passing libraries, you must either change the message-passing calls to MPI or use an intermediate layer that maps your library’s calls to MPI.

7.2 Blue Gene/P MPI implementation

The current MPI implementation on the Blue Gene/P system supports the MPI-1.2 and MPI-2 standards of MPI Version 1.2. They are both extensions to the MPI-1.1 standard. The only exception is process management. For more information about process management, see the paper “Dynamic Process Management in an MPI Setting,” by William Gropp and Ewing Lusk, on the Web at:


When starting applications on the Blue Gene/P system, you must consider that the microkernel running on the compute nodes does not provide any mechanism for a command interpreter or shell. Only the executables can be started. Shell scripts are not supported. Therefore, if your application consists of a number of shell scripts that control its workflow, the workflow must be adapted. If you start your application with the `mpirun` command, you cannot start the main shell script with this command. Instead, you must run the scripts on the front-end node and only call `mpirun` at the innermost shell script level where the main application binary is called.

The MPI implementation on the Blue Gene/P system is derived from the MPICH2 implementation of the Mathematics and Computer Science Division (MCS) at Argonne National Laboratory. For additional information, refer to the MPICH2 Web site at:

http://www-unix.mcs.anl.gov/mpi/mpich/

To support the Blue Gene/P hardware, the following additions and modifications have been made to the MPICH2 software architecture:

- A Blue Gene/P driver has been added underneath the MPICH2 abstract device interface (ADI).
- Three types of glue code are provided for some MPI collectives. One type of glue code is provided for each of the three networks that can be used for MPI communication on the Blue Gene/P system:
  - torus for the torus network
  - Tree for the collective network
  - Global Interrupt (GI) for the barrier network
- Optimized versions of the Cartesian functions exist (MPI_Dims_create, MPI_Cart_create, MPI_Cart_map).
MPIX functions create hardware-specific MPI extensions.

Other parallel paradigms have been included as shown in Figure 7-1.

From the application programmer’s view, the most important aspect of these changes is the fact that the collective operations can use different networks under different circumstances. In 7.2.1, “High-performance network for efficient parallel execution” on page 69, we briefly summarize the different networks on the Blue Gene/P system and network routing.

In 7.2.2, “Forcing MPI to allocate too much memory” on page 71, through 7.2.7, “Interlocking collectives with point-to-point calls” on page 75, we discuss several sample MPI codes to explain some of the implementation-dependent behaviors of the MPI library.

### 7.2.1 High-performance network for efficient parallel execution

The Blue Gene/P system does not have a single type of network that is capable of transporting all protocols that are needed in such an environment. Therefore, the Blue Gene/P system has implemented separate networks for different types of communications.

#### Collective network

The three-dimensional (3D) torus is an efficient network for communicating with neighbors. However, such calls as all-to-one, one-to-all, and all-to-all more efficiently use the collective network. The collective network connects all the compute nodes in the shape of a tree. Any node can be the tree root. The MPI implementation uses the collective network, which is more efficient than the torus network for collective communication.

#### Point-to-point network

All MPI point-to-point communications are carried out via the torus network. The route from a sender to a receiver on a torus network has two possible paths:

- Deterministic routing
  
  Packets from a sender to a receiver go along the same path. One advantage of this path is that the packet order is always maintained without additional logic. However, this technique also creates network hot spots if several point-to-point communications occur at the same time and their deterministic routes cross on some node.
Adaptive routing

Different packets from the same sender to the same receiver can travel along different paths. The exact route is determined at run time depending on the current load. This technique generates a more balanced network load but introduces a latency penalty.

Selecting deterministic or adaptive routing depends on the protocol used for communication. The Blue Gene/P MPI implementation supports three different protocols:

- **MPI short protocol**
  The MPI short protocol is used for short messages (less than 224 bytes), which consist of a single packet. These messages are always deterministically routed. The latency for eager messages is around 3.3 µs.

- **MPI eager protocol**
  The MPI eager protocol is used for medium-sized messages. It sends a message to the receiver without negotiating with the receiving side that the other end is ready to receive the message. This protocol also uses deterministic routes for its packets.

- **MPI rendezvous protocol**
  Large (greater than 1200 bytes) messages are sent using the MPI rendezvous protocol. In this case, an initial connection between the two partners is established. Only after that connection is established, does the receiver use direct memory access (DMA) to obtain the data from the sender. This protocol uses adaptive routing and is optimized for maximum bandwidth. Naturally, the initial rendezvous handshake increases the latency.

The Blue Gene/P MPI library supports a DCMF_EAGER variable (which can be set via `mpirun`) to set the message size (in bytes) above which the rendezvous protocol should be used. Consider the following guidelines:

- Decrease the rendezvous threshold if any of the following situations are true:
  - Many short messages are overloading the network.
  - Eager messages are creating artificial hot spots.
  - The program is not latency-sensitive.

- Increase the rendezvous threshold if any of the following situations are true:
  - Most communication is a nearest neighbor or at least close in Manhattan distance, where this distance is the shortest number of hops between a pair of nodes.
  - You mainly use relatively long messages.
  - You need better latency on medium-sized messages.

The following guidelines are also necessary for proper MPI usage:

- **Overlap communication and computation using** `MPI_Irecv` and `MPI_Isend`, which allow DMA to work in the background.

  **DMA and the tree and GI networks**: The tree and GI networks do not use DMA. In this case, operations cannot be completed in the background.

- **Avoid load imbalance.**
  This is important for all parallel systems. However, when scaling to the high numbers of tasks that are possible on the Blue Gene/P system, it is important to pay close attention to load balancing.

- **Avoid buffered and synchronous sends; post receives in advance.**
  The MPI standard defines several specialized communication modes in addition to the standard send function, `MPI_Send()`. Avoid the buffered send function, `MPI_Bsend()`. If you
use this function, forcing the MPI library to perform additional memory copies slows down the application, and you might run short of memory so additional buffering might not be possible at all. Using the synchronous send function \texttt{MPI\_Ssend()} is discouraged because it is a non-local operation that incurs an increased latency compared to the standard send.

- Avoid vector data and non-contiguous data types.

While the MPI-derived data types can elegantly describe the layout of complex data structures, using these data types is generally detrimental to performance. Many MPI implementations pack (that is, memory-copy) such data objects before sending them. This packing of data objects is contrary to the original purpose of MPI-derived data types, namely to avoid such memory copies. In addition, the Blue Gene/P MPI implementation uses the chips' special quad-word load and quad-word store instructions, which require appropriately aligned and continuous data.

### 7.2.2 Forcing MPI to allocate too much memory

Forcing MPI to allocate too much memory is relatively easy to do with basic code. For example, the snippets of legal MPI code shown in Example 7-1 and Example 7-2 run the risk of forcing the MPI support to allocate too much memory, resulting in failure, because it forces excessive buffering of messages.

**Example 7-1** CPU1 MPI code that can cause excessive memory allocation

```c
MPI\_ISend(cpu2, tag1);
MPI\_ISend(cpu2, tag2);
...
MPI\_ISend(cpu2, tagn);
```

**Example 7-2** CPU2 MPI code that can cause excessive memory allocation

```c
MPI\_Recv(cpu1, tagn);
MPI\_Recv(cpu1, tagn-1);
...
MPI\_Recv(cpu1, tag1);
```

Example 7-2 illustrates a section of code that was particularly important on the Blue Gene/L system. However, on the Blue Gene/P system, we recommend this as a good programming practice.

Keep in mind the following points:

- The Blue Gene/P MPI rendezvous protocol does not allocate an unexpected buffer for the receive. This proper buffer allocation prevents most problems by drastically reducing the memory footprint of unexpected messages.

- The message queue is searched linearly to meet MPI matching requirements. If several messages are on the queue, the search can take longer.

You can accomplish the same goal and avoid memory allocation issues by recoding as shown in Example 7-3 and Example 7-4 on page 72.

**Example 7-3** CPU1 MPI code that can avoid excessive memory allocation

```c
MPI\_ISend(cpu2, tag1);
MPI\_ISend(cpu2, tag2);
...
MPI\_ISend(cpu2, tagn);
```
Example 7-4  CPU2 MPI code that can avoid excessive memory allocation

MPI_Recv(cpu1, tag1);
MPI_Recv(cpu1, tag2);
...
MPI_Recv(cpu1, tagn);

7.2.3 Not waiting for MPI_Test

According to the MPI standard, an application must either wait or continue testing until
MPI_Test returns true. Not doing so causes small memory leaks, which can accumulate over
time and cause a memory overrun. Example 7-5 shows the code and the problem.

Example 7-5  Potential memory overrun caused by not waiting for MPI_Test

req = MPI_Isend( ... );
MPI_Test (req);
... do something else; forget about req ...

Remember to use MPI_Wait or loop until MPI_Test returns true.

7.2.4 Flooding of messages

The code shown in Example 7-6, while legal, floods the network with messages. It can cause
CPU 0 to run out of memory. Even though it can work, it is not scalable.

Example 7-6  Flood of messages resulting in a possible memory overrun

CPU 1 to n-1 code:
MPI_Send(cpu0);

CPU 0 code:
for (i=1; i<n; i++)
    MPI_Recv(cpu[i]);

7.2.5 Deadlock the system

The code shown in Example 7-7 is illegal according to the MPI standard. Each side does a
blocking send to its communication partner before posting a receive for the message coming
from the other partner.

Example 7-7  Deadlock code

TASK1 code:
MPI_Send(task2, tag1);
MPI_Recv(task2, tag2);
TASK2 code:
MPI_Send(task1, tag2);
MPI_Recv(task1, tag1);

In general, this code has a high probability of deadlocking the system. Obviously, you should
not program this way. Make sure that your code conforms to the MPI specification. You can
achieve this by either changing the order of sends and receives or by using non-blocking
communication calls.
While you should not rely on the run-time system to correctly handle non-conforming MPI code, it is easier to debug such situations when you receive a run-time error message than to try and detect a deadlock and trace it back to its root cause.

### 7.2.6 Violating MPI buffer ownership rules

A number of problems can arise when the send/receive buffers that participate in asynchronous message-passing calls are accessed before it is legal to do so. All of the following examples are illegal, and therefore, you must avoid them.

The most obvious case is when you write to a send buffer before the MPI_Wait for that request has completed as shown in Example 7-8.

**Example 7-8 Write to a send buffer before MPI_Wait has completed**

```c
req = MPI_Isend(buffer,&req);
buffer[0] = something;
MPI_Wait(req);
```

The code in Example 7-8 results in a race condition on any message-passing machine. Depending on the run-time factors that are outside the application’s control, sometimes the old buffer[0] is sent and sometimes the new value is sent.

A more subtle case is a read from the send buffer before the MPI_Wait for that request completes as shown in Example 7-9.

**Example 7-9 Read from the send buffer before the MPI_Wait completes**

```c
req = MPI_Isend(buffer,&req);
z = buffer[0];
MPI_Wait(req);
```

Although not as obvious as the write case, the code in Example 7-9 is also prohibited by the MPI standard. The MPI run-time system has full control over the buffer until the MPI_Wait for the request completes. The application is not allowed to read it. In the current Blue Gene/P implementation, such code works as expected, but there is no guarantee that future versions of the MPI library will behave the same way.

In the last example in this thread, a receive buffer is read before the MPI_Wait for the asynchronous receive request has completed (see Example 7-10).

**Example 7-10 Receive buffer before MPI_Wait has completed**

```c
req = MPI_Irecv(buffer);
z = buffer[0];
MPI_Wait (req);
```

The code shown in Example 7-10 is also illegal. The contents of the receive buffer are not guaranteed until after MPI_Wait is called.
Buffer alignment sensitivity

It is important to note that the MPI implementation on the Blue Gene/P system is sensitive to the alignment of the buffers that are being sent or received. Aligning buffers on 32-byte boundaries can improve performance. If the buffers are at least 16-bytes aligned, the messaging software can use internal math routines that are optimized for the double hammer architecture. Additionally, the L1 cache and DMA are optimized on 32-byte boundaries.

For buffers that are dynamically allocated (via `malloc()`), the following techniques can be used:

- Instead of using `malloc()`, use the following statement and specify 32 for the alignment parameter:

  ```c
  int posix_memalign(void **memptr, size_t alignment, size_t size)
  ```

  This statement returns a 32-byte aligned pointer to the allocated memory. You can use `free()` to free the memory.

- Use `malloc()`, but request 32 bytes of more storage than required. Then round the returned address up to a 32-byte boundary as shown in Example 7-11.

  ```c
  Example 7-11 Request 32 bytes more storage than required
  buffer_ptr_original = malloc(size + 32);
  buffer_ptr = (char*)( (unsigned)buffer_ptr_original + 32 ) & 0xFFFFFFFFFE0 ;
  /* Use buffer_ptr on MPI operations */
  free(buffer_ptr_original);
  ```

For buffers that are declared in static (global) storage, use `__attribute__((aligned(32)))` on the declaration as shown in Example 7-12.

  ```c
  Example 7-12 Buffers that are declared in static (global) storage
  struct DataInfo
  {
    unsigned int iarray[256];
    unsigned int count;
  } data_info __attribute__ ((aligned ( 32)));
  or
  unsigned int data __attribute__ ((aligned ( 32)));
  or
  char data_array[512] __attribute__((aligned(32)));
  ```

For buffers that are declared in automatic (stack) storage, only up to a 16-byte alignment is possible. Therefore, use dynamically allocated aligned static (global) storage instead.
### 7.2.7 Interlocking collectives with point-to-point calls

Consider the code shown in Example 7-13, in which task 1 issues a barrier synchronization before the preceding asynchronous send is known to have completed.

**Example 7-13  Barrier synchronization issued before the preceding asynchronous send completes**

**TASK1 code:**
```c
req = MPI_Isend(task2, &req);
MPI_Barrier();
MPI_Wait(req);
```

**TASK2 code:**
```c
MPI_Recv(task1);
MPI_Barrier();
```

The receiver does not join the barrier before its (blocking) receive has completed. Therefore, this code can potentially deadlock if task 1 enters the barrier before the asynchronous send completes and if task 1 relies on the MPI_Wait to complete the send operation.

On the Blue Gene/P system, this kind of code works because the asynchronous send is handled by the torus network, where the barrier is handled by the barrier (global interrupt) network. Even though task 1 might have already entered the barrier, it is still possible to make progress on the point-to-point communications on the torus network, and the blocking receive on task 2 eventually completes.

To avoid unexpected behavior, do not interlock collectives with point-to-point communications. For all collectives, except MPI_Barrier, the MPI standard clearly states that programmers should not rely on collective communications to synchronize the tasks, and at the same time, should structure their program in a way that allows for such synchronization to take place without causing a deadlock in the point-to-point communications. In general, we recommend that you do not mix collectives, which is not good programming practice.

### 7.3 MPI communications

In this section, we discuss Blue Gene/P-specific features that are related to MPI communications via the torus network.

#### 7.3.1 Blue Gene/P MPI extensions

Three new APIs make it easier to map nodes to specific hardware or processor set (pset) configurations. Application developers can use these functions, as explained in the following list, by including the mpix.h file:

- ```c
    int MPIX_Cart_comm_create (MPI_Comm *cart_comm);
    ```

  This function creates a four-dimensional (4D) Cartesian communicator that mimics the exact hardware on which it is run. The X, Y, and Z dimensions match those of the partition hardware, while the T dimension has cardinality 1 in symmetrical multiprocessing (SMP) node mode, cardinality 2 in dual node mode, and cardinality 4 in virtual node mode. The communicator wrap-around links match the true mesh or torus nature of the partition. In addition, the coordinates of a node in the communicator match exactly its coordinates in the partition.

  It is important to understand that this is a collective operation and it must be run on all nodes. The function might be unable to complete successfully for several different reasons, mostly likely when it is run on fewer nodes than the entire partition. It is important
to ensure that the return code is MPI_SUCCESS before continuing to use the returned communicator.

- `int MPIX_Pset_same_comm_create (MPI_Comm *pset_comm);`

  This function is a collective operation that creates a set of communicators (each node seeing only one), where all nodes in a given communicator are part of the same pset (all share the same I/O node) - see Figure 7-2.

  The most common use for this function is to coordinate access to the outside world to maximize the number of I/O nodes. For example, node 0 in each of the communicators can be arbitrarily used as the “master node” for the communicator, collecting information from the other nodes for writing to disk.

Figure 7-2  MPIX_Pset_same_comm_create() creating communicators

- `int MPIX_Pset_diff_comm_create (MPI_Comm *pset_comm);`

  This function is a collective operation that creates a set of communicators (each node seeing only one), where no two nodes in a given communicator are part of the same pset (all have different I/O nodes) - see Figure 7-3 on page 77. The most common use for this function is to coordinate access to the outside world to maximize the number of I/O nodes. For example, an application that has an extremely high bandwidth per node requirement can run both MPIX_Pset_same_comm_create() and MPIX_Pset_diff_comm_create().

  Nodes without rank 0 in MPIX_Pset_same_comm_create() can sleep, leaving those with rank 0 independent and parallel access to the functional Ethernet. Those nodes all belong to the same communicator from MPIX_Pset_diff_comm_create(), allowing them to use that communicator instead of MPI_COMM_WORLD for group communication or coordination.
7.4 MPI functions

MPI functions have been extensively documented in the literature. In this section, we provide several useful references that provide a comprehensive description of the MPI functions.

Appendix A in *Parallel Programming in C with MPI and OpenMP*, by Michael J. Quinn, describes all the MPI functions as defined in the MPI-1 standard. This reference also provides additional information and recommendations when to use each function.

In addition, you can find information about the MPI standard on the Message Passing Interface (MPI) standard Web site at:

http://www-unix.mcs.anl.gov/mpi/

A comprehensive list of the MPI functions is available on the MPI Routines page at:

The MPI Routines page includes MPI calls for C and Fortran. For more information, refer to the following books about MPI and MPI-2:


For general information about MPICH2, refer to the MPICH2 Web page at:

http://www-unix.mcs.anl.gov/mpi/mpich/

Because teaching MPI is beyond the scope of this book, refer to the following Web page for tutorials and extensive information about MPI:


### 7.5 Compiling MPI programs on Blue Gene/P

The XL C/C++ and Fortran95 family of compilers provide several different commands that you can use to invoke the compiler with the `bgxl` prefix, for example `bgxlc`. These are the most commonly used commands to invoke the different compilers. However, when using the standard commands, you must explicitly invoke all the libraries that are required.

Alternatively, simple scripts are available to compile or link MPI programs. These scripts include the default flags and libraries and can handle alternative compilers and the associated flags and libraries.

The following flags and environmental variables are provided with the scripts:

- `includedir, libdir` Directories that contain an installed mpich2
- `prefix, execprefix` Often used to define includedir and libdir
- `CC` C compiler
- `CXX` C++ compiler
- `FC` Fortran 77 compiler
- `MPI_CFLAGS` Any special flags needed to compile
- `MPI_LDFLAGS` Any special flags needed to link
- `MPILIBNAME` The name of the MPI library
- `MPI_OTHERLIBS` Other libraries that are needed to link

The compiler and the scripts also provide corresponding "r" versions of most invocation commands to instruct the compiler to link and bind object files to threadsafe components and libraries, and produce threadsafe object code for compiler-created data and procedures. The following scripts are provided in the `/bgsys/drivers/ppcfloor/comm/bin` directory:

- `mpicc` GNU C compiler with MPI libraries and default compiler flags
- `mpich2version` Script to provide MPICH2 version information
- `mpicxx` GNU C++ compiler with MPI libraries and default compiler flags
- `mpif77` GNU Fortran 77 (gfortran) compiler with MPI libraries and default compiler flags
- `mpixlc` IBM XL C compiler with MPI libraries and no default compiler flags
Example 7-14 shows a makefile that does not use the scripts and requires the programmer to identify all the libraries and include files.

```
Example 7-14   Makefile with explicit reference to libraries and include files

BGP_FLOOR   = /bgsys/drivers/ppcfloor
BGP_IDIRS   = -I$(BGP_FLOOR)/arch/include -I$(BGP_FLOOR)/comm/include
BGP_LIBS    = -L$(BGP_FLOOR)/comm/lib -lmpich.cnk -L$(BGP_FLOOR)/comm/lib
             -ldcmfcoll.cnk -ldcmf.cnk -lpthread -lrt
XL         = /opt/ibmcmp/xlf/bg/11.1/bin/bgxlf
EXE         =  fhello
OBJ         =  hello.o
SRC         =  hello.f
FLAGS       =  -O3 -qarch=450 -qtune=450 -I$(BGP_FLOOR)/comm/include
FLD         =  -O3 -qarch=450 -qtune=450

$(EXE): $(OBJ)
    ${XL} $(FLAGS) $(BGP_IDIRS) -o $(EXE) $(OBJ) $(BGP_LIBS)
$(OBJ): $(SRC)
    $(XL) $(FLAGS) $(BGP_IDIRS) -c $(SRC)

clean:
    rm hello.o fhello
```

Alternatively, Example 7-15 on page 80 uses MPI scripts, which create a simpler makefile. The MPI scripts also handle the proper order in which libraries should be called.

Note: When you invoke the previous scripts, if you do not specify -O (specify whether to optimize code during compilation), the default is set to -O0.
Example 7-15  Use of MPI script mpixlf77

XL = /bgsys/drivers/ppcfloor/comm/bin/mpixlf77
EXE = fhello
OBJ = hello.o
SRC = hello.f
FLAGS = -O3 -qarch=450 -qtune=450
FLD = -O3 -qarch=450 -qtune=450

$(EXE): $(OBJ)
# $(XL) $(FLAGS) $(BGP_LDIRS) -o $(EXE) $(OBJ) $(BGP_LIBS)
$(XL) $(FLAGS) -o $(EXE) $(OBJ) $(BGP_LIBS)

$(OBJ): $(SRC)
$(XL) $(FLAGS) $(BGP_IDIRS) -c $(SRC)

clean:
rm hello.o fhello

7.6 MPI communications performance

Communications performance is an important aspect when running parallel applications, particularly, when running on a distributed memory system such as the Blue Gene/P system. On both the Blue Gene/L and Blue Gene/P systems, instead of implementing a single type of network capable of transporting all required protocols, these two systems have separate networks for different types of communications.

Usually the following measurements provide information about the network and can be used to look at the parallel performance of applications:

- **Bandwidth**  The number of MB of data that can be sent from one node to another node in one second
- **Latency**  The amount of time it takes for the first byte that is sent from one node to reach its target node

The values for bandwidth and latency provide information about communication.

Here we illustrate two cases. The first case corresponds to a benchmark that involves a single transfer. The second case corresponds to a collective as defined in the “Intel® MPI Benchmarks” (see the URL that follows). “Intel MPI Benchmarks” was formerly known as “Pallas MPI Benchmarks” (PMB-MPI1 for MPI1 standard functions only). Intel MPI Benchmarks - MPI1 provides a set of elementary MPI benchmark kernels.

For more details, see the product documentation included in the package that you can download from the following Intel Web page:

7.6.1 MPI point-to-point

In the Intel MPI Benchmarks, a single transfer corresponds to the PingPong and PingPing benchmarks. We illustrate a comparison between the Blue Gene/L and Blue Gene/P systems for the case of PingPong. This benchmark illustrates a single message that is transferred between two MPI tasks, in our case, on two different nodes.

To run this benchmark, we used the Intel MPI Benchmark Suite Version 2.3, MPI-1 part. On the Blue Gene/L system, the benchmark was run in coprocessor mode. (See *Unfolding the IBM eServer Blue Gene Solution*, SG24-6686.) On the Blue Gene/P system, we used the SMP node mode. `mpirun` was invoked as shown in Example 7-16 and Example 7-17 for the Blue Gene/L and Blue Gene/P systems respectively.

**Example 7-16  mpirun on the Blue Gene/L system**

```bash
mpirun -nofree -timeout 120 -verbose 1 -mode CO -env "BGL_APP_L1_WRITE_THROUGH=0 BGL_APP_L1_SWOA=0" -partition R000 -cwd /bglscratch/pallas -exe /bglscratch/pallas/IMB-MPI1.4MB.perf.rts -args "-msglen 4194304.txt -npmin 512 PingPong" | tee IMB-MPI1.4MB.perf.PingPong.4194304.512.out) >> run.IMB-MPI1.4MB.perf.PingPong.4194304.512.out 2>&1
```

**Example 7-17  mpirun on the Blue Gene/P system**

```bash
mpirun -nofree -timeout 300 -verbose 1 -np 512 -mode SMP -partition R01-M1 -cwd /bgusr/BGTH_BGP/test512nDD2BGP/pallas/pall512DD2SMP/bgpdd2sys1-R01-M1 -exe /bgusr/BGTH_BGP/test512nDD2BGP/pallas/pall512DD2SMP/bgpdd2sys1-R01-M1/IMB-MPI1.4MB.perf.rts -args "-msglen 4194304.txt -npmin 512 PingPong" | tee IMB-MPI1.4MB.perf.PingPong.4194304.512.out) >> run.IMB-MPI1.4MB.perf.PingPong.4194304.512.out 2>&1
```

Figure 7-4 shows the bandwidth on the torus network as a function of the message size, for one simultaneous pair of nearest neighbor communications. The protocol switch from short to eager is visible in both cases, where the eager to rendezvous switch is most pronounced on the Blue Gene/L system (see the asterisks (*)). This figure also shows the improved performance on the Blue Gene/P system (see the diamonds).
7.6.2 MPI collective

In the Intel MPI Benchmarks, the collective benchmarks correspond to the Bcast, Allgather, Allgatherv, Alltoall, Reduce, Reduce_scatter, Allreduce, and Barrier benchmarks. We illustrate a comparison between the Blue Gene/L and Blue Gene/P systems for the case of Allreduce, which is a popular collective that is used in certain scientific applications. These benchmarks measure the message-passing power of a system as well as the quality of the implementation.

To run this benchmark, we used the Intel MPI Benchmark Suite Version 2.3, MPI-1 part. On the Blue Gene/P system, the benchmark was run in coprocessor mode. On the Blue Gene/P system, we used SMP node mode. mpirun was invoked as shown in Example 7-18 and Example 7-19 for the Blue Gene/L and Blue Gene/P systems, respectively.

Example 7-18  mpirun on the Blue Gene/L system

```bash
mpirun -nofree -timeout 120 -verbose 1 -mode CO -env "BGL_APP_L1_WRITE_THROUGH=0 BGL_APP_L1_SWOA=0" -partition R000 -cwd /bglscratch/BGTH/testsmall512nodeBGL/pallas -exe /bglscratch/BGTH/testsmall512nodeBGL/pallas/IMB-MPI1.4MB.perf.rts -args "-msglen 4194304.txt -npmin 512 Allreduce" | tee IMB-MPII.4MB.perf.Allreduce.4194304.512.out) >> run.IMB-MPII.4MB.perf.Allreduce.4194304.512.out 2>&1
```

Example 7-19  mpirun on the Blue Gene/P system

```bash
```

Collective operations are more efficient on the Blue Gene/P system. You should try to use collective operations instead of point-to-point communication wherever possible. The overhead for point-to-point communications is much larger than for collectives. Unless all of your point-to-point communication is purely to the nearest neighbor, it is difficult to avoid network congestion on the torus network.

Alternatively, collective operations can use the barrier (global interrupt) network or the torus network. If they run over the torus network, they can still be optimized by using specially designed communication patterns that achieve optimum performance. Doing this manually with point-to-point operations is possible in theory, but in general, the implementation in the Blue Gene/P MPI library offers superior performance.

With point-to-point communication, the goal of reducing the point-to-point Manhattan distances necessitates a good mapping of MPI tasks to the physical hardware. For collectives, mapping is equally important because most collective implementations prefer certain communicator shapes to achieve optimum performance. In general, collectives using “rectangular” subcommunicators (with the ranks organized in lines, planes, or cubes) will out perform “irregular” subcommunicators” (any communicator that is not rectangular). Refer to Appendix E, “Mapping” on page 303, which illustrates the technique of mapping.
Similar to point-to-point communications, collective communications also work best if you do not use complicated derived data types, and if your buffers are aligned to 16-byte boundaries. While the MPI standard explicitly allows for MPI collective communications to take place at the same time as point-to-point communications (on the same communicator), generally we do not recommend this for performance reasons.

Table 7-1 summarizes the MPI collectives that have been optimized on the Blue Gene/P system, together with their performance characteristics when executed on the various networks of the Blue Gene/P system.

<table>
<thead>
<tr>
<th>MPI routine</th>
<th>Condition</th>
<th>Network</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Barrier</td>
<td>MPI_COMM_WORLD</td>
<td>Barrier (global interrupt) network</td>
<td>1.2 μs</td>
</tr>
<tr>
<td>MPI_Barrier</td>
<td>Any communicator</td>
<td>Torus network</td>
<td>30 μs</td>
</tr>
<tr>
<td>MPI_Broadcast</td>
<td>MPI_COMM_WORLD</td>
<td>Collective network</td>
<td>817 MBps</td>
</tr>
<tr>
<td>MPI_Broadcast</td>
<td>Rectangular communicator</td>
<td>Torus network</td>
<td>934 MBps</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>MPI_COMM_WORLD</td>
<td>Collective network</td>
<td>778 MBps</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>MPI_COMM_WORLD fixed-point</td>
<td>Collective network</td>
<td>98 MBps</td>
</tr>
<tr>
<td>MPI_Alltoall[v]</td>
<td>Any communicator</td>
<td>Torus network</td>
<td>84-97% peak</td>
</tr>
<tr>
<td>MPI_Allgatherv</td>
<td>N/A</td>
<td>Torus network</td>
<td>same as broadcast</td>
</tr>
</tbody>
</table>

Figure 7-5 on page 84 and Figure 7-6 on page 85 show a comparison between the Blue Gene/L and Blue Gene/P systems for the MPI_Allreduce() type of communication on integer data types with the sum operation.
Figure 7-5  MPI_Allreduce() integer sum wall time performance on 512 nodes.

Figure 7-6 on page 85 shows MPI_Allreduce() integer sum bandwidth performance on 512 nodes.
Figure 7-6  MPI_Allreduce() integer sum bandwidth performance on 512 nodes
7.7 OpenMP

The OpenMP API is supported on the Blue Gene/P system for shared-memory parallel programming in C/C++ and Fortran. This API has been jointly defined by a group of hardware and software vendors and has evolved as a standard for shared-memory parallel programming.

OpenMP consists of a collection of compiler directives and a library of functions that can be invoked within an OpenMP program. This combination provides a simple interface for developing parallel programs on shared-memory architectures. In the case of the Blue Gene/P system, it allows the user to exploit the SMP mode on each compute node. Multi-threading is now enabled on the Blue Gene/P system. Using OpenMP, the user can have access to data parallelism as well as functional parallelism.

For additional information, refer to the official OpenMP Web site at:
http://www.openmp.org/

7.7.1 OpenMP implementation for Blue Gene/P

The Blue Gene/P system supports shared-memory parallelism on single nodes. The XL compilers support the following constructs:

- Full support for OpenMP 2.5 standard
- Support for the use of the same infrastructure as the OpenMP on AIX® and Linux
- Interoperability with MPI
  - MPI at outer level, across the compute nodes
  - OpenMP at the inner level, within a compute node
- Autoparallelization based on the same parallel execution framework
  - Enables autoparallelization as one of the loop optimizations
- Thread-safe version for each compiler
  - bgxlf_r
  - bgxlc_r
  - bgcc_r
- Use of the thread-safe compiler version with any threaded, OpenMP, or SMP application
  - -qsmp must be used on OpenMP or SMP applications.
  - -qsmp by itself automatically parallelizes loops.
  - -qsmp=omp parallelizes based on OpenMP directives in the code.
  - Shared-memory model is on the Blue Gene/P system.

7.7.2 Selected OpenMP compiler directives

The latest set of OpenMP compiler directives is documented in the OpenMP ARB release Version 2.5 specification. Version 2.5 combines Fortran and C/C++ specifications into a single specification. It also fixes inconsistencies. We summarize some of the directives as follows:

- **parallel** Directs the compiler for that section of the code to be executed in parallel by multiple threads
- **for** Directs the compiler to execute a for loop with independent iterations; iterations can be executed by different threads in parallel
- **parallel for** The syntax for parallel loops
sections Directs the compiler of blocks of non-iterative code that can be executed in parallel

parallel sections Syntax for parallel sections

critical Restrictions the following section of the code to be executed by a single thread at a time

single Directs the compiler to execute a section of the code by a single thread

Parallel operations are often expressed in C/C++ and Fortran95 programs as for loops as shown in Example 7-20.

Example 7-20 for loops in Fortran and C

```fortran
for (i = start; i < num; i += end)
{ array[i] = 1; m[i] = c; }
```
or

```fortran
integer i, n, sum
sum = 0
do 5 i = 1, n
sum = sum + i
5 continue
```

The compiler can automatically locate and, where possible, parallelize all countable loops in your program code in the following situations:

- There is no branching into or out of the loop.
- An increment expression is not within a critical section.
- A countable loop is automatically parallelized only if all of the following conditions are met:
  - The order in which loop iterations start or end does not affect the results of the program.
  - The loop does not contain I/O operations.
  - Floating-point reductions inside the loop are not affected by round-off error, unless the -qnostrict option is in effect.
  - The -qnostrict_induction compiler option is in effect.
  - The -qsmp=auto compiler option is in effect.
  - The compiler is invoked with a thread-safe compiler mode.

In the case of C/C++ programs, OpenMP is invoked via pragmas as shown in Example 7-21.

Pragma: The word *pragma* is short for *pragmatic information.* Pragma is a way to communicate information to the compiler:

```fortran
#pragma omp <rest of pragma>
```

Example 7-21 pragma usage

```fortran
#pragma omp parallel for
   for (i = start; i < num; i += end)
   { array[i] = 1; m[i] = c; }
```
The for loop must not contain statements, such as the following examples, that allow the loop to be exited prematurely:

- `break`
- `return`
- `exit`
- `go to labels outside the loop`

In a for loop, the master thread creates additional threads. The loop is executed by all threads, where every thread has its own address space that contains all of the variables the thread can access. Such variables might be:

- Static variables
- Dynamically allocated data structures in the heap
- Variables on the run-time stack

In addition, variables must be defined according to the type. Shared variables have the same address in the execution context of every thread. It is important to understand that all threads have access to shared variables. Alternatively, private variables have a different address in the execution memory of every thread. A thread can access its own private variables, but it cannot access the private variable of another thread.

**Pragma parallel:** In the case of the parallel for pragma, variables are shared by default, with exception of the loop index.

Example 7-22 shows a simple Fortran95 example that illustrates the difference between private and shared variables.

**Example 7-22  Fortran example using the parallel do directive**

```fortran
program testmem
    integer n
    parameter (n=2)
    parameter (m=1)
    integer a(n), b(m)
    !$OMP parallel do
    do i = 1, n
        a(i) = i
    enddo
    write(6,*),'Done: testmem'
end
```

In Example 7-22, no variables are explicitly defined as neither private nor shared. In this case, by default, the compiler assigns the variable that is used for the do-loop index as private. The rest of the variables are shared. Figure 7-7 on page 89 illustrates both private and shared variables as shown in *Parallel Programming in C with MPI and OpenMP*. In this figure, the blue and yellow arrows indicate which variables are accessible by all the threads.
7.7.3 Selected OpenMP compiler functions

The following functions are selected for the OpenMP compiler:

- **omp_get_num_procs**: Returns the number of processors
- **omp_get_num_threads**: Returns the number of threads in a particular parallel region
- **omp_get_thread_num**: Returns the thread identification number
- **omp_set_num_threads**: Allocates numbers of threads for a particular parallel region

7.7.4 Performance

To illustrate the effect of selected OpenMP compiler directives and the implications in terms of performance of a particular do loop, we chose the $\pi$ programs presented in *Parallel Programming in C with MPI and OpenMP* and apply them to the Blue Gene/P system. These simple examples illustrate how to use these directives and some of the implications in selecting a particular directive over another directive. Example 7-23 shows a simple program to compute $\pi$.

**Example 7-23  Sequential version of the pi.c program**

```c
int main(argc, argv)
    int argc;
    char *argv[];
{
    long n, i;
    double area, pi, x;
    n    = 1000000000;
    area = 0.0;
    for (i = 0; i < n; i++) {
        x = (i+0.5)/n;
        area += 4.0 / (1.0 + x*x);
```
\[ pi = \frac{\text{area}}{n}; \]
\[ \text{printf ("Estimate of pi: %7.5f\n", pi);} \]

The first way to parallelize this code is to include an OpenMP directive to parallelize the for loop as shown in Example 7-24.

**Example 7-24  Simple use of parallel for loop**

```
#include <omp.h>

long long timebase(void);

int main(argc, argv)
int argc;
char *argv[];
{
    int num_threads;
    long n, i;
    double area, pi, x;
    long long time0, time1;
    double cycles, sec_per_cycle, factor;
    n    = 1000000000;
    area = 0.0;
    time0 = timebase();
    #pragma omp parallel for private(x)
    for (i = 0; i < n; i++) {
        x = (i+0.5)/n;
        area += 4.0 / (1.0 + x*x);
    }
    pi = area / n;
    printf ("Estimate of pi: %7.5f\n", pi);
    time1 = timebase();
    cycles = time1 - time0;
    factor = 1.0/850000000.0;
    sec_per_cycle = cycles * factor;
    printf("Total time %lf \n",sec_per_cycle, "Seconds \n");
}
```

Unfortunately this simple approach creates a race condition when computing the area. While different threads compute and update the value of the area, other threads might be computing and updating area as well, therefore producing the wrong results. This particular race condition can be solved in two ways. One way is to use a critical pragma to ensure mutual exclusion among the threads, and the other way is to use the reduction clause.

Example 7-25 illustrates use of the critical pragma.

**Example 7-25  Usage of critical pragma**

```
#include <omp.h>

long long timebase(void);

int main(argc, argv)
int argc;
```
char *argv[];
{
    int num_threads;
    long n, i;
    double area, pi, x;
    long long time0, timel;
    double cycles, sec_per_cycle, factor;
    n    = 1000000000;
    area = 0.0;
    time0 = timebase();
    #pragma omp parallel for private(x)
    for (i = 0; i < n; i++) {
        x = (i+0.5)/n;
        #pragma omp critical
        area += 4.0 / (1.0 + x*x);
    }
    pi = area / n;
    printf("Estimate of pi: %7.5f\n", pi);
    timel = timebase();
    cycles = timel - time0;
    factor = 1.0/850000000.0;
    sec_per_cycle = cycles * factor;
    printf("Total time %lf \n",sec_per_cycle, "Seconds \n");
}

Example 7-26 corresponds to the reduction clause.

Example 7-26  Usage of the reduction clause

#include <omp.h>

long long timebase(void);

int main(argc, argv)
int argc;
char *argv[];
{
    int num_threads;
    long n, i;
    double area, pi, x;
    long long time0, timel;
    double cycles, sec_per_cycle, factor;
    n    = 1000000000;
    area = 0.0;
    time0 = timebase();
    #pragma omp parallel for private(x) reduction(+: area)
    for (i = 0; i < n; i++) {
        x = (i+0.5)/n;
        area += 4.0 / (1.0 + x*x);
    }
    pi = area / n;
    printf("Estimate of pi: %7.5f\n", pi);
    timel = timebase();
    cycles = timel - time0;
    factor = 1.0/850000000.0;
sec_per_cycle = cycles * factor;
printf("Total time %lf \n", sec_per_cycle, "Seconds \n");
}

To compile these two programs on the Blue Gene/P system, the makefile for pi_critical.c shown in Example 7-27 can be used. A similar makefile can be used for the program illustrated in Example 7-26 on page 91.

Example 7-27 Makefile for the pi_critical.c program

BGP_FLOOR = /bgsys/drivers/ppcfloor
BGP_IDIRS = -I$(BGP_FLOOR)/arch/include -I$(BGP_FLOOR)/comm/include
BGP_LIBS = -L$(BGP_FLOOR)/comm/lib -L$(BGP_FLOOR)/runtime/SPI -lmpich.cnk
-ldcmfcoll.cnk -ldcmf.cnk -lrt -lSPI.cna -lpthread

XL = /opt/ibmcmp/vac/bg/9.0/bin/bgxlc_r

EXE = pi_critical_bgp
OBJ = pi_critical.o
SRC = pi_critical.c
FLAGS = -O3 -qsmp=omp:noauto -qthreaded -qarch=450 -qtune=450
-I$(BGP_FLOOR)/comm/include

FLD = -O3 -qarch=450 -qtune=450

$(EXE): $(OBJ)
   ${XL} $(FLAGS) -o $(EXE) $(OBJ) timebase.o $(BGP_LIBS)

$(OBJ): $(SRC)
   ${XL} $(FLAGS) $(BGP_IDIRS) -c $(SRC)

clean:
   rm pi_critical.o pi_critical_bgp

Table 7-2 illustrates the performance improvement by using the reduction clause.

Table 7-2 Parallel performance using critical pragma versus reduction clause

<table>
<thead>
<tr>
<th>Threads</th>
<th>Using critical pragma</th>
<th>Using reduction clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POWER4 1.0 GHz</td>
<td>586.37</td>
<td>20.12</td>
</tr>
<tr>
<td>POWER5 1.9 GHz</td>
<td>145.03</td>
<td>5.22</td>
</tr>
<tr>
<td>POWER6™ 4.7 GHz</td>
<td>180.80</td>
<td>4.78</td>
</tr>
<tr>
<td>Blue Gene/P</td>
<td>560.08</td>
<td>12.80</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POWER4 1.0 GHz</td>
<td>458.84</td>
<td>10.08</td>
</tr>
<tr>
<td>POWER5 1.9 GHz</td>
<td>374.10</td>
<td>2.70</td>
</tr>
<tr>
<td>POWER6 4.7 GHz</td>
<td>324.71</td>
<td>2.41</td>
</tr>
<tr>
<td>Blue Gene/P</td>
<td>602.62</td>
<td>6.42</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
For more in-depth information with additional examples, we recommend you read *Parallel Programming in C with MPI and OpenMP*\(^\text{30}\). In this section, we selected to illustrate only the \( \pi \) program.

<table>
<thead>
<tr>
<th>Threads</th>
<th>Using critical pragma</th>
<th>Using reduction clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>POWER4 1.0 GHz</td>
<td>552.54</td>
<td>5.09</td>
</tr>
<tr>
<td>POWER5 1.9 GHz</td>
<td>428.42</td>
<td>1.40</td>
</tr>
<tr>
<td>POWER6 4.7 GHz</td>
<td>374.51</td>
<td>1.28</td>
</tr>
<tr>
<td>Blue Gene/P</td>
<td>582.95</td>
<td>3.24</td>
</tr>
</tbody>
</table>
Developing applications with IBM XL compilers

With the IBM XL family of optimizing compilers, you can develop C, C++, and Fortran applications for the Blue Gene/P system. This family comprises the following products, which we refer to in this chapter as Blue Gene XL compilers:

- XL C/C++ Advanced Edition V9.0 for Blue Gene
- XL Fortran Advanced Edition V11.1 for Blue Gene

The information that we present in this chapter is specific to the Blue Gene/P supercomputer. It does not include general XL compiler information. For complete documentation about these compilers, refer to the libraries at the following Web addresses:

- XL C/C++
- XL Fortran

In this chapter, we discuss specific considerations for developing, compiling, and optimizing C/C++ and Fortran applications for the Blue Gene/P PowerPC 450 processor and a Single Instruction Multiple Data (SIMD), double-precision floating-point multiply add unit (double floating-point multiply add (FMA)). The following topics are discussed:

- Compiler overview
- Compiling and linking applications on Blue Gene/P
- Default compiler options
- Unsupported options
- Support for pthreads and OpenMP
- Creation of libraries on Blue Gene/P
- XL run-time libraries
- Mathematical Acceleration Subsystem libraries 104
- Engineering and Scientific Subroutine Library libraries 104
- Python support 104
- Tuning your code for Blue Gene/P 105
Tips for optimizing applications 109
Identifying performance bottlenecks 110

Several documents cover part of the material presented in this chapter. In addition to the XL family of compilers manuals that we reference throughout this chapter, we recommend that you read the following documents:

- Unfolding the IBM eServer Blue Gene Solution, SG24-6686
- IBM System Blue Gene Solution: Application Development, SG24-7179

We also recommend that you read the article by Mark Mendell, “Exploiting the Dual Floating Point Units in Blue Gene/L,” which provides detailed information about the SIMD functionality in the XL family of compilers. You can find this article on the Web at:

http://www-1.ibm.com/support/docview.wss?uid=swg27007511
8.1 Compiler overview

The Blue Gene/P system uses the same XL family of compilers as the Blue Gene/L system. The Blue Gene/P system supports cross-compilation, and the compilers run on the front end node. The compilers for the Blue Gene/P system have specific optimizations for its architecture. In particular, the XL family of compilers generate code appropriate for the double floating-point unit (FPU) of the Blue Gene/P system.

The Blue Gene/P system has compilers for the C, C++, and Fortran programming languages. The compilers on the Blue Gene/P system take advantage of the double FPU available on the Blue Gene/P system. They also incorporate code optimizations specific to the Blue Gene/P instruction scheduling and memory hierarchy characteristics.

In addition to the XL family of compilers, the Blue Gene/P system supports a version of the GNU compilers for C, C++, and Fortran. These compilers do not generate highly optimized code for the Blue Gene/P system. In particular, they do not automatically generate code for the double FPUs, and they do not support OpenMP.

Tools that are commonly associated with the GNU compilers (known as binutils) are supported in the Blue Gene/P system. The same set of compilers and tools is used for both Linux and the Blue Gene/P proprietary operating system. The Blue Gene/P system supports the execution of Python-based user applications.

The GNU compiler toolchain also provides the dynamic linker, which is used both by Linux and the Blue Gene/P proprietary operating system to support dynamic objects. The toolchain is tuned to support both environments. The GNU "aux vector" technique is employed to pass kernel-specific information to the C library when tuning must be specific to one of the kernels.

8.2 Compiling and linking applications on Blue Gene/P

In this section, we provide information about compiling and linking applications that run on the Blue Gene/P system. For complete information about compiler and linker options, see the following documents available on the Web:

- "XL C/C++ Compiler Reference"
- "XL Fortran User Guide"

You can also find these documents in the following directories:

- /opt/ibmcmp/vaccpp/bg/9.0/doc (C and C++)
- /opt/ibmcmp/xlf/bg/11.1/doc (Fortran)

The compilers can be found in the following directories:

- /opt/ibmcmp/vac/bg/9.0/bin
- /opt/ibmcmp/vaccpp/bg/9.0/bin
- /opt/ibmcmp/xlf/bg/11.1/bin
The Blue Gene/P release includes the following differences for compiling and linking applications:

- Blue Gene/P compiler wrapper names have changed:
  - blrts_ is replaced by bg.
  - xlf 11.1, vacpp 9.0, and vac 9.0 on the Blue Gene/L system support both blrts_ and bg.
- -qarch=450d/450 is for the Blue Gene/P system, and 440d/440 is for the Blue Gene/L system.

8.3 Default compiler options

Compilations most commonly occur on the front end node. The resulting program can run on the Blue Gene/P system without manually copying the executable to the service node. See Chapter 9, “Running and debugging applications” on page 137, and Chapter 11, “mpirun” on page 167, to learn how to run programs on the Blue Gene/P system.

The script or makefile that you use to invoke the compilers should have certain compiler options. Specifically the architecture-specific options, which optimize processing for the Blue Gene/P 450d processor architecture, should be set to the following defaults:

- -qarch=450
  This option generates code for a single FPU only, but it can give correct results if invalid code is generated by -qarch=450d. You can follow up with -qarch=450d when optimizing performance via more aggressive compilation.

- -qtune=450
  Optimizes object code for the 450 family of processors. Single FPU only.

- -qcache=level=1:type=i:size=32:line=32:assoc=64:cost=8
  Specifies the L1 instruction cache configuration for the Blue Gene/P architecture to allow greater optimization with options -O4 and -O5.

- -qcache=level=1:type=d:size=32:line=32:assoc=64:cost=8
  Specifies the L1 data cache configuration for the Blue Gene/P architecture to allow greater optimization with options -O4 and -O5.

- -qcache=level=2:type=c:size=4096:line=128:assoc=8:cost=40
  Specifies the L2 (combined data and instruction) cache configuration for the Blue Gene/P architecture to allow greater optimization with options -O4 and -O5.

- -qnoautoconfig
  Allows code to be cross-compiled on other machines at optimization levels -O4 or -O5, by preserving the Blue Gene/P architecture-specific options.

Scripts are already available that do much of this for you. They reside in the same bin directory as the compiler binary (/opt/ibmcmp/xlf/bg/11.1/bin or /opt/ibmcmp/vacpp/bg/9.0/bin or /opt/ibmcmp/vac/bg/9.0/bin). Table 8-1 on page 99 lists the names.
Table 8-1 Scripts available in the bin directory for compiling and linking

<table>
<thead>
<tr>
<th>Language</th>
<th>Script name or names</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>bgc89, bgc99, bgcc, bgxlc bgc89_r, bgc99_r bgcc_r, bgxlc_r</td>
</tr>
<tr>
<td>C++</td>
<td>bgxlC++, bgxlC++_r, bgxlC, bgxlC_r</td>
</tr>
<tr>
<td>Fortran</td>
<td>bgf2003, bgf95, bgxlf2003, bgxlf90_r, bgxlf_r, bgfl77, bgfort77, bgxlf2003_r, bgxlf95, bgf90, bgfl, bgxlf90, bgxlf95_r</td>
</tr>
</tbody>
</table>

**Important:** The double FPU does not generate exceptions. Therefore, the \(-qflttrap\) option is invalid with the 450d processor. Instead you should reset the 450d processor to \(-qarch=450\).

### 8.4 Unsupported options

The following compiler options, although available for other IBM systems, are not supported by the Blue Gene/P hardware. Therefore, do not use them:

- \(-q64\): The Blue Gene/P system uses a 32-bit architecture; you cannot compile in 64-bit mode.
- \(-qaltivec\): The 450 processor does not support VMX instructions or vector data types.

### 8.5 Support for pthreads and OpenMP

The Blue Gene/P system supports shared-memory parallelism on single nodes. The XL compilers support the following constructs:

- Full support for the OpenMP 2.5 standard\(^{31}\)
- Use of the same infrastructure as the OpenMP that is supported on AIX and Linux
- Interoperability with MPI
  - MPI at outer level, across the compute nodes
  - OpenMP at the inner level, within a compute node
- Autoparallelization based on the same parallel execution framework
  Enablement of autoparallelization as one of the loop optimizations
- All the thread-safe scripts of the compiler end in _r; here are some examples:
  - bgxlf_r
  - bgxlc_r
  - bgxlC_r
  - bgcc_r
  
The thread-safe compiler version should be used with any threaded, OpenMP, or SMP application.
## Thread-safe libraries

Thread-safe libraries ensure that data access and updates are synchronized between threads.

- Usage of `-qsmp` OpenMP and pthreaded applications
  - `-qsmp` by itself automatically parallelizes loops.
  - `-qsmp=omp` automatically parallelizes based on OpenMP directives in the code.
  - `-qsmp=omp:noauto` should be used when parallelizing codes manually. It prevents the compiler from trying to automatically parallelize loops.

### Note

- `-qsmp` must be used only with thread-safe compiler mode invocations such as `xlc_r`. These invocations ensure that the pthreads, xlsmp, and thread-safe versions of all default run-time libraries are linked to the resulting executable. See the language reference for more details about the `-qsmp` suboptions at: [http://publib.boulder.ibm.com/infocenter/comphelp/v8v10l/index.jsp](http://publib.boulder.ibm.com/infocenter/comphelp/v8v10l/index.jsp)

### 8.6 Creation of libraries on Blue Gene/P

On Blue Gene/P three types of libraries can be created:

- Static libraries
- Shared libraries
- Dynamically loaded libraries

Static libraries are loaded into the program when the program is built. Static libraries are embedded as part of the Blue Gene/P executable that resides on the front end node. Example 8-1 illustrates how to create a static library on Blue Gene/P using the XL family of compilers.

#### Example 8-1  Static library creation using the XL compiler

```bash
#!/bin/csh
#
# Compile with the XL compiler
/opt/ibmcmp/vac/bg/9.0/bin/bgxlc -c pi.c
/opt/ibmcmp/vac/bg/9.0/bin/bgxlc -c main.c
#
# Create library
/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc-bgp-linux-ar rcs libpi.a pi.o
#
# Create executable
/opt/ibmcmp/vac/bg/9.0/bin/bgxlc -o pi main.o -L. -lpi
```
Example 8-2 shows the same procedure using the GNU collection of compilers.

**Example 8-2  Static library creation using the GNU compiler**

```bash
#!/bin/csh
#
# Compile with the GNU compiler
/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc-bgp-linux-gcc -c pi.c
/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc-bgp-linux-gcc -c main.c
#
# Create library
/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc-bgp-linux-ar rcs libpi.a pi.o
#
# Create executable
/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc-bgp-linux-gcc -o pi main.o -L. -lpi
```

On the other hand, shared libraries are loaded at execution time and shared among different executables. Dynamically loaded libraries (DLL) are loaded at any time that the application is executing. Example 8-3 shows the procedure for creating shared libraries using the XL family of compilers.

**Note:** `-qnostaticlink` indicates to build a dynamic binary, but by default the static libgcc.a is linked in. To indicate that the shared version of libgcc should be linked in, also specify `-qnostaticlink=libgcc`. For example, `/opt/ibmcmp/vacpp/bg/9.0/bin/bgxlc -o hello hello.c -qnostaticlink -qnostaticlink=libgcc`

**Example 8-3  Shared library creation using the XL compiler**

```bash
#!/bin/csh
#
# Use XL to create shared library
/opt/ibmcmp/vac/bg/9.0/bin/bgxlc -qpic -c libpi.c
/opt/ibmcmp/vac/bg/9.0/bin/bgxlc -qpic -c main.c
#
# Create shared library
/opt/ibmcmp/vac/bg/9.0/bin/bgxlc -qmkshrobj -qnostaticlink -qnostaticlink=libgcc -Wl,-soname, \
libpi.so.0 -o libpi.so.0.0 libpi.o
#
# Set up the soname
ln -sf libpi.so.0.0 libpi.so.0
#
# Create a linker name
ln -sf libpi.so.0 libpi
#
# Create executable
/opt/ibmcmp/vac/bg/9.0/bin/bgxlc -o pi main.o -L. -lpi -qnostaticlink -qnostaticlink=libgcc
```
Example 8-4 illustrates the same procedure with the GNU collection of compilers.

Example 8-4   Shared library creation using the GNU compiler

#!/bin/csh
#
# Compile with the GNU compiler
/bgsys/drivers/ppcfloor-gnu-linux/bin/powerpc-bgp-linux-gcc -fPIC -c libpi.c
/bgsys/drivers/ppcfloor-gnu-linux/bin/powerpc-bgp-linux-gcc -fPIC -c main.c
#
# Create shared library
/bgsys/drivers/ppcfloor-gnu-linux/bin/powerpc-bgp-linux-gcc -shared \ 
-Wl,-soname,libpi.so.0 -o libpi.so.0.0 libpi.o -l
#
# Set up the soname
ln -sf libpi.so.0.0 libpi.so.0
#
# Create a linker name
ln -sf libpi.so.0 libpi.so
#
# Create executable
/bgsys/drivers/ppcfloor-gnu-linux/bin/powerpc-bgp-linux-gcc -o pi main.o -L. -lpi -dynamic

8.7 XL run-time libraries

The libraries listed in Table 8-2 are linked into your application automatically by the XL linker when you create your application.

**MASS libraries:** The exception to this statement is for the libmassv.a file (the Mathematical Acceleration Subsystem (MASS) libraries). This file must be explicitly specified on the linker command. See 8.8, “Mathematical Acceleration Subsystem libraries” on page 103 for information about the MASS libraries.

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>libibmc++.a,</td>
<td>IBM C++ library</td>
</tr>
<tr>
<td>libibmc++.so</td>
<td></td>
</tr>
<tr>
<td>libxlf90.a,</td>
<td>IBM XLF run-time library</td>
</tr>
<tr>
<td>libxlf90.so</td>
<td></td>
</tr>
<tr>
<td>libxlfmath.a,</td>
<td>IBM XLF stubs for math routines in system library libm, for example, _sin()</td>
</tr>
<tr>
<td>libxlfmath.so</td>
<td>for \sin(), _cos() for \cos(), and so on</td>
</tr>
<tr>
<td>libxlfpmt4.a,</td>
<td>IBM XLF to be used with -qautobdl=dbl14 (promote floating-point objects that</td>
</tr>
<tr>
<td>libxlfpmt4.so</td>
<td>are single precision)</td>
</tr>
<tr>
<td>libxlfpad.a,</td>
<td>IBM XLF run-time routines to be used with -qautobdl=dblpad (promote</td>
</tr>
<tr>
<td>libxlfpad.so</td>
<td>floating-point objects and pad other types if they can share storage with</td>
</tr>
<tr>
<td></td>
<td>promoted objects)</td>
</tr>
<tr>
<td>libxlfpmt8.a,</td>
<td>IBM XLF run-time routines to be used with -qautobdl=dbl18 (promote</td>
</tr>
<tr>
<td>libxlfpmt8.so</td>
<td>floating-point objects that are double precision)</td>
</tr>
<tr>
<td>libxl.a</td>
<td>IBM low-level run-time library</td>
</tr>
</tbody>
</table>
8.8 Mathematical Acceleration Subsystem libraries

The MASS consists of libraries of tuned mathematical intrinsic functions that are available in versions for the AIX and Linux machines, including the Blue Gene/P system. The MASS libraries provide improved performance over the standard mathematical library routines, are thread-safe, and support compilations in C, C++, and Fortran applications. For more information about MASS, refer to the Mathematical Acceleration Subsystem Web page at:


8.9 Engineering and Scientific Subroutine Library libraries

The Engineering and Scientific Subroutine Library (ESSL) for Linux on POWER supports the Blue Gene/P system. ESSL provides over 150 math subroutines that have been specifically tuned for performance on the Blue Gene/P system. For more information about ESSL, refer to the Engineering Scientific Subroutine Library and Parallel ESSL Web page at:

http://www-03.ibm.com/systems/p/software/essl.html

**Important:** When using IBM XL Fortran V11.1 for IBM System Blue Gene, customers must use ESSL 4.3.1. If an attempt is made to install a wrong mix of ESSL and XLF, the rpm installation fails with a dependency error message.

8.10 Python support

Python is a dynamic object-oriented programming language, and now it is available on Blue Gene/P. Python V2.5 has been integrated into the tool chain, and it is part of the driver. Example 8-5 illustrates how to invoke a simple Python program. In this example, the -cwd parameter identifies the directory containing the Python source. For Python applications that require user-supplied shared libraries, refer to section 8.6, “Creation of libraries on Blue Gene/P” on page 100 for more information.

**Example 8-5   How to invoke Python on Blue Gene/P**

```bash
mpirun -np 1 -partition R00-M0-N12-J00 -exe
/bgys/drivers/ppcfloor/gnu-linux/bin/python -cwd
/bgusr/mtn/src0403/bgp/gnu/Python-2.5/Lib/test -verbose 1 -args "test_array.py"
```
Additional information about Python can be found at:

- Python official site at
  http://www.python.org/
- Python Tutorial at
  http://docs.python.org/tut/tut.html
- Python Library Reference at
  http://docs.python.org/lib/lib.html
- pyMPI at
  http://pympi.sourceforge.net/

8.11 Tuning your code for Blue Gene/P

In the sections that follow, we describe strategies that you can use to best exploit the SIMD capabilities of the Blue Gene/P 450 processor and the XL compilers’ advanced instruction scheduling.

8.11.1 Using the compiler optimization options

The -O3 compiler option provides a high level of optimization and automatically sets other options that are especially useful on the Blue Gene/P system. The -qhot=simd option enables SIMD vectorization of loops. It is enabled by default if you use -O4, -O5, or -qhot.

For more information about optimization options, see the following references:

- “Optimizing your applications” in the XL C/C++ Programming Guide, under Product Documentation on the following Web page

- “Optimizing XL Fortran programs” in the XL Fortran User Guide, under Product Documentation on the following Web page
8.11.2 Parallel operations on the PowerPC 450

Similar to the Blue Gene/L system, floating-point instructions can operate simultaneously on the primary and secondary registers. Figure 8-1 illustrates these registers.

The registers allow the PowerPC 450 processor to operate certain identical operations in parallel. Load/store instructions can also be issued with a single instruction. For more detailed information, see the white paper “Exploiting the Dual Floating Point Units in Blue Gene/L” on the Web at:

http://www-01.ibm.com/support/docview.wss?uid=swg27007511

The IBM XL compilers leverage this functionality under the following conditions:

- Parallel instructions are issued for load/store instructions if the alignment and size are aligned with natural alignment. This is 16 bytes for a pair of doubles, but only 8 bytes for a pair of floats.
- The compiler can issue parallel instructions when the application vectors have stride-one memory accesses. However, the compiler via IPA issues parallel instructions with non-stride-one data in certain loops, if it can be shown to improve performance.
- `-qhot=simd` is the default with `-qarch=450d`.
- `-O4` provides analysis at compile time with limited scope analysis and issuing parallel instructions (SIMD).
- `-O5` provides analysis for the entire program at link time to propagate alignment information. You must compile and link with `-O5` to obtain the full benefit.
8.11.3 Using Single Instruction Multiple Data instructions in applications

On the Blue Gene/P system, normal PowerPC assembler instructions use the primary floating-point pipe. To enable instructions in parallel, special assembly instructions must be generated using the following compiler options:

-qarch=450d  This flag in the compiler enables parallel instructions to use the primary and secondary registers (SIMD instructions). See Figure 8-1 on page 105.

-qtune=450  This flag optimizes code for the IBM 450 microprocessors, as previously mentioned.

-O2 and up  This option in the compiler enables parallel instructions.

The XL compiler optimizer consists of two major parts:

- Toronto Portable Optimizer (TPO) for high-level inter-procedural optimization
- Toronto Optimizing Back End with Yorktown (TOBEY) for low-level back-end optimization

SIMD instructions occur in both optimizers. SIMD instruction generation in TOBEY is activated by default for -O2 and up. SIMD generation in TPO is added when using -qhot, -O4, or -O5. Specifically, the -qhot option adds SIMD generation, but options -O4 and -O5 automatically call -qhot. For more details, see the C, C++, and Fortran manuals on the Web at the following addresses:

- XL C/C++

- XL Fortran

For some applications, the compiler generates more efficient code without the TPO SIMD level. If you have statically allocated array, and a loop in the same routine, call TOBEY with -qhot or -O4. Nevertheless, on top of SIMD generation from TOBEY, with -qhot, optimizations are enabled that can alter the semantic of the code and on rare occasions can generate less efficient code. Also, with -qhot=nosimd, you can suppress some of these optimizations.

To use the SIMD capabilities of the XL compilers:

1. Start to compile:
   -03 -qarch=450d -qtune=450
   We recommend that you use -qarch=450d -qtune=450, in this order. The compiler only generates SIMD instructions from -O2 and up.

2. Increase the optimization level, and call the high-level inter-procedural optimizer:
   - -O5 (link time, whole-program analysis, and SIMD instruction)
   - -O4 (compile time, limited scope analysis, and SIMD instructions)
   - -O3 -qhot=simd (compile time, less optimization, and SIMD instructions)

3. Tune your program:
   a. Check the SIMD instruction generation in the object code listing (-qsource -qlist).
   b. Use compiler feedback (-qreport -qhot) to guide you.
c. Help the compiler with extra information (directives and pragmas).
   - Enter the alignment information with directives and pragmas. In C, enter:
     
     __alignx
     
     In Fortran, enter:
     
     ALIGNX
   - Tell the compiler that data accessed through pointers is disjoint. In C, enter:
     
     #pragma disjoint
   - Use constant loop bound, #define, when possible.
   - Use data flow instead of control flow.
   - Use select instead of if/then/else. Use macros instead of calls.
   - Tell the compiler not to generate SIMD instructions if it is not profitable (trip count low). In C, enter:
     
     #pragma nosimd
     
     In Fortran, enter the following line just before the loop:
     
     !IBM* NOSIMD

   - Many applications can require modifying algorithms. The previous bullet, which explains how not to generate SIMD instructions, gives constructs that might help to modify the code. Here are hints to use when modifying your code:
     - Loops must be stride one accesses.
     - For function calls in loop:
       - Try to inline the calls.
       - Loop with if statement.
       - Use pointer and aliasing.
       - Use integer operations.
     - Assumed shape arrays in Fortran 90 can hurt enabling SIMD instructions.
   - Generate compiler diagnostics to help you modify and understand how the compiler is optimizing sections of your applications:
     - The -qreport compiler option generates a diagnostic report about SIMD instruction generation. To analyze the generated code and the use of quadword loads and stores, you must look at the pseudo assembler code within the object file listing. The diagnostic report provides two types of information about SIMD generation:
       - Information on success
         
         (simdizable) [feature][version]
         
         [feature] further characterizes the simdizable loop:
         
         misalign (compile time store)
         Refers to a simdizable loop with misaligned accesses.
         
         shift (4 compile time)
         Refers to a simdizable loop with 4 stream shift inserted. shift refers to the number of misaligned data references that were found. It has a performance impact because these loops must be loaded across, and then an extra select instruction must be inserted.
priv Indicates that the compiler has generated a private variable. priv means a private variable was found. In general, it should have no performance impact, but in practice it sometimes does.

reduct Indicates that a simdizable loop has a reduction construct. reduct means that a reduction was found. It is simdized using partial sums, which must be added at the end of the loop.

[version] further characterizes if and why versioned loops were created:

relative align Indicates the version for relative alignment. The compiler has generated a test and two versions.

trip count Versioned for a short run-time trip count.

- Information on failure
  - In case of misalignment: misalign(...)  
    * Non-natural: non-naturally aligned accesses 
    * Run time: run-time alignment
  - About the structure of the loop
    * Irregular loop structure (while-loop). 
    * Contains control flow: if/then/else. 
    * Contains function call: Function call bans SIMD instructions. 
    * Trip count too small. 
  - About dependences: dependence due to aliasing
  - About array references 
    * Access not stride one 
    * Memory accesses with unsupported alignment 
    * Contains run-time shift 
  - About pointer references: Non-normalized pointer accesses

8.12 Tips for optimizing applications

The following sections are an excerpt from IBM System Blue Gene Solution: Application Development, SG24-7179 but tailored to the Blue Gene/P system because they apply here as well. They provide useful tips on how to optimize certain constructs in your code.
8.13 Identifying performance bottlenecks

The first step in applications tuning requires identifying where the bottlenecks are located in the entire application. If multiple locations are identified as potential bottlenecks, prioritization is required. Figure 8-2 illustrates the initial set of steps required to identify where the bottleneck might be located.

Figure 8-2 Steps to identify performance bottlenecks

Proper I/O utilization tends to be a problem, and in many applications I/O optimization is required. Identify if that is the case for your application. The IBM toolkit for Blue Gene/P provides the Modular I/O (MIO) library that can be used for an applications-level I/O performance improvement (See IBM System Blue Gene Solution: High Performance Computing Toolkit for Blue Gene/P, REDP-4256). Memory utilization on Blue Gene/P involves optimal utilization of the memory hierarchy. This needs to be coordinated with the double float-unit to leverage the execution of instructions in parallel. As part of the IBM toolkit Xprofiler helps analyze applications by collecting data using the -pg compiler option to identify functions that are most CPU intensive. gmon profiler also provides similar information. Appendix G, “Use of GNU profiling tool on Blue Gene/P” on page 309 provides additional information about gmon. The IBM toolkit provides the MPI profiler and a tracing library for MPI programs.
8.13.1 Structuring data in adjacent pairs

The Blue Gene/P 450d processor's dual FPU includes special instructions for parallel computations. The compiler tries to pair adjacent single-precision or double-precision floating-point values to operate on them in parallel. Therefore, you can accelerate computations by defining data objects that occupy adjacent memory blocks and are naturally aligned. These include arrays or structures of floating-point values and complex data types.

Whether you use an array, a structure, or a complex scalar, the compiler searches for sequential pairs of data for which it can generate parallel instructions. For example, using the C code in Example 8-6, each pair of elements in a structure can be operated on in parallel.

Example 8-6  Adjacent paired data

```c
struct quad {
    double a, b, c, d;
};
struct quad x, y, z;
void foo()
{
    z.a = x.a + y.a;
    z.b = x.b + y.b; /* can load parallel (x.a,x.b), and (y.a, y.b), do parallel add, and
    store parallel (z.a, z.b) */
    z.c = x.c + y.c;
    z.d = x.d + y.d; /* can load parallel (x.c,x.d), and (y.c, y.d), do parallel add, and
    store parallel (z.c, z.d) */
}
```

The advantage of using complex types in arithmetic operations is that the compiler automatically uses parallel add, subtract, and multiply instructions when complex types appear as operands to addition, subtraction, and multiplication operators. Furthermore, the data that you provide does not need to represent complex numbers. In fact, both elements are represented internally as two real values. See 8.13.8, “Complex type manipulation functions” on page 119, for a description of the set of built-in functions that are available for the Blue Gene/P system. These functions are especially designed to efficiently manipulate complex-type data and include a function to convert non-complex data to complex types.

8.13.2 Using vectorizable basic blocks

The compiler schedules instructions most efficiently within extended basic blocks. Extended basic blocks are code sequences that can contain conditional branches but have no entry points other than the first instruction. Specifically, minimize the use of branching instructions for:

- Handling special cases, such as the generation of not-a-number (NaN) values.
- C/C++ error handling that sets a value for `errno`.

  To explicitly inform the compiler that none of your code will set `errno`, you can compile with the `-qignerrno` compiler option (automatically set with `-O3`).

- C++ exception handlers.

  To explicitly inform the compiler that none of your code will throw any exceptions, and therefore, that no exception-handling code must be generated, you can compile with the `-qnoeh` compiler option.
In addition, the optimal basic blocks remove dependencies between computations, so that the compiler views each statement as entirely independent. You can construct a basic block as a series of independent statements or as a loop that repeatedly computes the same basic block with different arguments.

If you specify the -qhot=simd compilation option, along with a minimum optimization level of -O2, the compiler can then vectorize these loops by applying various transformations, such as unrolling and software pipelining. See 8.13.4, “Removing possibilities for aliasing (C/C++)” on page 112, for additional strategies for removing data dependencies.

8.13.3 Using inline functions

An inline function is expanded in any context in which it is called. This expansion avoids the normal performance overhead associated with the branching for a function call, and it allows functions to be included in basic blocks. The XL C/C++ and Fortran compilers provide several options for inlining.

The following options instruct the compiler to automatically inline all functions it deems appropriate:

- **XL C/C++**
  - -0 through -05
  - -qipa
- **XL Fortran**
  - -O4 or -O5
  - -qipa

With the following options, you can select or name functions to be inlined:

- **XL C/C++**
  - -qinline
  - -Q
- **XL Fortran**
  - -Q

In C/C++, you can also use the standard `inline` function specifier or the `__attribute__((always_inline))` extension in your code to mark a function for inlining.

**Usage of inlining:** Do not overuse inlining because of the limits on how much inlining can be done. Mark the most important functions.

For more information about the various compiler options for controlling function inlining, see the following publications:

- “XL C/C++ Compiler Reference”
- “XL Fortran User Guide”

Also available from this Web address, the “XL C/C++ Language Reference” provides information about the different variations of the `inline` keyword supported by XL C and C++, as well as the inlining function attribute extensions.
8.13.4 Removing possibilities for aliasing (C/C++)

When you use pointers to access array data in C/C++, the compiler cannot assume that the memory accessed by pointers will not be altered by other pointers that refer to the same address. For example, if two pointer input parameters share memory, the instruction to store the second parameter can overwrite the memory read from the first load instruction. This means that, after a store for a pointer variable, any load from a pointer must be reloaded. Consider the code in Example 8-7.

Example 8-7  Sample code

```c
int i = *p;
*q = 0;
j = *p;
```

If *q aliases *p, then the value must be reloaded from memory. If *q does not alias *p, the old value that is already loaded into i can be used.

To avoid the overhead of reloading values from memory every time they are referenced in the code, and to allow the compiler to simply manipulate values that are already resident in registers, you can use several strategies. One approach is to assign input array element values to local variables and perform computations only on the local variables, as shown in Example 8-8.

Example 8-8  Array parameters assigned to local variables

```c
#include <math.h>
void reciprocal_roots (const double* x, double* f)
{
    double x0 = x[0] ;
    double x1 = x[1] ;
    double r0 = 1.0/sqrt(x0) ;
    double r1 = 1.0/sqrt(x1) ;
    f[0] = r0 ;
    f[1] = r1 ;
}
```

If you are certain that two references do not share the same memory address, another approach is to use the #pragma disjoint directive. This directive asserts that two identifiers do not share the same storage, within the scope of their use. Specifically, you can use pragma to inform the compiler that two pointer variables do not point to the same memory address. The directive in Example 8-9 indicates to the compiler that the pointers-to-arrays of double x and f do not share memory.

Example 8-9  The #pragma disjoint directive

```c
__inline void ten_reciprocal_roots (double* x, double* f)
{
    #pragma disjoint (*x, *f)
    int i;
    for (i=0; i < 10; i++)
        f[i]= 1.0 / sqrt (x[i]);
}
```

Important: The correct functioning of this directive requires that the two pointers be disjoint. If they are not, the compiled program cannot run correctly.
8.13.5 Structure computations in batches

Floating-point operations are pipelined in the 450 processor, so that one floating-point calculation is performed per cycle, with a latency of approximately five cycles. Therefore, to keep the 450 processor's floating-point units busy, organize floating-point computations to perform step-wise operations in batches - for example, arrays of five elements and loops of five iterations. For the 450d, which has two FPUs, use batches of ten.

For example, with the 450d, at high optimization, the function in Example 8-10 should perform ten parallel reciprocal roots in about five cycles more than a single reciprocal root. This is because the compiler performs two reciprocal roots in parallel and then uses the “empty” cycles to run four more parallel reciprocal roots.

Example 8-10 Function to calculate reciprocal roots for arrays of 10 elements

```c
__inline void ten_reciprocal_roots (double* x, double* f)
{
  #pragma disjoint (*x, *f)

  int i;
  for (i=0; i < 10; i++)
    f[i] = 1.0 / sqrt (x[i]);
}
```

The definition in Example 8-11 shows “wrapping” the inlined, optimized `ten_reciprocal_roots` function, in Example 8-10, inside a function that allows you to pass in arrays of any number of elements. This function then passes the values in batches of ten to the `ten_reciprocal_roots` function and calculates the remaining operations individually.

Example 8-11 Function to pass values in batches of ten

```c
static void unaligned_reciprocal_roots (double* x, double* f, int n)
{
  #pragma disjoint (*x, *f)

  while (n >= 10) {
    ten_reciprocal_roots (x, f);
    x += 10;
    f += 10;
  }
  /* remainder */
  while (n > 0) {
    *f = 1.0 / sqrt (*x);
    f++, x++;
  }
}
```

8.13.6 Checking for data alignment

Floating-point data alignment requirements are based on the size of the data: 4-byte data must be word aligned, 8-byte data must be double-word aligned, and 16-byte data must be quad-word aligned. If data is accessed or modified which is not properly aligned, the hardware generates an alignment exception. The user can determine how alignment exceptions should be handled by the setting of the environment variable BG_MAXALIGNEXP. If this variable is not set, the kernel will handle up to 1000 alignment exceptions; after this a SIGBUS signal is raised, the program ends and generates a core file. The core file provides information about the instruction address and stack trace where the alignment exception occurred. Setting BG_MAXALIGNEXP=-1 indicates that all alignment exceptions should be
handled. Setting BG_MAXALIGNEXP=0 indicates that no alignment exceptions should be handled. Because alignment exceptions can cause a severe performance penalty, this can be used to find code that is taking alignment exceptions unexpectedly.

The compiler does not generate these parallel load and store instructions unless it is sure that is safe to do so. For non-pointer local and global variables, the compiler knows when this is safe. To allow the compiler to generate these parallel loads and stores for accesses through pointers, include code that tests for correct alignment and that gives the compiler hints.

To test for alignment, first create one version of a function which asserts the alignment of an input variable at that point in the program flow. You can use the C/C++ __alignx built-in function or the Fortran ALIGNX function to inform the compiler that the incoming data is correctly aligned according to a specific byte boundary, so it can efficiently generate loads and stores.

The function takes two arguments. The first argument is an integer constant expressing the number of alignment bytes (must be a positive power of two). The second argument is the variable name, typically a pointer to a memory address.

Example 8-12 shows the C/C++ prototype for the function.

**Example 8-12  C/C++ prototype**

```c
extern
#ifdef __cplusplus
"builtin"
#endif
void __alignx (int n, const void *addr)
```

Here \( n \) is the number of bytes. For example, \( __alignx(16, y) \) specifies that the address \( y \) is 16-byte aligned.

In Fortran95, the built-in subroutine is ALIGNX(K,M), where K is of type INTEGER(4), and M is a variable of any type. When M is an integer pointer, the argument refers to the address of the pointee.

Example 8-13 asserts that the variables \( x \) and \( f \) are aligned along 16-byte boundaries.

**Example 8-13   Using the __alignx built-in function**

```c
#include <math.h>
#include <builtins.h>
__inline void aligned_ten_reciprocal_roots (double* x, double* f)
{
  #pragma disjoint (*x, *f)
  int i;
  __alignx (16, x);
  __alignx (16, f);
  for (i=0; i < 10; i++)
    f[i]= 1.0 / sqrt (x[i]);
}
```

The __alignx function: The __alignx function does not perform any alignment. It merely informs the compiler that the variables are aligned as specified. If the variables are not aligned correctly, the program does not run properly.
After you create a function to handle input variables that are correctly aligned, you can then create a function that tests for alignment and then calls the appropriate function to perform the calculations. The function in Example 8-14 checks to see whether the incoming values are correctly aligned. Then it calls the “aligned” (Example 8-13 on page 114) or “unaligned” (Example 8-10 on page 113) version of the function according to the result.

Example 8-14  Function to test for alignment

```c
void reciprocal_roots (double *x, double *f, int n)
{
    /* are both x & f 16 byte aligned? */
    if ( ((((int) x) | ((int) f)) & 0xf) == 0) /* This could also be done as:
        if (((int) x % 16 == 0) && ((int) f % 16) == 0) */
        aligned_ten_reciprocal_roots (x, f, n);
    else
        ten_reciprocal_roots (x, f, n);
}
```

The alignment test in Example 8-14 provides an optimized method of testing for 16-byte alignment by performing a bit-wise OR on the two incoming addresses and testing whether the lowest four bits are 0 (that is, 16-byte aligned).

### 8.13.7 Using XL built-in floating-point functions for Blue Gene/P

The XL C/C++ and Fortran95 compilers include a large set of built-in functions that are optimized for the PowerPC architecture. For a full description of them, refer to the following documents:

- Appendix B: “Built-In Functions” in *XL C/C++ Compiler Reference*
  

- “Intrinsic Procedures” in *XL Fortran Language Reference*
  

In addition, on the Blue Gene/P system, the XL compilers provide a set of built-in functions that are specifically optimized for the PowerPC 450d dual FPU. These built-in functions provide an almost one-to-one correspondence with the dual floating-point instruction set.

All of the C/C++ and Fortran built-in functions operate on complex data types, which have an underlying representation of a two-element array, in which the real part represents the primary element and the imaginary part represents the second element. The input data you provide does not need to represent complex numbers. In fact, both elements are represented internally as two real values. None of the built-in functions performs complex arithmetic. A set of built-in functions designed to efficiently manipulate complex-type variables is also available.
The Blue Gene/P built-in functions perform several types of operations as explained in the following paragraphs.

**Parallel operations** perform SIMD computations on the primary and secondary elements of one or more input operands. They store the results in the corresponding elements of the output. As an example, Figure 8-3 illustrates how a parallel-multiply operation is performed.

![Parallel operations](image)

**Cross operations** perform SIMD computations on the opposite primary and secondary elements of one or more input operands. They store the results in the corresponding elements in the output. As an example, Figure 8-4 illustrates how a cross-multiply operation is performed.

![Cross-multiply operations](image)

**Copy-primary operations** perform SIMD computation between the corresponding primary and secondary elements of two input operands, where the primary element of the first...
operand is replicated to the secondary element. As an example, Figure 8-5 illustrates how a cross-primary-multiply operation is performed.

**Figure 8-5  Copy-primary multiply operations**

*Copy-secondary operations* perform SIMD computation between the corresponding primary and secondary elements of two input operands, where the secondary element of the first operand is replicated to the primary element. As an example, Figure 8-6 illustrates how a cross-secondary multiply operation is performed.

**Figure 8-6  Copy-secondary multiply operations**
In *cross-copy operations*, the compiler crosses either the primary or secondary element of the first operand, so that copy-primary and copy-secondary operations can be used interchangeably to achieve the same result. The operation is performed on the total value of the first operand. As an example, Figure 8-7 illustrates the result of a cross-copy multiply operation.

```
Input operand a
Primary element + Secondary element

Input operand b
Primary element
Secondary element

Output
Primary element
Secondary element
```

*Figure 8-7  Cross-copy multiply operations*

In the following paragraphs, we describe the available built-in functions by category. For each function, the C/C++ prototype is provided. In C, you do not need to include a header file to obtain the prototypes. The compiler includes them automatically. In C++, you must include the header file builtins.h.

Fortran does not use prototypes for built-in functions. Therefore, the interfaces for the Fortran95 functions are provided in textual form. The function names omit the double underscore (__) in Fortran95.

All of the built-in functions, with the exception of the complex type manipulation functions, require compilation under -qarch=450d. This is the default setting on the Blue Gene/P system.

To help clarify the English description of each function, the following notation is used:

```
element(variable)
```

Here *element* represents one of *primary* or *secondary*, and *variable* represents input variable *a*, *b*, or *c*, and the output variable *result*. For example, consider the following formula:

```
primary(result) = primary(a) + primary(b)
```

This formula indicates that the primary element of input variable *a* is added to the primary element of input variable *b* and stored in the primary element of the result.

To optimize your calls to the Blue Gene/P built-in functions, follow the guidelines provided in 8.11, "Tuning your code for Blue Gene/P" on page 104. Using the alignx built-in function (described in 8.13.6, "Checking for data alignment" on page 113), and specifying the disjoint pragma (described in 8.13.4, "Removing possibilities for aliasing (C/C++)", on page 112), are recommended for code that calls any of the built-in functions.
8.13.8 Complex type manipulation functions

Complex type manipulation functions, listed in Table 8-3, are useful for efficiently manipulating complex data types. Using these functions, you can automatically convert real floating-point data to complex types and extract the real (primary) and imaginary (secondary) parts of complex values.

Table 8-3 Complex type manipulation functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Convert dual reals to complex (single-precision): __cmplxf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Converts two single-precision real values to a single complex value. The real ( a ) is converted to the primary element of the return value, and the real ( b ) is converted to the secondary element of the return value.</td>
</tr>
<tr>
<td>Formula</td>
<td>primary(result) =a secondary(result) = b</td>
</tr>
<tr>
<td>C/C++ prototype</td>
<td>float _Complex __cmplxf (float a, float b);</td>
</tr>
<tr>
<td>Fortran descriptions</td>
<td>CMPLXF(A,B) where A is of type REAL(4) where B is of type REAL(4) result is of type COMPLEX(4)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Convert dual reals to complex (double-precision): __cmplx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Converts two double-precision real values to a single complex value. The real ( a ) is converted to the primary element of the return value, and the real ( b ) is converted to the secondary element of the return value.</td>
</tr>
<tr>
<td>Formula</td>
<td>primary(result) =a secondary(result) = b</td>
</tr>
<tr>
<td>C/C++ prototype</td>
<td>double _Complex __cmplx (double a, double b); long double _Complex __cmplxl (long double a, long double b);^a</td>
</tr>
<tr>
<td>Fortran descriptions</td>
<td>CMPLX(A,B) where A is of type REAL(8) where B is of type REAL(8) result is of type COMPLEX(8)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Extract real part of complex (single-precision): __crealf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Extracts the primary part of a single-precision complex value ( a ), and returns the result as a single real value.</td>
</tr>
<tr>
<td>Formula</td>
<td>result =primary(a)</td>
</tr>
<tr>
<td>C/C++ prototype</td>
<td>float __crealf (float _Complex a);</td>
</tr>
<tr>
<td>Fortran descriptions</td>
<td>CREALF(A) where A is of type COMPLEX(4) result is of type REAL(4)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Extract real part of complex (double-precision): __creal, __creall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Extracts the primary part of a double-precision complex value ( a ), and returns the result as a single real value.</td>
</tr>
<tr>
<td>Formula</td>
<td>result =primary(a)</td>
</tr>
</tbody>
</table>
### 8.13.9 Load and store functions

Table 8-4 lists and explains the various parallel load and store functions that are available.

<table>
<thead>
<tr>
<th>Function</th>
<th>Parallel load (single-precision): __lfps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Loads parallel single-precision values from the address of <em>a</em>, and converts the results to double-precision. The first word in <code>address(a)</code> is loaded into the primary element of the return value. The next word, at location <code>address(a)+4</code>, is loaded into the secondary element of the return value.</td>
</tr>
</tbody>
</table>
| Formula   | \[
|           | primary(result) = a[0] \\
|           | secondary(result) = a[1] \\
| C/C++ prototype | double _Complex __lfps (float * a); |

#### Table 8-4  Load and store functions

<table>
<thead>
<tr>
<th>C/C++ prototype</th>
<th>C/C++ prototype</th>
</tr>
</thead>
<tbody>
<tr>
<td>double __creal (double _Complex a);</td>
<td></td>
</tr>
<tr>
<td>long double __creall (long double _Complex a);</td>
<td></td>
</tr>
<tr>
<td>Fortran descriptions</td>
<td>Fortran descriptions</td>
</tr>
<tr>
<td>CREAL(A) where A is of type COMPLEX(8) result is of type REAL(8)</td>
<td></td>
</tr>
<tr>
<td>CREALL(A) where A is of type COMPLEX(16) result is of type REAL(16)</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Extract imaginary part of complex (single-precision): __cimagf</td>
</tr>
<tr>
<td>Purpose</td>
<td>Extracts the secondary part of a single-precision complex value <em>a</em>, and returns the result as a single real value.</td>
</tr>
<tr>
<td>Formula</td>
<td>result = secondary(a)</td>
</tr>
<tr>
<td>C/C++ prototype</td>
<td>float __cimagf (float _Complex a);</td>
</tr>
<tr>
<td>Fortran descriptions</td>
<td>Fortran descriptions</td>
</tr>
<tr>
<td>CIMAGF(A) where A is of type COMPLEX(4) result is of type REAL(4)</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Extract imaginary part of complex (double-precision): __cimag, __cimagl</td>
</tr>
<tr>
<td>Purpose</td>
<td>Extracts the imaginary part of a double-precision complex value <em>a</em>, and returns the result as a single real value.</td>
</tr>
<tr>
<td>Formula</td>
<td>result = secondary(a)</td>
</tr>
<tr>
<td>C/C++ prototype</td>
<td>double __cimag (double _Complex a); long double __cimagl (long double _Complex a);</td>
</tr>
<tr>
<td>Fortran descriptions</td>
<td>Fortran descriptions</td>
</tr>
<tr>
<td>CIMAG(A) where A is of type COMPLEX(8) result is of type REAL(8)</td>
<td></td>
</tr>
<tr>
<td>CIMAGL(A) where A is of type COMPLEX(16) result is of type REAL(16)</td>
<td></td>
</tr>
</tbody>
</table>

---

*a. 128-bit C/C++ long double types are not supported on Blue Gene/L. Long doubles are treated as regular double-precision longs.*
| Fortran description | LOADFP(A)  
where A is of type REAL(4)  
result is of type COMPLEX(8) |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function</strong></td>
<td>Cross load (single-precision): __lfxs</td>
</tr>
<tr>
<td><strong>Purpose</strong></td>
<td>Loads single-precision values that have been converted to double-precision, from the address of a. The first word in address(a) is loaded into the secondary element of the return value. The next word, at location address(a)+4, is loaded into the primary element of the return value.</td>
</tr>
</tbody>
</table>
| **Formula**       | primary(result) = a[1]  
secondary(result) = a[0] |
| **C/C++ prototype** | double _Complex __lfxs (float * a); |
| Fortran description | LOADFX(A)  
where A is of type REAL(4)  
result is of type COMPLEX(8) |
| **Function**      | Parallel load: __lfpd |
| **Purpose**       | Loads parallel values from the address of a. The first word in address(a) is loaded into the primary element of the return value. The next word, at location address(a)+8, is loaded into the secondary element of the return value. |
| **Formula**       | primary(result) = a[0]  
secondary(result) = a[1] |
| **C/C++ prototype** | double _Complex __lfpd(double* a); |
| Fortran description | LOADFP(A)  
where A is of type REAL(8)  
result is of type COMPLEX(8) |
| **Function**      | Cross load: __lfxd |
| **Purpose**       | Loads values from the address of a. The first word in address(a) is loaded into the secondary element of the return value. The next word, at location address(a)+8, is loaded into the primary element of the return value. |
| **Formula**       | primary(result) = a[1]  
secondary(result) = a[0] |
| **C/C++ prototype** | double _Complex __lfxd (double * a); |
| Fortran description | LOADFX(A)  
where A is of type REAL(8)  
result is of type COMPLEX(8) |
| **Function**      | Parallel store (single-precision): __stfps |
| **Purpose**       | Stores in parallel double-precision values that have been converted to single-precision, into address(b). The primary element of a is converted to single-precision and stored as the first word in address(b). The secondary element of a is converted to single-precision and stored as the next word at location address(b)+4. |
| **Formula**       | b[0] = primary(a)  
b[1] = secondary(a) |
<table>
<thead>
<tr>
<th><strong>C/C++ prototype</strong></th>
<th><strong>void __stfps (float * b, double _Complex a);</strong></th>
</tr>
</thead>
</table>
| **Fortran description** | **STOREFP(B, A)**  
where B is of type REAL(4)  
A is of type COMPLEX(8)  
result is none |
| **Function** | **Cross store (single-precision): __stfxs** |
| **Purpose** | Stores double-precision values that have been converted to single-precision, into address(b). The secondary element of a is converted to single-precision and stored as the first word in address(b). The primary element of a is converted to single-precision and stored as the next word at location address(b)+4. |
| **Formula** | b[0] = secondary(a)  
b[1] = primary(a) |
| **C/C++ prototype** | **void __stfxs (float * b, double _Complex a);** |
| **Fortran description** | **STOREFX(B, A)**  
where B is of type REAL(4)  
A is of type COMPLEX(8)  
result is none |
| **Function** | **Parallel store: __stfpd** |
| **Purpose** | Stores in parallel values into address(b). The primary element of a is stored as the first double word in address(b). The secondary element of a is stored as the next double word at location address(b)+8. |
| **Formula** | b[0] = primary(a)  
b[1] = secondary(a) |
| **C/C++ prototype** | **void __stfpd (double * b, double _Complex a);** |
| **Fortran description** | **STOREFP(B, A)**  
where B is of type REAL(8)  
A is of type COMPLEX(8)  
result is none |
| **Function** | **Cross store: __stfxd** |
| **Purpose** | Stores values into address(b). The secondary element of a is stored as the first double word in address(b). The primary element of a is stored as the next double word at location address(b)+8. |
| **Formula** | b[0] = secondary(a)  
b[1] = primary(a) |
| **C/C++ prototype** | **void __stfxd (double * b, double _Complex a);** |
| **Fortran description** | **STOREFP(B, A)**  
where B is of type REAL(8)  
A is of type COMPLEX(8)  
result is none |
| **Function** | **Parallel store as integer: __stfpiw** |
### Chapter 8. Developing applications with IBM XL compilers

#### 8.13.10 Move functions

Table 8-5 lists and explains the parallel move functions that are available.

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Stores in parallel floating-point double-precision values into <code>b</code> as integer words. The lower-order 32 bits of the primary element of <code>a</code> are stored as the first integer word in <code>address(b)</code>. The lower-order 32 bits of the secondary element of <code>a</code> are stored as the next integer word at location <code>address(b)+4</code>. This function is typically preceded by a call to the <code>__fpctiw</code> or <code>__fpctiwz</code> built-in functions, described in “Unary functions” on page 123, which perform parallel conversion of dual floating-point values to integers.</th>
</tr>
</thead>
</table>
| Formula | `b[0] = primary(a)`  
`b[1] = secondary(a)` |
| C/C++ prototype | `void __stfpiw (int * b, double _Complex a);` |
| Fortran description | `STOREFP(B, A)`  
where `B` is of type INTEGER(4)  
`A` is of type COMPLEX(8)  
result is none |

<table>
<thead>
<tr>
<th>Function</th>
<th>Cross move: <code>__fxmr</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Swaps the values of the primary and secondary elements of operand <code>a</code>.</td>
</tr>
</tbody>
</table>
| Formula | `primary(result) = secondary(a)`  
`secondary(result) = primary(a)` |
| C/C++ prototype | `double _Complex __fxmr (double _Complex a);` |
| Fortran description | `FXMR(A)`  
where `A` is of type COMPLEX(8)  
result is of type COMPLEX(8) |

#### 8.13.11 Arithmetic functions

In the following sections, we describe all the arithmetic built-in functions, categorized by their number of operands.

**Unary functions**

Unary functions operate on a single input operand. These functions are listed in Table 8-6.

<table>
<thead>
<tr>
<th>Function</th>
<th>Parallel convert to integer: <code>__fpctiw</code></th>
</tr>
</thead>
</table>
| Purpose | Converts in parallel the primary and secondary elements of operand `a` to 32-bit integers using the current rounding mode.  
After a call to this function, use the `__stfpiw` function to store the converted integers in parallel, as explained in 8.13.9, “Load and store functions” on page 120. |
| Formula | `primary(result) = primary(a)`  
`secondary(result) = secondary(a)` |
<p>| C/C++ prototype | <code>double _Complex __fpctiw (double _Complex a);</code> |</p>
<table>
<thead>
<tr>
<th>Fortran purpose</th>
<th>Parallel convert to integer and round to zero: __fpctiwz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Converts in parallel the primary and secondary elements of operand a to 32-bit integers and rounds the results to zero. After a call to this function, use the __stfpiw function to store the converted integers in parallel, as explained in 8.13.9, “Load and store functions” on page 120.</td>
</tr>
<tr>
<td>Formula</td>
<td>primary(result) = primary(a) secondary(result) = secondary(a)</td>
</tr>
<tr>
<td>C/C++ prototype</td>
<td>double _Complex __fpctiwz(double _Complex a);</td>
</tr>
<tr>
<td>Fortran description</td>
<td>FPCTIWZ(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fortran purpose</th>
<th>Parallel round double-precision to single-precision: __fprsp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Rounds in parallel the primary and secondary elements of double-precision operand a to single precision.</td>
</tr>
<tr>
<td>Formula</td>
<td>primary(result) = primary(a) secondary(result) = secondary(a)</td>
</tr>
<tr>
<td>C/C++ prototype</td>
<td>double _Complex __fprsp (double _Complex a);</td>
</tr>
<tr>
<td>Fortran description</td>
<td>FPRSP(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fortran purpose</th>
<th>Parallel reciprocal estimate: __fpre</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Calculates in parallel double-precision estimates of the reciprocal of the primary and secondary elements of operand a.</td>
</tr>
<tr>
<td>Formula</td>
<td>primary(result) = primary(a) secondary(result) = secondary(a)</td>
</tr>
<tr>
<td>C/C++ prototype</td>
<td>double _Complex __fpre(double _Complex a);</td>
</tr>
<tr>
<td>Fortran description</td>
<td>FPRE(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fortran purpose</th>
<th>Parallel reciprocal square root: __fprsqrte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Calculates in parallel double-precision estimates of the reciprocals of the square roots of the primary and secondary elements of operand a.</td>
</tr>
<tr>
<td>Formula</td>
<td>primary(result) = primary(a) secondary(result) = secondary(a)</td>
</tr>
<tr>
<td>C/C++ prototype</td>
<td>double _Complex __fprsqrte (double _Complex a);</td>
</tr>
<tr>
<td>Fortran description</td>
<td>FPRSQRTE(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)</td>
</tr>
</tbody>
</table>
### Binary functions

Binary functions operate on two input operands. The functions are listed in Table 8-7.

**Table 8-7  Binary functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Parallel add: __fpadd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Adds in parallel the primary and secondary elements of operands a and b.</td>
</tr>
</tbody>
</table>
| Formula  | primary(result) = primary(a) + primary(b)  
 secondary(result) = secondary(a) + secondary(b) |
| C/C++ prototype | double _Complex __fpadd (double _Complex a, double _Complex b); |
| Fortran description | FPADD(A,B)  
 where A is of type COMPLEX(8)  
 where B is of type COMPLEX(8)  
 result is of type COMPLEX(8) |
<table>
<thead>
<tr>
<th>Function</th>
<th>Parallel subtract: __fpsub</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Subtracts in parallel the primary and secondary elements of operand b from the corresponding primary and secondary elements of operand a.</td>
</tr>
</tbody>
</table>
| Formula  | primary(result) = primary(a) - primary(b)  
secondary(result) = secondary(a) - secondary(b) |
| C/C++ prototype | double _Complex __fpsub (double _Complex a, double _Complex b); |
| Fortran description | FPSUB(A,B)  
where A is of type COMPLEX(8)  
where B is of type COMPLEX(8)  
result is of type COMPLEX(8) |

<table>
<thead>
<tr>
<th>Function</th>
<th>Parallel multiply: __fpmul</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Multiplies in parallel the values of primary and secondary elements of operands a and b.</td>
</tr>
</tbody>
</table>
| Formula  | primary(result) = primary(a) × primary(b)  
secondary(result) = secondary(a) × secondary(b) |
| C/C++ prototype | double _Complex __fpmul (double _Complex a, double _Complex b); |
| Fortran description | FPMUL(A,B)  
where A is of type COMPLEX(8)  
where B is of type COMPLEX(8)  
result is of type COMPLEX(8) |

<table>
<thead>
<tr>
<th>Function</th>
<th>Cross multiply: __fxmul</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>The product of the secondary element of a and the primary element of b is stored as the primary element of the return value. The product of the primary element of a and the secondary element of b is stored as the secondary element of the return value.</td>
</tr>
</tbody>
</table>
| Formula  | primary(result) = secondary(a) x primary(b)  
secondary(result) = primary(a) x secondary(b) |
| C/C++ prototype | double _Complex __fxmul (double _Complex a, double _Complex b); |
| Fortran description | FXMUL(A,B)  
where A is of type COMPLEX(8)  
where B is of type COMPLEX(8)  
result is of type COMPLEX(8) |

<table>
<thead>
<tr>
<th>Function</th>
<th>Cross copy multiply: _fxpmul, __fxsmul</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Both of these functions can be used to achieve the same result. The product of a and the primary element of b is stored as the primary element of the return value. The product of a and the secondary element of b is stored as the secondary element of the return value.</td>
</tr>
</tbody>
</table>
| Formula  | primary(result) = a x primary(b)  
secondary(result) = a x secondary(b) |
Multiply-add functions

Multiply-add functions take three input operands, multiply the first two, and add or subtract the third. Table 8-8 lists these functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Parallel multiply-add: __fpmadd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>The sum of the product of the primary elements of a and b, added to the primary element of c, is stored as the primary element of the return value. The sum of the product of the secondary elements of a and b, added to the secondary element of c, is stored as the secondary element of the return value.</td>
</tr>
<tr>
<td>Formula</td>
<td>primary(result) = primary(a) × primary(b) + primary(c) secondary(result) = secondary(a) × secondary(b) + secondary(c)</td>
</tr>
<tr>
<td>C/C++ prototype</td>
<td>double _Complex __fpmadd (double _Complex c, double _Complex b, double _Complex a);</td>
</tr>
<tr>
<td>Fortran description</td>
<td>FPMADD(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Parallel negative multiply-add: __fpnmadd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>The sum of the product of the primary elements of a and b, added to the primary element of c, is negated and stored as the primary element of the return value. The sum of the product of the secondary elements of a and b, added to the secondary element of c, is negated and stored as the secondary element of the return value.</td>
</tr>
<tr>
<td>Formula</td>
<td>primary(result) = -(primary(a) × primary(b) + primary(c)) secondary(result) = -(secondary(a) × secondary(b) + secondary(c))</td>
</tr>
<tr>
<td>C/C++ prototype</td>
<td>double _Complex __fpnmadd (double _Complex c, double _Complex b, double _Complex a);</td>
</tr>
<tr>
<td>Fortran description</td>
<td>FPNMADD(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Parallel multiply-subtract: __fpmsub</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>The difference of the primary element of c, subtracted from the product of the primary elements of a and b, is stored as the primary element of the return value. The difference of the secondary element of c, subtracted from the product of the secondary elements of a and b, is stored as the secondary element of the return value.</td>
</tr>
<tr>
<td>Formula</td>
<td>primary(result) = primary(a) × primary(b) - primary(c) secondary(result) = secondary(a) × secondary(b) - secondary(c)</td>
</tr>
</tbody>
</table>
### Parallel negative multiply-subtract: __fpnmsub

**Purpose**
The difference of the primary element of `c`, subtracted from the product of the primary elements of `a` and `b`, is negated and stored as the primary element of the return value. The difference of the secondary element of `c`, subtracted from the product of the secondary elements of `a` and `b`, is negated and stored as the secondary element of the return value.

**Formula**
```
primary(result) = -(primary(a) × primary(b) - primary(c))
secondary(result) = -(secondary(a) × secondary(b) - secondary(c))
```

### Cross multiply-add: __fxmadd

**Purpose**
The sum of the product of the primary element of `a` and the secondary element of `b`, added to the primary element of `c`, is stored as the primary element of the return value. The sum of the product of the secondary element of `a` and the primary element of `b`, added to the secondary element of `c`, is stored as the secondary element of the return value.

**Formula**
```
primary(result) = primary(a) × secondary(b) + primary(c)
secondary(result) = secondary(a) × primary(b) + secondary(c)
```

### Cross negative multiply-add: __fxnmadd

**Purpose**
The sum of the product of the primary element of `a` and the secondary element of `b`, added to the primary element of `c`, is negated and stored as the primary element of the return value. The sum of the product of the secondary element of `a` and the primary element of `b`, added to the secondary element of `c`, is negated and stored as the secondary element of the return value.

**Formula**
```
primary(result) = -(primary(a) × secondary(b) + primary(c))
secondary(result) = -(secondary(a) × primary(b) + secondary(c))
```
| Fortran description | FXNMADD(C,B,A)  
where C is of type COMPLEX(8)  
where B is of type COMPLEX(8)  
where A is of type COMPLEX(8)  
result is of type COMPLEX(8) |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function</strong></td>
<td><strong>Cross multiply-subtract: __fxmsub</strong></td>
</tr>
<tr>
<td><strong>Purpose</strong></td>
<td>The difference of the primary element of c, subtracted from the product of the primary element of a and the secondary element of b, is stored as the primary element of the return secondary element of a, and the primary element of b is stored as the secondary element of the return value.</td>
</tr>
</tbody>
</table>
| **Formula**         | primary(result) = primary(a) × secondary(b) - primary(c)  
secondary(result) = secondary(a) × primary(b) - secondary(c) |
| **C/C++ prototype** | double _Complex __fxmsub (double _Complex c, double _Complex b, double _Complex a); |

| Fortran description | FXMSUB(C,B,A)  
where C is of type COMPLEX(8)  
where B is of type COMPLEX(8)  
where A is of type COMPLEX(8)  
result is of type COMPLEX(8) |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function</strong></td>
<td><strong>Cross negative multiply-subtract: __fxnmsub</strong></td>
</tr>
<tr>
<td><strong>Purpose</strong></td>
<td>The difference of the primary element of c, subtracted from the product of the primary element of a and the secondary element of b, is negated and stored as the primary element of the return value. The difference of the secondary element of c, subtracted from the product of the secondary element of a and the primary element of b, is negated and stored as the secondary element of the return value.</td>
</tr>
</tbody>
</table>
| **Formula**         | primary(result) = -(primary(a) × secondary(b) - primary(c))  
secondary(result) = -(secondary(a) × primary(b) - secondary(c)) |
| **C/C++ prototype** | double _Complex __fxnmsub (double _Complex c, double _Complex b, double _Complex a); |

| Fortran description | FXNMSUB(C,B,A)  
where C is of type COMPLEX(8)  
where B is of type COMPLEX(8)  
where A is of type COMPLEX(8)  
result is of type COMPLEX(8) |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function</strong></td>
<td><strong>Cross copy multiply-add: __fxcpmadd, __fxcsmadd</strong></td>
</tr>
<tr>
<td><strong>Purpose</strong></td>
<td>Both of these functions can be used to achieve the same result. The sum of the product of a and the primary element of b, added to the primary element of c, is stored as the primary element of the return value. The sum of the product of a and the secondary element of b, added to the secondary element of c, is stored as the secondary element of the return value.</td>
</tr>
</tbody>
</table>
| **Formula**         | primary(result) = a x primary(b) + primary(c)  
secondary(result) = a x secondary(b) + secondary(c) |
| **C/C++ prototype** | double _Complex __fxcpmadd (double _Complex c, double _Complex b, double a);  
double _Complex __fxcsmadd (double _Complex c, double _Complex b, double a); |
<table>
<thead>
<tr>
<th>Fortran description</th>
<th>FXCPMADD(C,B,A) or FXCSMADD(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function</strong></td>
<td>**Cross copy negative multiply-add: <strong>__fxcpnmadd, __fxcsnmadd</strong></td>
</tr>
<tr>
<td><strong>Purpose</strong></td>
<td>Both of these functions can be used to achieve the same result. The difference of the primary element of c, subtracted from the product of a and the primary element of b, is negated and stored as the primary element of the return value. The difference of the secondary element of c, subtracted from the product of a and the secondary element of b, is negated and stored as the secondary element of the return value.</td>
</tr>
<tr>
<td><strong>Formula</strong></td>
<td>primary(result) = -(a x primary(b) + primary(c)) secondary(result) = -(a x secondary(b) + secondary(c))</td>
</tr>
<tr>
<td><strong>C/C++ prototype</strong></td>
<td>double _Complex __fxcpnmadd (double _Complex c, double _Complex b, double a); double _Complex __fxcsnmadd (double _Complex c, double _Complex b, double a);</td>
</tr>
<tr>
<td>Fortran description</td>
<td>FXCPNMADD(C,B,A) or FXCSNMADD(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)</td>
</tr>
<tr>
<td><strong>Function</strong></td>
<td>**Cross copy multiply-subtract: **__fxcpmsub, __fxcsmsub</td>
</tr>
<tr>
<td><strong>Purpose</strong></td>
<td>Both of these functions can be used to achieve the same result. The difference of the primary element of c, subtracted from the product of a and the primary element of b, is stored as the primary element of the return value. The difference of the secondary element of c, subtracted from the product of a and the secondary element of b, is stored as the secondary element of the return value.</td>
</tr>
<tr>
<td><strong>Formula</strong></td>
<td>primary(result) = a x primary(b) - primary(c) secondary(result) = a x secondary(b) - secondary(c)</td>
</tr>
<tr>
<td><strong>C/C++ prototype</strong></td>
<td>double _Complex __fxcpmsub (double _Complex c, double _Complex b, double a); double _Complex __fxcsmsub (double _Complex c, double _Complex b, double a);</td>
</tr>
<tr>
<td>Fortran description</td>
<td>FXCPMSUB(C,B,A) or FXCSMSUB(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)</td>
</tr>
<tr>
<td><strong>Function</strong></td>
<td>**Cross copy negative multiply-subtract: **__fxcpnmsub, __fxcsnmsub</td>
</tr>
<tr>
<td><strong>Purpose</strong></td>
<td>Both of these functions can be used to achieve the same result. The difference of the primary element of c, subtracted from the product of a and the primary element of b, is negated and stored as the primary element of the return value. The difference of the secondary element of c, subtracted from the product of a and the secondary element of b, is negated and stored as the secondary element of the return value.</td>
</tr>
<tr>
<td><strong>Formula</strong></td>
<td>primary(result) = -(a x primary(b) - primary(c)) secondary(result) = -(a x secondary(b) - secondary(c))</td>
</tr>
<tr>
<td><strong>C/C++ prototype</strong></td>
<td>double _Complex __fxcpnmsub (double _Complex c, double _Complex b, double a); double _Complex __fxcsnmsub (double _Complex c, double _Complex b, double a);</td>
</tr>
</tbody>
</table>

130 Blue Gene/P Application Development
| Fortran description | FXCPNMSUB(C,B,A) or FXCSNMSUB(C,B,A)  
where C is of type COMPLEX(8)  
where B is of type COMPLEX(8)  
where A is of type REAL(8)  
result is of type COMPLEX(8) |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>Cross copy sub-primary multiply-add: __fxcpnpma, __fxcsnpma</td>
</tr>
<tr>
<td>Purpose</td>
<td>Both of these functions can be used to achieve the same result. The difference of the primary element of c, subtracted from the product of a and the primary element of b, is negated and stored as the primary element of the return value. The sum of the product of a and the secondary element of b, added to the secondary element of c, is stored as the secondary element of the return value.</td>
</tr>
</tbody>
</table>
| Formula             | primary(result) = -(a x primary(b) - primary(c))  
secondary(result) = a x secondary(b) + secondary(c) |
| C/C++ prototype     | double _Complex __fxcpnpma (double _Complex c, double _Complex b, double a);  
double _Complex __fxcsnpma (double _Complex c, double _Complex b, double a); |

| Fortran description | FXCPNPMA(C,B,A) or FXCSNPMA(C,B,A)  
where C is of type COMPLEX(8)  
where B is of type COMPLEX(8)  
where A is of type REAL(8)  
result is of type COMPLEX(8) |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>Cross copy sub-secondary multiply-add: __fxcpnsma, __fxcsnsma</td>
</tr>
<tr>
<td>Purpose</td>
<td>Both of these functions can be used to achieve the same result. The sum of the product of a and the primary element of b, added to the primary element of c, is stored as the primary element of the return value. The difference of the secondary element of c, subtracted from the product of a and the secondary element of b, is negated and stored as the secondary element of the return value.</td>
</tr>
</tbody>
</table>
| Formula             | primary(result) = a x primary(b) + primary(c))  
secondary(result) = -(a x secondary(b) - secondary(c)) |
| C/C++ prototype     | double _Complex __fxcpnsma (double _Complex c, double _Complex b, double a);  
double _Complex __fxcsnsma (double _Complex c, double _Complex b, double a); |

| Fortran description | FXCPNSMA(C,B,A) or FXCSNSMA(C,B,A)  
where C is of type COMPLEX(8)  
where B is of type COMPLEX(8)  
where A is of type REAL(8)  
result is of type COMPLEX(8) |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>Cross mixed multiply-add: __fxcxma</td>
</tr>
<tr>
<td>Purpose</td>
<td>The sum of the product of a and the secondary element of b, added to the primary element of c, is stored as the primary element of the return value. The sum of the product of a and the primary element of b, added to the secondary element of c, is stored as the secondary element of the return value.</td>
</tr>
</tbody>
</table>
| Formula             | primary(result) = a x secondary(b) + primary(c))  
secondary(result) = a x primary(b) + secondary(c) |
<p>| C/C++ prototype     | double _Complex __fxcxma (double _Complex c, double _Complex b, double a); |</p>
<table>
<thead>
<tr>
<th>Forran description</th>
<th>FXCXMA(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function</strong></td>
<td>Cross mixed negative multiply-subtract: __fcxnms</td>
</tr>
<tr>
<td><strong>Purpose</strong></td>
<td>The difference of the primary element of ( c ), subtracted from the product of ( a ) and the secondary element of ( b ), is negated and stored as the primary element of the return value. The difference of the secondary element of ( c ), subtracted from the product of ( a ) and the primary element of ( b ), is negated and stored as the primary secondary of the return value.</td>
</tr>
<tr>
<td><strong>Formula</strong></td>
<td>primary(result) = -(a × secondary(b) - primary(c)) secondary(result) = -(a × primary(b) - secondary(c))</td>
</tr>
<tr>
<td>C/C++ prototype</td>
<td>double _Complex __fcxnms (double _Complex c, double _Complex b, double a);</td>
</tr>
<tr>
<td>Fortran description</td>
<td>FXCXNMS(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)</td>
</tr>
<tr>
<td><strong>Function</strong></td>
<td>Cross mixed sub-primary multiply-add: __fcxnpm</td>
</tr>
<tr>
<td><strong>Purpose</strong></td>
<td>The difference of the primary element of ( c ), subtracted from the product of ( a ) and the secondary element of ( b ), is stored as the primary element of the return value. The sum of the product of ( a ) and the primary element of ( b ), added to the secondary element of ( c ), is stored as the secondary element of the return value.</td>
</tr>
<tr>
<td><strong>Formula</strong></td>
<td>primary(result) = -(a × secondary(b) - primary(c)) secondary(result) = a × primary(b) + secondary(c)</td>
</tr>
<tr>
<td>C/C++ prototype</td>
<td>double _Complex __fcxnpm (double _Complex c, double _Complex b, double a);</td>
</tr>
<tr>
<td>Fortran description</td>
<td>FXCXNPMA(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)</td>
</tr>
<tr>
<td><strong>Function</strong></td>
<td>Cross mixed sub-secondary multiply-add: __fcxnsm</td>
</tr>
<tr>
<td><strong>Purpose</strong></td>
<td>The sum of the product of ( a ) and the secondary element of ( b ), added to the primary element of ( c ), is stored as the primary element of the return value. The difference of the secondary element of ( c ), subtracted from the product of ( a ) and the primary element of ( b ), is stored as the secondary element of the return value.</td>
</tr>
<tr>
<td><strong>Formula</strong></td>
<td>primary(result) = a × secondary(b) + primary(c)) secondary(result) = -(a × primary(b) - secondary(c))</td>
</tr>
<tr>
<td>C/C++ prototype</td>
<td>double _Complex __fcxnsm (double _Complex c, double _Complex b, double a);</td>
</tr>
</tbody>
</table>
8.13.12 Select functions

Table 8-9 lists and explains the parallel select functions that are available.

Table 8-9  Select functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Parallel select: __fpsel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>The value of the primary element of a is compared to zero. If its value is equal to or greater than zero, the primary element of c is stored in the primary element of the return value. Otherwise, the primary element of b is stored in the primary element of the return value. The value of the secondary element of a is compared to zero. If its value is equal to or greater than zero, the secondary element of c is stored in the secondary element of the return value. Otherwise, the secondary element of b is stored in the secondary element of the return value.</td>
</tr>
<tr>
<td>Formula</td>
<td>primary(result) = if primary(a) ( \geq 0 ) then primary(c); else primary(b) secondary(result) = if secondary(a) ( \geq 0 ) then primary(c); else secondary(b)</td>
</tr>
<tr>
<td>C/C++ prototype</td>
<td>double _Complex __fpsel (double _Complex a, double _Complex b, double _Complex c);</td>
</tr>
<tr>
<td>Fortran description</td>
<td>FPSEL(A,B,C) where A is of type COMPLEX(8) where B is of type COMPLEX(8) where C is of type COMPLEX(8) result is of type COMPLEX(8)</td>
</tr>
</tbody>
</table>

8.13.13 Examples of built-in functions usage

Using the following definitions, you can create a custom parallel add function that uses the parallel load and add built-in functions to add two double floating-point values in parallel and return the result as a complex number. See Example 8-15 for C/C++ and Example 8-16 for Fortran.

Example 8-15  Using built-in functions in C/C++

```c
double _Complex padd(double *x, double *y)
{
    double _Complex a,b,c;
    /* note possibility of alignment trap if (((unsigned int) x) % 32) >= 17) */

    a = __lfpd(x);  //load x[0] to the primary part of a, x[1] to the secondary part of a
    b = __lfpd(y);  //load y[0] to primary part of b, y[1] to the secondary part of b
    c = __fpadd(a,b);  // the primary part of c = x[0] + y[0]
    /* the secondary part of c = x[1] + y[1] */
    return c;

    /* alternately: */
    return __fpadd(__lfpd(x), __lfpd(y));    /* same code generated with optimization enabled */
}
```

Example 8-16  Using built-in functions in Fortran

```fortran
FUNCTION PADD (X, Y)
    COMPLEX(8) PADD
    REAL(8) X, Y
    COMPLEX(8) A, B, C
    FPSEL(A,B,C) where A is of type COMPLEX(8) where B is of type COMPLEX(8) where C is of type COMPLEX(8) result is of type COMPLEX(8)
```
A = LOADFP(X)
B = LOADFP(Y)
PADD = FPADD(A,B)

RETURN
END

Example 8-17 provides a sample of double-precision square matrix-matrix multiplication. This version uses 6x4 outer loop unrolling.

Example 8-17  Double-precision square matrix multiply example

subroutine dsqmm(a, b, c, n)
!
!# (C) Copyright IBM Corp. 2006 All Rights Reserved.
!# Rochester, MN
!
implicit none
integer i, j, k, n
integer ii, jj, kk
integer istop, jstop, kstop
integer, parameter :: nb = 36   ! blocking factor
complex(8) zero
complex(8) a00, a01
complex(8) a20, a21
complex(8) b0, b1, b2, b3, b4, b5
complex(8) c00, c01, c02, c03, c04, c05
complex(8) c20, c21, c22, c23, c24, c25
real(8) a(n,n), b(n,n), c(n,n)
zero = (0.0d0, 0.0d0)
!
!------------------------------------------------------
! Double-precision square matrix-matrix multiplication.
!------------------------------------------------------
! This version uses 6x4 outer loop unrolling.
! The cleanup loops have been left out, so the results
! are correct for dimensions that are multiples of the
! two unrolling factors: 6 and 4.
!------------------------------------------------------

do jj = 1, n, nb
    if ((jj + nb - 1) .lt. n) then
        jstop = (jj + nb - 1)
    else
        jstop = n
    endif

do ii = 1, n, nb
    if ((ii + nb - 1) .lt. n) then
        istop = (ii + nb - 1)
    else
        istop = n
    endif
!
!---------------------------------
! initialize a block of c to zero
!---------------------------------
do j = jj, jstop - 5, 6
  do i = ii, istop - 1, 2
    call storefp(c(i,j) , zero)
    call storefp(c(i,j+1), zero)
    call storefp(c(i,j+2), zero)
    call storefp(c(i,j+3), zero)
    call storefp(c(i,j+4), zero)
    call storefp(c(i,j+5), zero)
  end do
end do

!--------------------------------------------------------
! multiply block by block with 6x4 outer loop un-rolling
!--------------------------------------------------------

do kk = 1, n, nb
  if ((kk + nb - 1) .lt. n) then
    kstop = (kk + nb - 1)
  else
    kstop = n
  endif

  do j = jj, jstop - 5, 6
    do i = ii, istop - 3, 4
      c00 = loadfp(c(i,j  ))
      c01 = loadfp(c(i,j+1))
      c02 = loadfp(c(i,j+2))
      c03 = loadfp(c(i,j+3))
      c04 = loadfp(c(i,j+4))
      c05 = loadfp(c(i,j+5))
      c20 = loadfp(c(i+2,j  ))
      c21 = loadfp(c(i+2,j+1))
      c22 = loadfp(c(i+2,j+2))
      c23 = loadfp(c(i+2,j+3))
      c24 = loadfp(c(i+2,j+4))
      c25 = loadfp(c(i+2,j+5))
      a00 = loadfp(a(i,kk ))
      a20 = loadfp(a(i+2,kk ))
      a01 = loadfp(a(i,kk+1))
      a21 = loadfp(a(i+2,kk+1))
      do k = kk, kstop - 1, 2
        b0 = loadfp(b(k,j ))
        b1 = loadfp(b(k,j+1))
        b2 = loadfp(b(k,j+2))
        b3 = loadfp(b(k,j+3))
        b4 = loadfp(b(k,j+4))
        b5 = loadfp(b(k,j+5))
        c00 = fxcpmadd(c00, a00, real(b0))
        c01 = fxcpmadd(c01, a00, real(b1))
        c02 = fxcpmadd(c02, a00, real(b2))
        c03 = fxcpmadd(c03, a00, real(b3))
        c04 = fxcpmadd(c04, a00, real(b4))
        c05 = fxcpmadd(c05, a00, real(b5))
        c20 = fxcpmadd(c20, a20, real(b0))
        c21 = fxcpmadd(c21, a20, real(b1))
        c22 = fxcpmadd(c22, a20, real(b2))
        c23 = fxcpmadd(c23, a20, real(b3))
    end do
  end do
end do
c24 = fxcpmadd(c24, a20, real(b4))
c25 = fxcpmadd(c25, a20, real(b5))
a00 = loadfp(a(i,k+2))
a20 = loadfp(a(i+2,k+2))
c00 = fxcpmadd(c00, a01, imag(b0))
c01 = fxcpmadd(c01, a01, imag(b1))
c02 = fxcpmadd(c02, a01, imag(b2))
c03 = fxcpmadd(c03, a01, imag(b3))
c04 = fxcpmadd(c04, a01, imag(b4))
c05 = fxcpmadd(c05, a01, imag(b5))
c20 = fxcpmadd(c20, a21, imag(b0))
c21 = fxcpmadd(c21, a21, imag(b1))
c22 = fxcpmadd(c22, a21, imag(b2))
c23 = fxcpmadd(c23, a21, imag(b3))
c24 = fxcpmadd(c24, a21, imag(b4))
c25 = fxcpmadd(c25, a21, imag(b5))
a01 = loadfp(a(i,k+3))
a21 = loadfp(a(i+2,k+3))
end do

call storefp(c(i,j), c00)
call storefp(c(i,j+1), c01)
call storefp(c(i,j+2), c02)
call storefp(c(i,j+3), c03)
call storefp(c(i,j+4), c04)
call storefp(c(i,j+5), c05)
call storefp(c(i+2,j), c20)
call storefp(c(i+2,j+1), c21)
call storefp(c(i+2,j+2), c22)
call storefp(c(i+2,j+3), c23)
call storefp(c(i+2,j+4), c24)
call storefp(c(i+2,j+5), c25)
end do

end do
end do !kk
end do !ii

end do !jj

end
Running and debugging applications

In this chapter, we explain how to run and debug applications on the Blue Gene/P system. These types of tools are essential for applications developers. Although, we do not cover all of the existing tools, we provide an overview of some of the currently available tools.

We cover the following topics:

- Running applications
- Debugging applications
9.1 Running applications

Blue Gene/P applications can be run in several ways. We briefly discuss each method and provide references for more detailed documentation.

9.1.1 MMCS console

It is possible to run applications directly from the MMCS console. The main drawback to using this approach is that it requires users to have direct access to the service node, which is undesirable from a security perspective.

When using the MMCS console, it is necessary to first manually select and allocate a block. A block in this case refers to a partition or set of nodes to run the job. (See Appendix A, “Blue Gene/P hardware-naming conventions” on page 287, for more information.) At this point, it is possible to run Blue Gene/P applications. The set of commands in Example 9-1 from the MMCS console window show how to accomplish this. The names can be site specific, but the example illustrates the procedure.

To start the console session, use the sequence of commands shown in Example 9-1 on the service node.

Example 9-1  Starting the console session

cd /bgsys/drivers/ppcfloor/bin
source ~bgpsysdb/sqllib/db2profile
mmcs_db_console --bgpdmingroup p/bluegene/bgpall
connecting to mmcs_server
connected to mmcs_server
connected to DB2
mmcs$list_blocks
OK
N00_64_1   B manojd  (1)  connected
N02_32_1   I walkup  (0)  connected
N04_32_1   B manojd  (1)  connected
N05_32_1   B manojd  (1)  connected
N06_32_1   I sameer77(1)  connected
N07_32_1   I gdozsa  (1)  connected
N08_64_1   I vezolle (1)  connected
N12_32_1   I vezolle (0)  connected
mmcs$ allocate N14_32_1
OK
mmcs$ list_blocks
OK
N00_64_1   B manojd  (1)  connected
N02_32_1   I walkup  (0)  connected
N04_32_1   B manojd  (1)  connected
N05_32_1   B manojd  (1)  connected
N06_32_1   I sameer77(1)  connected
N07_32_1   I gdozsa  (1)  connected
N08_64_1   I vezolle (1)  connected
N12_32_1   I vezolle (0)  connected
N14_32_1   I cpsosa  (1)  connected
mmcs$ submitjob N14_32_1 /bgusr/cpsosa/hello/c/omp_hello_bgp /bgusr/cpsosa/hello/c
OK
jobId=14008
mmcs$ free N14_32_1
OK
mmcs$ quit
OK
mmcs_db_console is terminating, please wait...
mmcs_db_console: closing database connection
mmcs_db_console: closed database connection
mmcs_db_console: closing console port
mmcs_db_console: closed console port

For more information about using the MMCS console, see *IBM System Blue Gene Solution: Blue Gene/P System Administration*, SG24-7417.

### 9.1.2 mpirun

In the absence of a scheduling application, we recommend that you use `mpirun` to run Blue Gene/P applications on statically allocated partitions. Users can access this application from the front end node, which provides better security protection than using the MMCS console. For more complete information about using `mpirun`, see Chapter 11, “mpirun” on page 167.

With `mpirun`, you can select and allocate a block and run a Message Passing Interface (MPI) application, all in one step as shown in Example 9-2.

**Example 9-2 Using mpirun**

```bash
cpsosa@descartes:/bgusr/cpsosa/red/pi/c> csh
descartes pi/c> set MPIRUN="/bgsys/drivers/ppcfloor/bin/mpirun"
descartes pi/c> set MPIOPT="-np 1"
descartes pi/c> set MODE="-mode SMP"
descartes pi/c> set PARTITION="-partition N14_32_1"
descartes pi/c> set WDIR="-cwd /bgusr/cpsosa/red/pi/c"
descartes pi/c> set EXE="-exe /bgusr/cpsosa/red/pi/c/pi_critical_bgp"
descartes pi/c> $MPIRUN $PARTITION $MPIOPT $MODE $WDIR $EXE -env "OMP_NUM_THREADS=1"
Estimate of pi: 3.14159
Total time 560.055988
```

All output in this example is sent to the display. To specify that you want this information sent to a file, you must add the following line, for example, to the end of the `mpirun` command:

```bash
>bgusr/cpsosa/red/pi/c/pi_critical.stdout 2>bgusr/cpsosa/red/pi/c/pi_critical.stderr
```

This line sends standard output to the `pi_critical.stdout` file and standard error to the `pi_critical.stderr` file. Both files are in the `/bgusr/cpsosa/red/pi/c` directory.

### 9.1.3 submit

In HTC mode you must use the `submit` command, which is analogous to `mpirun` because its purpose is to act as a shadow of the job. It transparently forwards stdin, and receives stdout and stderr. More detailed usage information is available in Chapter 12, “High-Throughput Computing (HTC) paradigm” on page 187.

### 9.1.4 LoadLeveler

At present, LoadLeveler support for the Blue Gene/P system is provided via a programming request for price quotation (PRPQ). The IBM Tivoli® Workload Scheduler LoadLeveler product is intended to manage both serial and parallel jobs over a cluster of servers. This
distributed environment consists of a pool of machines or servers, often referred to as a
LoadLeveler cluster. Machines in the pool can be of several types: desktop workstations
available for batch jobs (usually when not in use by their owner), dedicated servers, and
parallel machines.

LoadLeveler allocates machine resources in the cluster to run jobs. The scheduling of jobs
depends on the availability of resources within the cluster and various rules, which can be
defined by the LoadLeveler administrator. A user submits a job using a job command file. The
LoadLeveler scheduler attempts to find resources within the cluster to satisfy the
requirements of the job. LoadLeveler maximizes the efficiency of the cluster by maximizing
the utilization of resources, while at the same time minimizing the job turnaround time
experienced by users.

LoadLeveler provides a rich set of functions for job scheduling and cluster resource
management. Some of the tasks that LoadLeveler can perform include:

- Choosing the next job to run.
- Examining the job requirements.
- Collecting available resources in the cluster.
- Choosing the “best” machines for the job.
- Dispatching the job to the selected machine.
- Controlling running jobs.
- Creating reservations and scheduling jobs to run in the reservations.
- Job preemption to enable high-priority jobs to run immediately.
- Fair share scheduling to automatically balance resources among users or groups of users.
- Co-scheduling to enable several jobs to be scheduled to run at the same time.
- Multi-cluster support to allow several LoadLeveler clusters to work together to run user
  jobs.

For more information about LoadLeveler support, see Chapter 10 of IBM System Blue Gene
Solution: Configuring and Maintaining Your Environment, SG24-7352, which describes step
by step how to use LoadLeveler on the Blue Gene/L system. Almost all of the contents are
applicable to a Blue Gene/P system.

9.1.5 Other scheduler products

You can use custom scheduling applications to run applications on the Blue Gene/P system.
You write custom “glue” code between the scheduler and the Blue Gene/P system by using
the Bridge APIs, which are described in Chapter 13, “Control system (Bridge) APIs” on
page 195, and Chapter 14, “Real-time Notification APIs” on page 237.

9.2 Debugging applications

In this section, we discuss the debuggers that are supported by the Blue Gene/P system.

Note: You cannot debug applications in HTC mode; use mpirun -np 1.
9.2.1 General debugging architecture

Four pieces of code are involved when debugging applications on the Blue Gene/P system:

- The compute node kernel, which provides the low-level primitives that are necessary to debug an application
- The control and I/O daemon (CIOD) running on the I/O nodes, which provides control and communications to compute nodes
- A “debug server” running on the I/O nodes, which is vendor-supplied code that interfaces with the CIOD
- A debug client running on a front end node, which is where the user does their work interactively

A debugger must interface to the compute node through an API implemented in CIOD to debug an application running on a compute node. This debug code is started on the I/O nodes by the control system and can interface with other software, such as a GUI or command-line utility on a front-end system. The code running on the I/O nodes using the API in CIOD is referred to as a debug server. It is provided by the debugger vendor for use with the Blue Gene/P system. Many possible debug servers are possible.

A debug client is a piece of code that runs on a front end node that the user interacts with directly. It makes remote requests to the debug server running on the I/O nodes, which in turn passes the request through CIOD and eventually to the compute node. The debug client and debug server usually communicate using TCP/IP.

9.2.2 GNU Project debugger

The GNU Project debugger (GDB) is the primary debugger of the GNU project. You can learn more about GDB on the Web at the following address:

http://www.gnu.org/software/gdb/gdb.html

A great amount of documentation is available about the GDB. Because we do not discuss how to use it in this book, refer to the following Web site for details:

http://www.gnu.org/software/gdb/documentation/

Support has been added to the Blue Gene/P system for which the GDB can work with applications that run on compute nodes. IBM provides a simple debug server called gdbserver. Each running instance of GDB is associated with one, and only one, compute node. If you must debug an MPI application that runs on multiple compute nodes, and you must, for example, view variables that are associated with more than one instance of the application, you run multiple instances of GDB.

Most people use GDB to debug local processes that run on the same machine on which they are running GDB. With GDB, you also have the ability to debug remotely via a GDB server on the remote machine. GDB on the Blue Gene/L system is used in this mode. We refer to GDB as the GDB client, although most users recognize it as GDB used in a slightly different manner.
Limitations

Gdbserver implements the minimum number of primitives required by the GDB remote protocol specification. As such, advanced features that might be available in other implementations are not available in this implementation. However, sufficient features are implemented to make it a useful tool. Here are some of the limitations:

- Each instance of a GDB client can connect to and debug one compute node. To debug multiple compute nodes at the same time, you must run multiple GDB clients at the same time. Although you might need multiple GDB clients for multiple compute nodes, one gdbserver on each I/O node is all that is required. The Blue Gene/P control system manages that part.

- IBM does not ship a GDB client with the Blue Gene/P system. The user can use an existing GDB client to connect to the IBM-supplied gdbserver. Most functions will work, but standard GDB clients are not aware of the full “double hummer” floating-point register set that Blue Gene/L provides. The GDB clients that come with SUSE Linux Enterprise Server (SLES) 10 for PowerPC are known to work.

- To debug an application, the debug server must be started and running before you attempt to debug. Using an option on the mpirun command, you can get the debug server running before your application does. If you do not use this option and you must debug your application, you do not have a mechanism to start the debug server and thus have no way to debug your application.

- Gdbserver is not aware of user-specified MPI topologies. You can still debug your application, but the connection information given to you by mpirun for each MPI rank can be incorrect.

Prerequisite software

The GDB should have been installed during the installation procedure. You can verify the installation by seeing whether the /bgsys/drivers/ppcfloor/gnu-linux/bin/gdb file exists on your front end node.

The rest of the software support required for GDB should be installed as part of the control programs.

Preparing your program

The MPI, OpenMP, or MPI-OpenMP program that you want to debug must be compiled in a manner that allows for debugging information (symbol tables, ties to source, and so on) to be included in the executable. In addition, do not use compiler optimization because it makes it difficult, if not impossible, to tie object code back to source. For example, when compiling a program written in Fortran that you want to debug, compile the application using an invocation similar to one shown in Example 9-3.

Example 9-3 Makefile used for building the program with debugging flags

```bash
BGP_FLOOR   = /bgsys/drivers/ppcfloor
BGP_IDIRS   = -I$(BGP_FLOOR)/arch/include -I$(BGP_FLOOR)/comm/include
BGP_LIBS    = -L$(BGP_FLOOR)/comm/lib -lmpich.cnk -L$(BGP_FLOOR)/comm/lib -ldcmfcoll.cnk
             -ldcmf.cnk -lpthread -lrt -L$(BGP_FLOOR)/runtime/SPI -lSPI.cna

XL         = /opt/ibmcmp/xlf/bg/11.1/bin/bgxlf90

EXE         = example_9_4_bgp
OBJ         = example_9_4.o
SRC         = example_9_4.f
FLAGS       = -g -00 -qarch=450 -qtune=450 -I$(BGP_FLOOR)/comm/include
```
$(EXE): $(OBJ) $(XL) $(FLAGS) -o $(EXE) $(OBJ) $(BGP_LIBS)
$(OBJ): $(SRC) $(XL) $(FLAGS) $(BGP_IDIRS) -c $(SRC)

clean:
    rm *.o example_9_4_bgp

cpsosa@descartes:/bgusr/cpsosa/red/debug> make
/opt/ibmcmp/xlf/bg/11.1/bin/bgxlf90 -g -00 -qarch=450 -qtune=450
-I/bgsys/drivers/ppcfloor/comm/include -I/bgsys/drivers/ppcfloor/arch/include
-I/bgsys/drivers/ppcfloor/comm/include -c example_9_4.f
** nooffset === End of Compilation 1 ===
1501-510  Compilation successful for file example_9_4.f.
/opt/ibmcmp/xlf/bg/11.1/bin/bgxlf90 -g -00 -qarch=450 -qtune=450
-I/bgsys/drivers/ppcfloor/comm/include -o example_9_4_bgp example_9_4.o
-ldcmf.cnk -lpthread -lrt -L/bgsys/drivers/ppcfloor/runtime/SPI -lSPI.cna

The -g switch tells the compiler to include debug information. The -00 (the letter capital “O” followed by a zero) switch tells it to disable optimization.

For more information about the IBM XL compilers for the Blue Gene/P system, see Chapter 8, “Developing applications with IBM XL compilers” on page 95.

**Important:** Make sure that the text file that contains the source for your program is located in the same directory as the program itself and has the same file name (different extension).

### Debugging

Follow the steps in this section to start debugging your application. For the sake of this example, let us say that the program's name is example_9_4_bgp as illustrated in Example 9-4 on page 144 (source code not shown), and the source code file is example_9_4.f. We use a partition (block) called N14_32_1.

An extra parameter (-start_gdbserver...) is passed in on the mpirun command. The extra option changes the way mpirun loads and executes your code. Here is a brief summary of the changes:

1. The code is loaded onto the compute nodes (in our example, the executable is example_9_4_bgp), but it does not start running immediately.
2. The control system starts the specified debug server (gdbserver) on all of the I/O nodes in the partition that is running your job, which in our example is N14_32_1.
3. The mpirun command pauses, so that you get a chance to connect GDB clients to the compute nodes that you are going to debug.
4. When you are finished connecting GDB clients to compute nodes, you press Enter to signal the mpirun command, and then the application starts running on the compute nodes.

During the pause in step 3, you have an opportunity to connect the GDB clients to the compute nodes before the application runs, which is desirable if you must start the application under debugger control. This step is optional. If you do not connect before the application
starts running on the compute nodes, you can still connect later because the debugger server was started on the I/O nodes.

To start debugging your application:

1. Open two separate console shells.
2. Go to the first shell window.
   a. Change to the directory (cd) that contains your program executable. In our example, the directory is /bgusr/cpsosa/red/debug.
   b. Start your application using mpirun with a command similar to the one shown in Example 9-4. You should see messages in the console, similar to those shown in Example 9-4.

Example 9-4  Messages in the console

```bash
set MPIRUN="/bgsys/drivers/ppcfloor/bin/mpirun"
set MPIOPT="-np 1"
set MODE="-mode SMP"
set PARTITION="-partition N14_32_1"
set WDIR="-cwd /bgusr/cpsosa/red/debug"
set EXE="-exe /bgusr/cpsosa/red/debug/example_9_4_bgp"
#
$MPIRUN $PARTITION $MPIOPT $MODE $WDIR $EXE -env "OMP_NUM_THREADS=4" -start_gdbserver /bgsys/drivers/ppcfloor/ramdisk/sbin/gdbserver -verbose 1
#
```

descartes red/debug> set EXE="-exe /bgusr/cpsosa/red/debug/example_9_4_bgp"
descartes red/debug> $MPIRUN $PARTITION $MPIOPT $MODE $WDIR $EXE -env "OMP_NUM_THREADS=4" -start_gdbserver /bgsys/drivers/ppcfloor/ramdisk/sbin/gdbserver -verbose 1
<Sep 15 10:14:58.642369> FE_MPI [Info] : Invoking mpirun backend
<Sep 15 10:14:05.741121> BRIDGE [Info] : rm_set_serial() - The machine serial number (alias) is BGP
<Sep 15 10:15:00.461655> FE_MPI [Info] : Preparing partition
<Sep 15 10:14:05.821585> BE_MPI [Info] : Examining specified partition
<Sep 15 10:14:10.085997> BE_MPI [Info] : Checking partition N14_32_1 initial state ...
<Sep 15 10:14:10.086041> BE_MPI [Info] : Partition N14_32_1 initial state = READY ('I')
<Sep 15 10:14:10.086059> BE_MPI [Info] : Checking partition owner ...
<Sep 15 10:14:10.086087> BE_MPI [Info] : partition N14_32_1 owner is 'cpsosa'
<Sep 15 10:14:10.088375> BE_MPI [Info] : Partition owner matches the current user
<Sep 15 10:14:10.088470> BE_MPI [Info] : Done preparing partition
<Sep 15 10:15:04.804078> FE_MPI [Info] : Adding job to database ...
<Sep 15 10:15:06.104035> FE_MPI [Info] : Job added with the following id: 14035
<Sep 15 10:15:06.104096> FE_MPI [Info] : Loading Blue Gene job
<Sep 15 10:14:11.426987> BE_MPI [Info] : Loading job 14035 ...
<Sep 15 10:14:11.450495> BE_MPI [Info] : Job load command successful
<Sep 15 10:14:11.450525> BE_MPI [Info] : Waiting for job 14035 to get to Loaded/Running state ...
<Sep 15 10:14:16.458474> BE_MPI [Info] : Job 14035 switched to state LOADED
<Sep 15 10:14:21.467401> BE_MPI [Info] : Job loaded successfully
<Sep 15 10:15:16.179023> FE_MPI [Info] : Starting debugger setup for job 14035
<Sep 15 10:15:16.179090> FE_MPI [Info] : Setting debug info in the block record
<Sep 15 10:14:21.502593> BE_MPI [Info] : Setting debugger executable and arguments in block description
```

144  Blue Gene/P Application Development
Chapter 9. Running and debugging applications

Make your connections to the compute nodes now - press [Enter] when you are ready to run the app. To see the IP connection information for a specific compute node, enter its MPI rank and press [Enter]. To see all of the compute nodes, type 'dump_proctable'.

Example 9-5 Finding the IP address and port of the compute node for debugging

> 2
MPI Rank 2: Connect to 172.30.255.85:7302
> 4
MPI Rank 4: Connect to 172.30.255.85:7304
> or
> dump_proctable
3. From the second shell, follow these steps:
   a. Change to the directory (cd) that contains your program executable.
   b. Type the following command, using the name of your own executable instead of
      example_9_4_bgp:
      
      /bgsys/drivers/ppcfloor/gnu-linux/bin/gdb example_9_4_bgp
   c. Enter the following command, using the address of the compute node that you want to
      debug and determined in step 2:
      
      target remote ipaddr:port

You are now debugging the specified application on the configured compute node.

4. Set one or more breakpoints (using the GDB break command). Press Enter from the first
   shell to continue that application.

   If successful, your breakpoint should eventually be reached in the second shell, and you
   can use standard GDB commands to continue.

9.2.3 Core Processor debugger

Core Processor is a basic tool that can help you debug your application. This tool is
discussed in detail in IBM System Blue Gene Solution: Blue Gene/P System Administration,
SG24-7417. In the following sections, we briefly describe how to use it to debug applications.

9.2.4 Starting the Core Processor tool

To start the Core Processor tool:

1. Export DISPLAY and make sure it works.
2. Type coreprocessor.pl to specify the Core Processor tool. You might need to specify the
   full path.
3. From the GUI window that opens, click OK. The Perl script is invoked automatically.
Figure 9-1 shows how the Core Processor tool GUI looks after the Perl script is invoked. The Core Processor windows do not provide any initial information. You must explicitly select a task that is provided via the GUI.

9.2.5 Attaching running applications

To do a live debug on compute nodes:

1. Start the Core Processor GUI as explained in the previous section.
2. Select File → Attach To Block.
3. In the Attach Coreprocessor window (see Figure 9-2), supply the following information:
   - Session Name: You can run more than one session at a time, so use this option to distinguish between multiple sessions.
   - Block name.
   - CNK binary (with path): To see both your application and the compute node kernel in the stack, specify your application binary and the compute node kernel image separated by a colon (:) as shown in the following example:
     
     ```
     /bgsys/drivers/ppcfloor/cnk/bgp_kernel.cnk:/bguser/bguser/hello_mpi_loop.rts
     ```
   - User name or owner of the Midplane Management Control System (MMCS) block.
   - Port: TCP port on which the MMCS server is listening for console connections, which is probably 32031.
   - Host name or TCP/IP address for the MMCS server: Typically it is localhost or the service node’s TCP/IP address.

   Click the Attach button.

   ![Core Processor attach window](image)

   **Figure 9-2 Core Processor attach window**
4. At this point, you have not yet affected the state of the processors. Choose **Select Grouping Mode → Processor Status**.

Notice the text in the upper-left pane (Figure 9-3). The Core Processor tool posts the status `?RUN?` because it does not yet know the state of the processors. (2048) is the number of nodes in the block that are in that state. The number in parentheses always indicates the number of nodes that share the attribute displayed on the line, which is the processor state in this case.

![Figure 9-3 Processor status](image)

5. Back at the main window (refer to Figure 9-1 on page 147), click the **Select Grouping Mode** button.

6. Choose one of the **Stack Traceback** options. The Core Processor tool halts all the compute node processor cores and displays the requested information. Choose each of the options on that menu in turn so that you can see the variety of data formats available.
Stack Traceback (condensed)

In the condensed version of Stack Traceback, data from all nodes is captured. The unique instruction addresses per stack frame are grouped and displayed. However, the last stack frame is grouped based on the function name, not the IAR. This is normally the most useful mode for debug (Figure 9-4).

![Figure 9-4 Stack Traceback (condensed)](image-url)
Stack Traceback (detailed)

In Stack Traceback (detailed), data from all nodes is captured (Figure 9-5). The unique instruction addresses per stack frame are grouped and displayed. The IAR at each stack frame is also displayed.

![Stack Traceback (detailed)](image-url)
Stack Traceback (survey)

Stack Traceback (survey) is a quick but potentially inaccurate mode. IARs are initially captured, and stack data is collected for each node from a group of nodes that contain the same IAR. The stack data fetched for that one node is then applied to all nodes with the same IAR. Figure 9-6 shows an example of the survey mode.

![Figure 9-6 Stack Traceback (survey)](image)

Refer to the following points to help you use the tool more effectively:

- The number at the far left, before the colon, indicates the depth within the stack.
- The number in parentheses at the end of each line indicates the number of nodes that share the same stack frame.
- If you click any line in the stack dump, the pane on the right (labeled Common nodes) shows the list of nodes that share that stack frame. See Figure 9-7 on page 153.
- When you click one of the stack frames and then select Control → Run, the action is performed for all nodes that share that stack frame. A new Processor Status summary is displayed. If you again chose a Stack Traceback option, the running processors are halted and the stacks are refetched.
- You can hold down the Shift key and click several stack frames if you want to control all procedures that are at a range of stack frames.
- From the Filter menu option, you can select Group Selection → Create Filter to add a filter with the name that you specify in the Filter pull-down. When the box for your filter is highlighted, only the data for those processors is displayed in the upper-left window. You can create several filters if you want.
- Set Group Mode to Ungrouped or Ungrouped with Traceback to control one processor at a time.
9.2.6 Saving your information

To save the current contents of Traceback information about the upper-left pane, select **File → Save Traceback** to a file of your choice.

To gain more complete data, select **File → Take Snapshot™**. Notice that you then have two sessions to choose from on the Sessions menu. The original session is (MMCS), and the second one is (SNAP). The snapshot is exactly what the name implies, a picture of the debug session at a particular point. Notice that you cannot start or stop the processors from the snapshot session. You can choose **File → Save Snapshot** to save the snapshot to a file. If you are sending data to IBM for debug, Save Snapshot is a better choice than Save Traceback because the snapshot includes objdump data.

If you choose **File → Quit** and processors are halted, you are given an option to restart them before quitting.

9.2.7 Debugging live I/O node problems

It is possible to debug I/O nodes as well as compute nodes, but you normally want to avoid doing so. Collecting data causes the processor to be stopped, and stopping the I/O node processors can cause problems with your file system. In addition, the compute nodes are not able to communicate with the I/O nodes. If you want to debug an I/O node, you must specify the I/O node binary when you select **File → Attach** to block the window, and choose **Filter → Debug I/O Nodes**.
9.2.8 Debugging core files

To work with core files, select **File → Load Core**. In the window, specify the following information:

- The location of the compute node kernel binary or binaries
- The core files location
- The lowest and highest-numbered core files that you want to work with (The default is all available core files.)

Click the **Load Cores** button when you have specified the information.

The same Grouping Modes are available for core file debug as for live debug. Figure 9-8 shows an output example of the Condensed Stack Traceback options from a core file. Condensed mode is the easiest format to work with.

![Core file condensed stack trace](image)
Figure 9-9 shows the detailed version of the same trace.

The Survey option is less useful for core files because speed is not such a concern.
When you select a stack frame in the Traceback output (Figure 9-10), two additional pieces of information are displayed. The core files that share that stack frame are displayed in the Common nodes pane. The Location field under the Traceback pane displays the location of that function and the line number represented by the stack frame. If you select one of the core files in the Common nodes pane, the contents of that core file are displayed in the bottom pane.

![Figure 9-10  Core file common nodes](image)

### 9.2.9 The addr2line utility

The `addr2line` utility is a standard Linux program. You can find additional information about this utility in any Linux manual as well as at the following Web site:

http://www.linuxcommand.org/man_pages/addr2line1.html

The `addr2line` utility translates an address into file names and line numbers. Using an address and an executable, this utility uses the debugging information in the executable to provide information about the file name and line number. To take advantage of this utility, compile your program with the `-g` option. On the Blue Gene/P system, the core file is a plain text file that you can view with the vi editor.

You can use the Linux `addr2line` command on the front-end node and enter the address found in the core file and the `-g` executable. Then the utility points you to the source line where the problem occurred.

Example 9-6 on page 157 shows a core file and how to use the `addr2line` utility to identify potential problems in the code. In this particular case, the program was not compiled with the `-g` flag option because this was a production run. However, notice in Example 9-6 on page 157 that `addr2line` points to `malloc()`. This can be a hint that perhaps the amount of
memory is insufficient to run this particular calculation, or some other problems might be related to the usage of malloc() in the code.

Example 9-6 Using addr2line to identify potential problems in your code

vi core.0 and select the addresses between +++STACK and ---STACK and use them as input for addr2line
+++STACK
0x01342cb8
0x0134653c
0x0106e5f8
0x010841ec
0x0103946c
0x010af40c
0x010b5e44
0x01004fa0
0x010027cc
0x0100c028
0x0100133c
0x013227ec
0x01322a4c
0xffffffffc
---STACK

Run addr2line with your executable
 addr2line -e a.out
0x01342cb8
0x0134653c
0x0106e5f8
0x010841ec
0x0103946c
0x010af40c
0x010b5e44
0x01004fa0
0x010027cc
0x0100c028
0x0100133c
0x013227ec
0x01322a4c
modify.cpp:0
???:0
???:0
???:0
main.cpp:0
main.cpp:0
main.cpp:0
???:0
../csu/libc-start.c:231
../sysdeps/unix/sysv/linux/powerpc/libc-start.c:127
In this chapter, we provide details about the checkpoint and restart support provided by the Blue Gene/P system. The contents of this chapter reflect the information presented in IBM System Blue Gene Solution: Application Development, SG24-7179, but have been updated for the Blue Gene/P system.

Scientific and engineering applications tend to consume most of the compute cycles on high-performance computers. This is certainly the case on the Blue Gene/P system. Many of the simulations run for extended periods of time and checkpoint and restart capabilities are critical for fault recovery.

Checkpoint and restart capabilities are critical for fault recovery. If an application is running for a long period of time, you do not want it to fail after consuming many hours of compute cycles, losing all the calculations made up until the failure. By using checkpoint and restart, you can restart the application at the last checkpoint position, losing a much smaller slice of processing time. In addition, checkpoint and restart are helpful in cases where the given access to a Blue Gene/P system is in relatively small increments of time and you know that your application run will take longer than your allotted amount of processing time. With checkpoint and restart capabilities, you can execute your application in fragmented periods of time rather than an extended interval of time.

We discuss the following topics in this chapter:

- Checkpoint and restart
- Technical overview
- Checkpoint API
- Directory and file-naming conventions
- Restart
10.1 Checkpoint and restart

Checkpoint and restart are among the primary techniques for fault recovery. A special user-level checkpoint library has been developed for Blue Gene/P applications. Using this library, application programs can take a checkpoint of their program state at the appropriate stages. Then the program can be restarted later from the last successful checkpoint.

10.2 Technical overview

The checkpoint library is a user-level library that provides support for user-initiated checkpoints in parallel applications. The current implementation requires application developers to insert calls manually to checkpoint library functions at proper places in the application code. However, the restart is transparent to the application and requires only the user or system to set specific environment variables while launching the application.

The application is expected to make a call to the BGCheckpointInit() function at the beginning of the program, to initialize the checkpoint-related data structures and to carry out an automated restart when required. The application can then make calls to the BGCheckpoint() function to store a snapshot of the program state in stable storage (files on a disk). The current model assumes that when an application must take a checkpoint, all of the following points are true:

- All processes of the application make a call to the BGCheckpoint() function.
- When a process makes a call to BGCheckpoint(), no outstanding messages are in the network or buffers. That is, the recv that corresponds to all the send calls has occurred.
- After a process has made a call to BGCheckpoint(), other processes do not send messages to the process until their checkpoint is complete. Typically, applications are expected to place calls to BGCheckpoint() immediately after a barrier operation, such as MPI_Barrier, or after a collective operation, such as MPI_Allreduce, when no outstanding messages are in the Message Passing Interface (MPI) buffers and the network.

BGCheckpoint() can be called multiple times. Successive checkpoints are identified and distinguished by a checkpoint sequence number. A program state that corresponds to different checkpoints is stored in separate files. It is possible to safely delete the old checkpoint files after a newer checkpoint is complete.

The data that corresponds to the checkpoints is stored in a user-specified directory. A separate checkpoint file is made for each process. This checkpoint file contains header information and a dump of the process’s memory, including its data and stack segments, but excluding its text segment and read-only data. It also contains information that pertains to the input/output (I/O) state of the application, including open files and the current file positions.

For restart, the same job is launched again with the environment variables BG_CHKPTRESTARTSEQNO and BG_CHKPTDIRPATH set to the appropriate values. The BGCheckpointInit() function checks for these environment variables and, if specified, restarts the application from the desired checkpoint.
10.2.1 Input/output considerations

All the external I/O calls made from a program are shipped to the corresponding I/O node using a function-shipping procedure implemented in the compute node kernel.

The checkpoint library intercepts calls to the following main file I/O functions:

- open()
- close()
- read()
- write()
- lseek()

The function name `open()` is a weak alias that maps to the `_libc_open` function. The checkpoint library intercepts this call and provides its own implementation of `open()` that internally uses the `_libc_open` function.

The library maintains a file state table that stores the file name, current file position, and the mode of all the files that are currently open. The table also maintains a translation that translates the file descriptors used by the compute node kernel to another set of file descriptors to be used by the application. While taking a checkpoint, the file state table is also stored in the checkpoint file. Upon a restart, these tables are read. Also the corresponding files are opened in the required mode, and the file pointers are positioned at the desired locations as given in the checkpoint file.

The current design assumes that the programs either always read the file or write the files sequentially. A read followed by an overlapping write, or a write followed by an overlapping read, is not supported.

10.2.2 Signal considerations

Applications can register handlers for signals using the `signal()` function call. The checkpoint library intercepts calls to `signal()` and installs its own signal handler instead. It also updates a signal-state table that stores the address of the signal handler function (`sighandler`) registered for each signal (`signum`). When a signal is raised, the checkpoint signal handler calls the appropriate application handler given in the signal-state table.

While taking checkpoints, the signal-state table is also stored in the checkpoint file in its signal-state section. At the time of restart, the signal-state table is read, and the checkpoint signal handler is installed for all the signals listed in the signal-state table. The checkpoint handler calls the required application handlers when needed.

Signals during checkpoint

The application can potentially receive signals while the checkpoint is in progress. If the application signal handlers are called while a checkpoint is in progress, it can change the state of the memory being checkpointed. This can make the checkpoint inconsistent. Therefore, the signals arriving while a checkpoint is under progress must be handled carefully.

For certain signals, such as SIGKILL and SIGSTOP, the action is fixed, and the application terminates without much choice. The signals without any registered handler are simply ignored. For signals with installed handlers, the two choices are as follows:

- Deliver the signal immediately.
- Postpone the signal delivery until the checkpoint is complete.
All signals are classified into one of these two categories as shown in Table 10-1. If the signal must be delivered immediately, the memory state of the application might change, making the current checkpoint file inconsistent. Therefore, the current checkpoint must be aborted. The checkpoint routine periodically checks if a signal has been delivered since the current checkpoint began. In case a signal has been delivered, it aborts the current checkpoint and returns to the application.

For signals that are to be postponed, the checkpoint handler simply saves the signal information in a pending signal list. When the checkpoint is complete, the library calls application handlers for all the signals in the pending signal list. If more than one signal of the same type is raised while the checkpoint is in progress, the checkpoint library ensures that the handler registered by the application is called at least once. However, it does not guarantee in-order-delivery of signals.

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Signal type</th>
<th>Action to be taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGINT</td>
<td>Critical</td>
<td>Deliver</td>
</tr>
<tr>
<td>SIGXCPU</td>
<td>Critical</td>
<td>Deliver</td>
</tr>
<tr>
<td>SIGILL</td>
<td>Critical</td>
<td>Deliver</td>
</tr>
<tr>
<td>SIGABRT/SIGIOT</td>
<td>Critical</td>
<td>Deliver</td>
</tr>
<tr>
<td>SIGBUS</td>
<td>Critical</td>
<td>Deliver</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>Critical</td>
<td>Deliver</td>
</tr>
<tr>
<td>SIGSTP</td>
<td>Critical</td>
<td>Deliver</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>Critical</td>
<td>Deliver</td>
</tr>
<tr>
<td>SIGPIPE</td>
<td>Critical</td>
<td>Deliver</td>
</tr>
<tr>
<td>SIGSTP</td>
<td>Critical</td>
<td>Deliver</td>
</tr>
<tr>
<td>SIGSTKFLT</td>
<td>Critical</td>
<td>Deliver</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>Critical</td>
<td>Deliver</td>
</tr>
<tr>
<td>SIGHUP</td>
<td>Non-critical</td>
<td>Postpone</td>
</tr>
<tr>
<td>SIGALRM</td>
<td>Non-critical</td>
<td>Postpone</td>
</tr>
<tr>
<td>SIGUSR1</td>
<td>Non-critical</td>
<td>Postpone</td>
</tr>
<tr>
<td>SIGUSR2</td>
<td>Non-critical</td>
<td>Postpone</td>
</tr>
<tr>
<td>SIGTSTP</td>
<td>Non-critical</td>
<td>Postpone</td>
</tr>
<tr>
<td>SIGVTALRM</td>
<td>Non-critical</td>
<td>Postpone</td>
</tr>
<tr>
<td>SIGPROF</td>
<td>Non-critical</td>
<td>Postpone</td>
</tr>
<tr>
<td>SIGPOLL/SIGIO</td>
<td>Non-critical</td>
<td>Postpone</td>
</tr>
<tr>
<td>SIGSYS/SIGUNUSED</td>
<td>Non-critical</td>
<td>Postpone</td>
</tr>
<tr>
<td>SIGTRAP</td>
<td>Non-critical</td>
<td>Postpone</td>
</tr>
</tbody>
</table>
Signals during restart
The pending signal list is not stored in the checkpoint file. Therefore, if an application is restarted from a checkpoint, the handlers for pending signals received during the checkpoint are not called. If some signals are raised while the restart is in progress, they are ignored. The checkpoint signal handlers are installed only after the memory state, I/O state, and signal-state table have been restored. This ensures that, when the application signal handlers are called, they see a consistent memory and I/O state.

10.3 Checkpoint API

The checkpoint interface consists of the following items:

- A set of library functions used by the application developer to checkpoint enable the application
- A set of conventions used to name and store the checkpoint files
- A set of environment variables used to communicate with the application

In the following section, we describe each of these components in detail.

Note: Blue Gene/P supplied checkpoint and restart APIs are not supported for HTC applications.

10.3.1 Checkpoint library API

To ensure minimal overhead, the basic interface has been kept fairly simple. Ideally, a programmer must call only two functions, one at the time of initialization and the other at the places where the application must be checkpointed. Restart is done transparently using the environment variable BG_CHKPTRESTARTSEQNO specified at the time of job launch. Alternatively, an explicit restart API is also provided to the programmer to manually restart the application from a specified checkpoint. The remainder of this section describes the checkpoint API in detail.

void BGCheckpointInit(char * ckptDirPath)

BGCheckpointInit is a mandatory function that must be invoked at the beginning of the program. You use this function to initialize the data structures of the checkpoint library. In addition, you use this function for transparent restart of the application program.

The ckptDirPath parameter specifies the location of checkpoint files. If ckptDirPath is NULL, then the default checkpoint file location is assumed as explained in 10.4, “Directory and file-naming conventions” on page 165.

int BGCheckpoint()

BGCheckpoint takes a snapshot of the program state at the instant at which it is called. All the processes of the application must make a call to BGCheckpoint to take a consistent global checkpoint.

When a process makes a call to BGCheckpoint, no outstanding messages should be in the network or buffers. That is, the recv that corresponds to all the send calls should have occurred. In addition, after a process has made a call to BGCheckpoint, other processes must not send messages to the process until their call to BGCheckpoint is complete. Typically, applications are expected to place calls to BGCheckpoint immediately after a barrier operation,
such as MPI_Barrier, or after a collective operation, such as MPI_Allreduce, when no outstanding message is in the MPI buffers and the network.

The state that corresponds to each application process is stored in a separate file. The location of checkpoint files is specified by 

\texttt{ckptDirPath} in the call to \texttt{BGCheckpointInit}. If \texttt{ckptDirPath} is NULL, then the checkpoint file location is decided by the storage rules mentioned in 10.4, “Directory and file-naming conventions” on page 165.

\texttt{void BGCheckpointRestart(int restartSqNo)}

\texttt{BGCheckpointRestart} restarts the application from the checkpoint given by the argument \texttt{restartSqNo}. The directory where the checkpoint files are searched is specified by \texttt{ckptDirPath} in the call to \texttt{BGCheckpointInit}. If \texttt{ckptDirPath} is NULL, then the checkpoint file location is decided by the storage rules provided in 10.4, “Directory and file-naming conventions” on page 165.

An application developer does not need to explicitly invoke this function. \texttt{BGCheckpointInit} automatically invokes this function whenever an application is restarted. The environment variable \texttt{BG_CHKPTRESTARTSEQNO} is set to an appropriate value. If the \texttt{restartSqNo}, the environment variable \texttt{BG_CHKPTRESTARTSEQNO}, is zero, then the system picks up the most recent consistent checkpoint files. However, the function is available for use if the developer chooses to call it explicitly. The developer must know the implications of using this function.

\texttt{int BGCheckpointExcludeRegion(void *addr, size_t len)}

\texttt{BGCheckpointExcludeRegion} marks the specified region (\texttt{addr} to \texttt{addr + len - 1}) to be excluded from the program state, while a checkpoint is being taken. The state that corresponds to this region is not saved in the checkpoint file. Therefore, after restart the corresponding memory region in the application is not overwritten. You can use this facility to protect critical data that should not be restored at the time of restart such as personality and checkpoint data structures. An application programmer can also use this call to exclude a scratch data structure that does not have to be saved at checkpoint time.

\texttt{int BGAtCheckpoint((void *) function(void *arg), void *arg)}

\texttt{BGAtCheckpoint} registers the functions to be called just before taking the checkpoint. You can use this function to take some action at the time of checkpoint. For example, you can call this function to close all the communication states open at the time of checkpoint. The functions registered are called in the reverse order of their registration. The argument \texttt{arg} is passed to the function that is being called.

\texttt{int BGAtRestart((void *) function (void *arg), void *arg)}

\texttt{BGAtRestart} registers the functions to be called during restart after the program state has been restored, but before jumping to the appropriate position in the application code. The functions that are registered are called in the reverse order of their registration. You can use this function to resume or re-initialize functions or data structures at the time of restart. For example, in the symmetrical multiprocessing node mode (SMP node mode), the SMP must be re-initialized at the time of restart. The argument \texttt{arg} is passed to the function that is being called.

\texttt{int BGAtContinue((void *) function (void *arg), void *arg)}

\texttt{BGAtContinue} registers the functions to be called when continuing after a checkpoint. You can use this function to re-initialize or resume some functions or data structures that were closed or stopped at the time of checkpoint. The functions that are registered are called in the reverse order of their registration. The argument \texttt{arg} is passed to the function that is being called.
10.4 Directory and file-naming conventions

By default, all the checkpoint files are stored, and retrieved during restart, in the directory specified by \texttt{ckptDirPath} in the initial call to \texttt{BGCheckpointInit()}. If \texttt{ckptDirPath} is not specified (or is NULL), the directory is picked from the environment variable \texttt{BG_CHKPTDIRPATH}. This environment variable can be set by the job control system at the time of job launch to specify the default location of the checkpoint files. If this variable is not set, the Blue Gene/P system looks for a $(HOME)/checkpoint directory. Finally, if this directory is also not available, $(HOME)$ is used to store all checkpoint files.

The checkpoint files are automatically created and named with the following convention:

\<\texttt{ckptDirPath} >/ckpt.<xxx-yyy-zzz><seqNo>

Note the following explanation:

- \<\texttt{ckptDirPath} >: Name of the executable, for example, sweep3d or mg.W.2
- \<xxx-yyy-zzz>: Three-dimensional torus coordinates of the process
- \<seqNo>: The checkpoint sequence number

The checkpoint sequence number starts at one and is incremented after every successful checkpoint.

10.5 Restart

A transparent restart mechanism is provided through the use of the \texttt{BGCheckpointInit()} function and the \texttt{BG_CHKPTRESTARTSEQNO} environment variable. Upon startup, an application is expected to make a call to \texttt{BGCheckpointInit()}. The \texttt{BGCheckpointInit()} function initializes the checkpoint library data structures.

Moreover the \texttt{BGCheckpointInit()} function checks for the environment variable \texttt{BG_CHKPTRESTARTSEQNO}. If the variable is not set, a job launch is assumed, and the function returns normally. In case the environment variable is set to zero, the individual processes restart from their individual latest consistent global checkpoint. If the variable is set to a positive integer, the application is started from the specified checkpoint sequence number.

10.5.1 Determining the latest consistent global checkpoint

Existence of a checkpoint file does not guarantee consistency of the checkpoint. An application might have crashed before completely writing the program state to the file. We have changed this by adding a checkpoint write complete flag in the header of the checkpoint file. As soon as the checkpoint file is opened for writing, this flag is set to zero and written to the checkpoint file. When complete checkpoint data is written to the file, the flag is set to one indicating the consistency of the checkpoint data. The job launch subsystem can use this flag to verify the consistency of checkpoint files and delete inconsistent checkpoint files.

During a checkpoint, some of the processes can crash, while others might complete. This can create consistent checkpoint files for some processes and inconsistent or non-existent checkpoint files for other processes. The latest consistent global checkpoint is determined by the latest checkpoint for which all the processes have consistent checkpoint files.

It is the responsibility of the job launch subsystem to make sure that \texttt{BG_CHKPTRESTARTSEQNO} corresponds to a consistent global checkpoint. In case \texttt{BG_CHKPTRESTARTSEQNO} is set to zero, the job launch subsystem must make sure that files with the highest checkpoint sequence number correspond to a consistent global checkpoint. The behavior of the checkpoint library is undefined if \texttt{BG_CHKPTRESTARTSEQNO} does not correspond to a global consistent checkpoint.
10.5.2 Checkpoint and restart functionality

It is often desirable to enable or disable the checkpoint functionality at the time of job launch. Application developers are not required to provide two versions of their programs: one with checkpoint enabled and another with checkpoint disabled. We have used environment variables to transparently enable and disable the checkpoint and restart functionality.

The checkpoint library calls check for the environment variable BG_CHKPTENABLED. The checkpoint functionality is invoked only if this environment variable is set to a value of 1. Table 10-2 summarizes the checkpoint-related function calls.

Table 10-2  Checkpoint and restart APIs

<table>
<thead>
<tr>
<th>Function name</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>BGCheckpointInit(char *ckptDirPath);</td>
<td>Sets the checkpoint directory to ckptDirPath. Initializes the checkpoint library data structures. Carries out restart if environment variable BG_CHKPTRESTARTSEQNO is set.</td>
</tr>
<tr>
<td>BGCheckpoint();</td>
<td>Takes a checkpoint. Stores the program state in the checkpoint directory.</td>
</tr>
<tr>
<td>BGCheckpointRestart(int rstartSqNo);</td>
<td>Carries out an explicit restart from the specified sequence number.</td>
</tr>
<tr>
<td>BGCheckpointExcludeRegion (void *addr, size_t len);</td>
<td>Excludes the specified region from the checkpoint state.</td>
</tr>
</tbody>
</table>

Table 10-3 summarizes the environment variables.

Table 10-3  Checkpoint and restart environment variables

<table>
<thead>
<tr>
<th>Environment variables</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>BG_CHKPTENABLED</td>
<td>Is set (to 1) if checkpoints are desired; otherwise, it is not specified.</td>
</tr>
<tr>
<td>BG_CHKPTDIRPATH</td>
<td>Default path to keep checkpoint files.</td>
</tr>
<tr>
<td>BG_CHKPTRESTARTSEQNO</td>
<td>Set to a desired checkpoint sequence number from where a user wants the application to restart. If set to zero, each process restarts from its individual latest consistent checkpoint. This option must not be specified, if no restart is desired.</td>
</tr>
</tbody>
</table>

The most common environment variable settings are:

- `BG_CHKPTENABLED=1`
- `BG_CHKPTDIRPATH= checkpoint directory`
- `BG_CHKPTRESTARTSEQNO=0`

A combination of `BG_CHKPTENABLED` and `BG_CHKPTRESTARTSEQNO` (as in Table 10-3) automatically signifies that after restart, further checkpoints are taken. A developer can restart an application but disable further checkpoints by simply unsetting (removing altogether) the `BG_CHKPTENABLED` variable.
mpirun

mpirun is a software utility for launching, monitoring, and controlling programs (applications) that run on the BlueGene/P system. mpirun on the Blue Gene/P system serves the same function as on the Blue Gene/L system.

The name mpirun comes from Message Passing Interface (MPI) because its primary use is to launch parallel jobs. mpirun can be used as a standalone program by providing parameters either directly through a command line or from environmental variable arguments, or indirectly through the framework of a scheduler that submits the job on the user's behalf. In the former case, mpirun can be invoked as a shell command. It allows the user to interact with the running job via the job's standard input, standard output, and standard error. mpirun acts as a shadow of the actual Blue Gene job by monitoring its status, as well as providing access to standard input, output, and errors. After the job has terminated, mpirun terminates as well. If the user wants to prematurely end the job before it has terminated, mpirun provides a mechanism to do so explicitly or via a timeout period.

mpirun provides the capability to debug the job. In this chapter, we describe the standalone interactive use of mpirun. We also provide a brief overview of mpirun on the Blue Gene/P system. In addition, we define a list of APIs that allow interaction with the mpirun program. These APIs are used by applications, such as external resource managers, that want to programmatically invoke jobs via mpirun.

We address the following topics in this chapter and provide examples:
- mpirun implementation on Blue Gene/P
- mpirun setup
- Invoking mpirun
- Environmental variables
- Return codes
- mpirun APIs
11.1 mpirun implementation on Blue Gene/P

mpirun accepts a rich set of parameters, following the philosophy of the Blue Gene/L system, that describe its behavior prior to submitting the application for execution on the compute nodes and during execution of the application. These parameters can be divided into three groups. The first group identifies resources that are required to run the application. The second group identifies the application (binary) to execute and the environment settings for that particular run or executable. The third group identifies the level of verbosity that mpirun prints to STDOUT or STDERR.

Although mpirun has kept all the functionality that is available on the Blue Gene/L system, its implementation on the Blue Gene/P system differs in the following ways:

- The rsh/ssh mechanism has been eliminated for starting the back-end process due to security concerns of allowing users access to the service node. In the Blue Gene/P system, this is replaced with a daemon process that runs on the service node whose purpose is to handle connections from front-end mpirun processes and fork back-end mpirun processes as illustrated in Figure 11-1.

![Figure 11-1 mpirun interacting with the rest of the control system on the Blue Gene/P system](image)

- After mpirun_be is forked, the sequence of events for booting partitions, starting jobs, and collecting stdout/stderr is similar to the use of mpirun on the Blue Gene/L system.
- The freepartition program has been integrated as an option in mpirun for the Blue Gene/P system as illustrated in Example 11-1.

**Example 11-1 mpirun example with -nofree option**

```
mpirun -partition N01_32_1 -np 32 -cwd /bgusr/cpsosa -exe a.out -nofree
```
Example 11-2 shows how the free option is now used as part of mpirun on the Blue Gene/P system.

**Example 11-2  mpirun example with -free option**

```
[descartes:/bgusr/cpsosa/red/example.5_1] mpirun -partition N01_32_1 -free wait -verbose 1
<Jul 06 15:10:48.401421> FE_MPI (Info) : Invoking free partition
<Jul 06 15:10:48.414677> FE_MPI (Info) : freePartition() - connected to mpirun server at spinoza
<Jul 06 15:10:48.414768> FE_MPI (Info) : freePartition() - sent free partition request
<Jul 06 15:11:19.202335> FE_MPI (Info) : freePartition() - partition N01_32_1 was freed successfully
<Jul 06 15:11:19.202746> FE_MPI (Info) : ==   FE completed   ==
<Jul 06 15:11:19.202790> FE_MPI (Info) : == Exit status:   0 ==
```

- Also new in mpirun for the Blue Gene/P system is the support for multiple program, multiple data (MPMD) style jobs where a different executable, arguments, environment, and current working directory can be supplied for a single job on a processor set (pset) basis. For example, with this capability, a user can run four different executables on a partition with four psets.

This capability is handled by a new tool called mpiexec, which should not be confused with the mpiexec style of submitting a Single Program Multiple Data (SPMD) parallel MPI job.

### 11.1.1 mpiexec

mpiexec is the method for launching and interacting with parallel Multiple Program Multiple Data (MPMD) jobs on Blue Gene/P. It is very similar to mpirun with the only exception being the arguments supported by mpiexec are slightly different.

#### Unsupported parameters

The following parameters are not supported by mpiexec because their use is ambiguous for MPMD jobs. The corresponding environmental variables are shown to the right of the argument name.

**Table 11-1 Unsupported parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Environmental variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>-partition</td>
<td>MPIRUN_PARTITION</td>
</tr>
<tr>
<td>-exe</td>
<td>MPIRUN_CWD MPIRUN_WDIR</td>
</tr>
<tr>
<td>-env</td>
<td>MPIRUN_ENV</td>
</tr>
<tr>
<td>-exp_env</td>
<td>MPIRUN_EXP_ENV</td>
</tr>
<tr>
<td>-env_all</td>
<td>MPIRUN_EXP_ENV_ALL</td>
</tr>
<tr>
<td>-mapfile</td>
<td>MPIRUN_ARGS</td>
</tr>
<tr>
<td>-args</td>
<td>MPIRUN_ARGS</td>
</tr>
</tbody>
</table>

#### New parameters

The only parameter supported by mpiexec that is not supported by mpirun is the -configfile argument; see “Examples” on page 170 for sample usage.

- configfile MPIRUN_MPMD_CONFIGFILE
Limitations
Due to some underlying designs in the Blue Gene/P software stack, when using MPMD, the following limitations are applicable:

- A pset is the smallest granularity for each executable, though one executable can span multiple pssets.
- You must use every compute node of each pset; specifically different -np values are not supported.
- The job node mode (SMP, DUAL, VNM) must be uniform across all psets.
- MPI using DMA is not supported.
- MPI one-sided operations are not supported.
- Global arrays and ARMCI are not supported.

Examples
Example 11-3 illustrates running /bin/hostname on a single 32-node pset, helloworld.sh on another 32-node pset, and goodbyeworld.sh on two 32-node psets. The partition bar consists of 128 nodes, with 4 I/O nodes.

Example 11-3  mpiexe example

```
mpiexec -host dd2sys1fen3 -partition bar : -n 32 -wdir /bgusr/hello /bin/hostname :
-n 32 -wdir /bgusr/goodbye /bglhome/helloworld.sh : -n 64 -wdir 
/bgusr/samjmill/temp  
/bglhome/goodbyeworld.sh
```

11.2 mpirun setup

`mpirun` does not require setup from a user point of view. However, on the service node, `mpirun` requires slightly more setup for a system administrator. We have classified the setup of `mpirun` as the following types:

- User setup
- System administrator setup

11.2.1 User setup

Some setup has changed for Blue Gene/P as compared to `mpirun` for the Blue Gene/L system. The following changes are among those for user setup:

- It is not required to set up .rhosts or ssh-agent.
- It is not required to set up .bashrc, .tcshrc, or .profile to include BRIDGE_CONFIG_FILE or DB_PROPERTY environmentals.
- The `freepartition` program is now an option in `mpirun`.
- The -backend option is no longer available.

Due to the removal or the ssh/rsh mechanism to start a back-end `mpirun` process, users no longer are required to create an .rhosts file in their home directory for `mpirun` to work properly.
11.2.2 System administrator setup

System administrators can change the following configuration files for the `mpirun` daemon (`mpirund`):

- **db.properties**: Contains information about the DB2 database
- **bridge.config**: Contains locations of the default I/O node and compute node images when allocating partitions
- **mpirun.cfg**: Contains the shared secret that is used for challenge authentication between `mpirun` and `mpirund`

**Database properties and Bridge configuration files**

The location of the database properties and bridge configuration files can be changed by passing the appropriate arguments to bgpmaster when starting `mpirund`. The `mpirun` daemon then passes these locations to each `mpirun_be` forked. Example 11-4 shows a sample Bridge configuration file.

```plaintext
Example 11-4   Sample Bridge configuration file

BGP_MACHINE_SN     BGP
BGP_MLOADER_IMAGE  /bgsys/drivers/ppcfloor/boot/uloader
BGP_CNLOAD_IMAGE  /bgsys/drivers/ppcfloor/boot/cns,/bgsys/drivers/ppcfloor/boot/cnk
BGP_IOLOAD_IMAGE  /bgsys/drivers/ppcfloor/boot/cns,/bgsys/drivers/ppcfloor/boot/linux,/bgsys/drivers/ppcfloor/boot/ramdisk
BGP_BOOT_OPTIONS
BGP_DEFAULT_CWD   $PWD
```

*BGP_DEFAULT_CWD* is used for `mpirun` jobs when a user does not give the `-cwd` argument or one of its environmental values. This value can be optionally changed to something more site specific, such as `/bgp/users`, `/gpfs/`, and so on. The special keyword `$PWD` is expanded to the user's current working directory from where the user executed `mpirun`.

**Challenge protocol**

The challenge protocol, which is used to authenticate the `mpirun` front end when connecting to the `mpirun` daemon on the service node, is a challenge/response protocol. It uses a shared secret to create a hash of a random number, thereby verifying that the `mpirun` front end has access to the secret.

To protect the secret, the challenge protocol is stored in a configuration file that is accessible only by the bgpadmin user on the service node and by a special `mpirun` user on the front-end nodes. The front-end `mpirun` binary has its setuid flag enabled so that it can change its uid to match the `mpirun` user and read the configuration file to access the secret.
11.3 Invoking mpirun

The first method of using mpirun is to specify the parameters explicitly as shown in the following example:

```plaintext
mpirun [options]
```

Here is a practical example of using mpirun:

```plaintext
mpirun -partition R00-M0 -mode VN -cwd /bgusr/tmp a.out --timeout 50
```

Alternatively, you can use the mpiexec style where the executable and arguments are implicit, as shown in the following example (see 11.1.1, “mpiexec” on page 169):

```plaintext
mpirun [options] binary [arg1 arg2 ... argn]
```

Here is a practical example of using mpiexec:

```plaintext
mpirun -partition R00-M0 -mode VN -cwd /bgusr/tmp -exe a.out --args "--timeout 50"
```

Specifying parameters

You can specify parameters for the mpirun program in the following different ways:

- Command-line arguments
- Environmental variables
- Scheduler interface plug-in

In general, users normally use the command-line arguments and the environmental variables. Certain schedulers use the scheduler interface plug-in to restrict or enable mpirun features according to their environment. For example, the scheduler might have a policy where interactive job submission with mpirun can be allowed only during certain hours of the day.

Command-line arguments

The mpirun arguments consist of the following categories:

- Job control
- Block control
- Output
- Other

Job control arguments

Table 11-2 lists the job control arguments.

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-args &quot;program args&quot;</td>
<td>Passes &quot;program args&quot; to the BlueGene job on the compute nodes.</td>
</tr>
<tr>
<td>-env &quot;ENVVAR=value&quot;</td>
<td>Sets an environment variable in the environment of the job on the compute nodes.</td>
</tr>
<tr>
<td>-exp_env ENVVAR</td>
<td>Exports an environment variable in the current environment of mpirun to the job on the compute nodes.</td>
</tr>
<tr>
<td>-env_all</td>
<td>Exports all environment variables in the current environment of mpirun to the job on the compute nodes.</td>
</tr>
<tr>
<td>-np &lt;n&gt;</td>
<td>Creates exactly n MPI ranks for the job. Aliases are -nodes and -n.</td>
</tr>
</tbody>
</table>
Block control options

`mpirun` can also allocate partitions and create new partitions if necessary. Use the following general rules for block control:

- If `mpirun` is told to use a pre-existing partition and it is already booted, `mpirun` uses it as is without trying to boot it again.
- If `mpirun` creates a partition or is told to use a pre-existing partition that is not already allocated, `mpirun` allocates the partition.
- If `mpirun` allocates a partition, it deallocates the partition when it is done.

Table 11-3 summarizes the options that modify this behavior.

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-mode &lt;SMP or DUAL or VN&gt;</code></td>
<td>Specifies the mode in which the job will run. Choices are SMP (1 rank, 4 threads), DUAL (2 ranks, 2 threads each), or virtual node mode (4 ranks, 1 thread each).</td>
</tr>
<tr>
<td><code>-exe &lt;executable&gt;</code></td>
<td>Specifies the full path to the executable to run on the compute nodes. The path is specified as seen by the I/O and compute nodes.</td>
</tr>
<tr>
<td><code>-cwd &lt;path&gt;</code></td>
<td>Specifies the full path to use as the current working directory on the compute nodes. The path is specified as seen by the I/O and compute nodes.</td>
</tr>
<tr>
<td><code>-mapfile &lt;mapfile&gt;</code></td>
<td>Specifies an alternative MPI topology. The mapfile path must be fully qualified as seen by the I/O and compute nodes.</td>
</tr>
<tr>
<td><code>-timeout &lt;n&gt;</code></td>
<td>Timeout after $n$ seconds. <code>mpirun</code> monitors the job and terminates it if the job runs longer than the time specified. The default is never to timeout.</td>
</tr>
<tr>
<td><code>-partition &lt;block&gt;</code></td>
<td>Specifies a predefined block to use.</td>
</tr>
<tr>
<td><code>-nofree</code></td>
<td>If <code>mpirun</code> booted the block, it does not deallocate the block when the job is done. This is useful for when you want to run a string of jobs back-to-back on a block but do not want <code>mpirun</code> to boot and deallocate the block each time (which happens if you had not booted the block first using the console.) When your string of jobs is finally done, use the <code>freepartition</code> command to deallocate the block.</td>
</tr>
<tr>
<td>`-free &lt;wait</td>
<td>nowait&gt;`</td>
</tr>
<tr>
<td><code>-noallocate</code></td>
<td>This option is more interesting for job schedulers. It tells <code>mpirun</code> not to use a block that is not already booted.</td>
</tr>
<tr>
<td><code>-shape &lt;XxYxZ&gt;</code></td>
<td>Specifies a hardware configuration to use. The dimensions are in the compute nodes. If hardware matching is found, a new partition is created and booted. Implies that <code>-partition</code> is not specified.</td>
</tr>
<tr>
<td><code>-psets_per_bp &lt;n&gt;</code></td>
<td>Specifies the I/O node to compute node ratio. The default is to use the best possible ratio of I/O nodes to compute nodes. Specifying a higher number of I/O nodes than what is available results in an error.</td>
</tr>
</tbody>
</table>

a. For additional information about mapping, see Appendix E, “Mapping” on page 303.
Output options
The output options (in Table 11-4) control information that is sent to STDIN, STDOUT, and STDERR.

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-connect &lt;MESH</td>
<td>TORUS&gt;</td>
</tr>
<tr>
<td>-reboot</td>
<td>Reboots all the compute nodes of an already booted partition that is specified with -partition before running the job. If the partition is in any other state, this is an error.</td>
</tr>
<tr>
<td>-boot_options &lt;options&gt;</td>
<td>Specifies boot options to use when booting a freshly created partition.</td>
</tr>
</tbody>
</table>

Other options
Table 11-5 lists other options. These options provide general information about selected software and hardware features.

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-verbose [0-4]</td>
<td>Sets the verbosity level. The default is 0, which means that mpirun does not output any status or diagnostic messages unless a severe error occurs. If you are curious about what is happening, try levels 1 or 2. All mpirun generated status and error messages appear on STDERR.</td>
</tr>
<tr>
<td>-label</td>
<td>Use this option to have mpirun label the source of each line of output. The source is the MPI rank, and stderr or stdout from which the output originated.</td>
</tr>
<tr>
<td>-enable tty_reporting</td>
<td>By default, mpirun tells the control system and the C run time on the compute nodes that STDIN, STDOUT, and STDERR are tied to TTY type devices. While semantically correct for the BlueGene system, this prevents blocked I/O to these file descriptors, which can slow down operations. If you use this option, mpirun senses whether these file descriptors are tied to TTYs and reports the results accurately to the control system.</td>
</tr>
<tr>
<td>-strace &lt;all</td>
<td>none</td>
</tr>
</tbody>
</table>
11.4 Environmental variables

An alternative way to control `mpirun` execution is to use environmental variables. Most command-line options for `mpirun` can be specified using an environment variable. The variables are useful for options that are used in production runs. If you do need to alter the option, you can modify it on the command line to override the environment variable. Table 11-6 summarizes all the environmental variables. The variables must be defined before execution of `mpirun` starts.

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-nw</td>
<td>Reports <code>mpirun</code>-generated return code instead of an application-generated return code. Useful only for debugging <code>mpirun</code>.</td>
</tr>
<tr>
<td>-only_test_protocol</td>
<td>Simulates a job without using any hardware or talking to the control system. It is useful for making sure that <code>mpirun</code> can start <code>mpirun_be</code> correctly.</td>
</tr>
</tbody>
</table>

Table 11-6 Environmental variables

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Environmental variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>-partition</td>
<td><code>MPIRUN_PARTITION</code></td>
</tr>
<tr>
<td>-nodes</td>
<td><code>MPIRUN_NODES</code> <code>MPIRUN_N</code> <code>MPIRUN_NP</code></td>
</tr>
<tr>
<td>-mode</td>
<td><code>MPIRUN_MODE</code></td>
</tr>
<tr>
<td>-exe</td>
<td><code>MPIRUN_EXE</code></td>
</tr>
<tr>
<td>-cwd</td>
<td><code>MPIRUN_CWD</code> <code>MPIRUN_WDIR</code></td>
</tr>
<tr>
<td>-host</td>
<td><code>MMCS_SERVER_IP</code> <code>_MPIRUN_SERVER_HOSTNAME</code></td>
</tr>
<tr>
<td>-port</td>
<td><code>MPIRUN_SERVER_PORT</code></td>
</tr>
<tr>
<td>-env</td>
<td><code>MPIRUN_ENV</code></td>
</tr>
<tr>
<td>-exp_env</td>
<td><code>MPIRUN_EXP_ENV</code></td>
</tr>
<tr>
<td>-env_all</td>
<td><code>MPIRUN_EXP_ENV_ALL</code></td>
</tr>
<tr>
<td>-mapfile</td>
<td><code>MPIRUN_MAPFILE</code></td>
</tr>
<tr>
<td>-args</td>
<td><code>MPIRUN_ARGS</code></td>
</tr>
<tr>
<td>-timeout</td>
<td><code>MPIRUN_TIMEOUT</code></td>
</tr>
<tr>
<td>-start_gdbserver</td>
<td><code>MPIRUN_START_GDBSERVER</code></td>
</tr>
<tr>
<td>-label</td>
<td><code>MPIRUN_LABEL</code></td>
</tr>
<tr>
<td>-nw</td>
<td><code>MPIRUN_NW</code></td>
</tr>
<tr>
<td>-nofree</td>
<td><code>MPIRUN_NOFREE</code></td>
</tr>
<tr>
<td>-noallocate</td>
<td><code>MPIRUN_NOALLOCATE</code></td>
</tr>
<tr>
<td>-reboot</td>
<td><code>MPIRUN_REBOOT</code></td>
</tr>
<tr>
<td>-boot_options</td>
<td><code>MPIRUN_BOOT_OPTIONS</code> <code>MPIRUN_KERNEL_OPTIONS</code></td>
</tr>
<tr>
<td>-verbose</td>
<td><code>MPIRUN_VERBOSE</code></td>
</tr>
</tbody>
</table>

Chapter 11. mpirun 175
11.5 Return codes

If `mpirun` fails for any reason, such as a bug, boot failure, or job failure, it returns a return code to your shell if you supply the `-nw` argument. If you omit the `-nw` argument, it returns the job’s return code if it is present in the job table. Table 11-7 lists the possible error codes.

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Environmental variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>-only_test_protocol</td>
<td>MPIRUN_ONLY_TEST_PROTOCOL</td>
</tr>
<tr>
<td>-shape</td>
<td>MPIRUN_SHAPE</td>
</tr>
<tr>
<td>-psets_per_bp</td>
<td>MPIRUN_PSETS_PER_BP</td>
</tr>
<tr>
<td>-connect</td>
<td>MPIRUN_CONNECTION</td>
</tr>
<tr>
<td>-enable_tty_reporting</td>
<td>MPIRUN_ENABLE_TTY_REPORTING</td>
</tr>
<tr>
<td>-config</td>
<td>MPIRUN_CONFIG_FILE</td>
</tr>
</tbody>
</table>

### Table 11-7 Return codes

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>OK; successful</td>
</tr>
<tr>
<td>10</td>
<td>Communication error</td>
</tr>
<tr>
<td>11</td>
<td>Version handshake failed</td>
</tr>
<tr>
<td>12</td>
<td>Front-end initialization failed</td>
</tr>
<tr>
<td>13</td>
<td>Failed to execute back-end <code>mpirun</code> on service node</td>
</tr>
<tr>
<td>14</td>
<td>Back-end initialization failed</td>
</tr>
<tr>
<td>15</td>
<td>Failed to locate db.properties file</td>
</tr>
<tr>
<td>16</td>
<td>Failed to get the machine serial number (bridge configuration file not found?)</td>
</tr>
<tr>
<td>17</td>
<td>Execution interrupted by message from the front end</td>
</tr>
<tr>
<td>18</td>
<td>Failed to prepare the partition</td>
</tr>
<tr>
<td>19</td>
<td>Failed to initialize allocator</td>
</tr>
<tr>
<td>20</td>
<td>Partition name already exists</td>
</tr>
<tr>
<td>21</td>
<td>No free space left to allocate partition for this job</td>
</tr>
<tr>
<td>22</td>
<td>Failed to allocate partition</td>
</tr>
<tr>
<td>23</td>
<td>Failed to allocate a partition; job has illegal requirements</td>
</tr>
<tr>
<td>24</td>
<td>Specified partition does not exist</td>
</tr>
<tr>
<td>25</td>
<td>Failed to get a partition state</td>
</tr>
<tr>
<td>26</td>
<td>Specified partition is in an incompatible state</td>
</tr>
<tr>
<td>27</td>
<td>Specified partition is not ready</td>
</tr>
<tr>
<td>28</td>
<td>Failed to get a partition owner</td>
</tr>
<tr>
<td>29</td>
<td>Failed to set a partition owner</td>
</tr>
<tr>
<td>Return code</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>30</td>
<td>Failed while checking to see if the partition is busy</td>
</tr>
<tr>
<td>31</td>
<td>Partition is occupied by another job</td>
</tr>
<tr>
<td>32</td>
<td>Failed while checking to see if the user is in the partition's user list</td>
</tr>
<tr>
<td>33</td>
<td>A user does not have permission to run the job on the specified partition</td>
</tr>
<tr>
<td>34</td>
<td>Failed while examining the specified partition</td>
</tr>
<tr>
<td>35</td>
<td>Failed while setting kernel options; the rm_modify_partition() API failed</td>
</tr>
<tr>
<td>36</td>
<td>Kernel options were specified but the partition is not in a FREE state</td>
</tr>
<tr>
<td>37</td>
<td>Failed to boot the partition</td>
</tr>
<tr>
<td>38</td>
<td>Failed to reboot the partition</td>
</tr>
<tr>
<td>39</td>
<td>Failed to get the number of psets in the partition</td>
</tr>
<tr>
<td>40</td>
<td>Failed to create MPMD configuration file on the service node</td>
</tr>
<tr>
<td>41</td>
<td>Found a zero-length line while writing to the MPMD configuration file</td>
</tr>
<tr>
<td>42</td>
<td>Failed to write a line to the MPMD configuration file</td>
</tr>
<tr>
<td>43</td>
<td>Failed to validate the MPMD configuration file</td>
</tr>
<tr>
<td>44</td>
<td>Failed to add the new job to the database</td>
</tr>
<tr>
<td>45</td>
<td>Failed to get an ID for the new job</td>
</tr>
<tr>
<td>46</td>
<td>Failed to start the job</td>
</tr>
<tr>
<td>47</td>
<td>An error occurred while mpirun was waiting for the job to terminate</td>
</tr>
<tr>
<td>48</td>
<td>Job timed out</td>
</tr>
<tr>
<td>49</td>
<td>The job was moved to the history table before it terminated</td>
</tr>
<tr>
<td>50</td>
<td>Job execution failed; job switched to an error state</td>
</tr>
<tr>
<td>51</td>
<td>Job execution interrupted; job queued</td>
</tr>
<tr>
<td>52</td>
<td>Failed to get a job exit status</td>
</tr>
<tr>
<td>53</td>
<td>Failed to get a job error text</td>
</tr>
<tr>
<td>54</td>
<td>Executable path for the debugger server is not specified</td>
</tr>
<tr>
<td>55</td>
<td>Failed to set debug information; unable to attach the debugger</td>
</tr>
<tr>
<td>56</td>
<td>Failed to get proctable; unable to attach the debugger</td>
</tr>
<tr>
<td>57</td>
<td>Failed while attaching to the job; unable to attach the debugger</td>
</tr>
<tr>
<td>58</td>
<td>Failed debugging job; unable to attach the debugger</td>
</tr>
<tr>
<td>59</td>
<td>Failed to begin a job</td>
</tr>
<tr>
<td>60</td>
<td>Failed to load a job</td>
</tr>
<tr>
<td>61</td>
<td>Failed to wait for job to load</td>
</tr>
<tr>
<td>62</td>
<td>Failed to clean up a job, partition, or both</td>
</tr>
<tr>
<td>63</td>
<td>Failed to cancel a job</td>
</tr>
</tbody>
</table>
11.6 Examples

In this section, we present various examples of `mpirun` commands.

Display information

Example 11-5 shows how to display information using the -h flag.

Example 11-5 Invoking mpirun -h or -help to list all the options available

```
[descartes:/bgusr/cpsosa] mpirun -h
Usage:
  mpirun [options]
  or
  mpirun [options] binary [arg1 arg2 ... argn]
```

Options:
- `-h` Provides this extended help information; can also use -help
- `-version` Display version information
- `-partition <partition_id>` ID of the partition to run the job on
- `-np <compute_nodes>` The number of compute nodes to use for the job
- `-mode <SMP|DUAL|VN>` Execution mode, either SMP, DUAL, or virtual node mode; the default is SMP
- `-exe <binary>` Full path to the binary to execute
- `-cwd <path>` Current working directory of the job, as seen by the compute nodes; can also use -wdir
- `-host <service_node_host>` Host name of the service node
- `-port <service_node_port>` Port of the `mpirun` server on the service node
- `-env <env=val>` Environment variable that should be set
- `-exp_env <env vars>` Environment variable in the current environment to export
- `-env_all` Export all current environment variables to the job environment
- `-mapfile <mapfile|mapping>` Mapfile contains a user specified MPI topology; mapping is a permutation of XYZT
- `-args "<arguments">` Arguments to pass to the job; must be enclosed in double quotation marks
- `-timeout <seconds>` The limit of the job execution time
- `-start_gdbserver <path>` Start gdbserver for the job; must specify the path to gdbserver
- `-label` Add labels (STDOUT, STDERR, and MPI rank) to the job output
- `-nw` Return `mpirun` job cycle status instead of the job exit status
- `-nofree` Do not deallocate the partition if `mpirun` allocated it

<table>
<thead>
<tr>
<th>Return code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>Failed to destroy a partition</td>
</tr>
<tr>
<td>65</td>
<td>Failed to remove a partition</td>
</tr>
<tr>
<td>66</td>
<td>Failed to reset boot options; the <code>rm_modify_partition()</code> API failed</td>
</tr>
<tr>
<td>67</td>
<td>One or more threads died</td>
</tr>
<tr>
<td>68</td>
<td>Unexpected message</td>
</tr>
<tr>
<td>69</td>
<td>Failed to dequeue control message</td>
</tr>
<tr>
<td>70</td>
<td>Out of memory</td>
</tr>
<tr>
<td>71</td>
<td>Execution interrupted by signal</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-free &lt;wait</td>
<td>nowait&gt;</td>
</tr>
<tr>
<td>-noallocate</td>
<td>Do not allocate the partition; the job will only start</td>
</tr>
<tr>
<td></td>
<td>if the partition was already INITIALIZED or CONFIGURING</td>
</tr>
<tr>
<td>-reboot</td>
<td>Reboot all compute nodes of the specified partition before</td>
</tr>
<tr>
<td></td>
<td>running the job; the partition must be INITIALIZED prior to rebooting</td>
</tr>
<tr>
<td>-backend</td>
<td>Use a specified mpirun backend binary on the service node</td>
</tr>
<tr>
<td>-boot_options &lt;options&gt;</td>
<td>Low-level options used when booting a partition</td>
</tr>
<tr>
<td>-verbose &lt;0</td>
<td>1</td>
</tr>
<tr>
<td>-trace &lt;0-7&gt;</td>
<td>Trace level; output is sent to a file in the current working directory;</td>
</tr>
<tr>
<td></td>
<td>default level is 0</td>
</tr>
<tr>
<td>-only_test_protocol</td>
<td>Test the mpirun frontend to backend communication;</td>
</tr>
<tr>
<td></td>
<td>no job will be run</td>
</tr>
<tr>
<td>-strace &lt;all</td>
<td>none</td>
</tr>
<tr>
<td>-shape &lt;XxYxZ&gt;</td>
<td>Shape of job in XxYxZ format; if not specified, you must use</td>
</tr>
<tr>
<td></td>
<td>-partition or -np</td>
</tr>
<tr>
<td>-psets_per_bp &lt;n&gt;</td>
<td>Number of psets per base partition required in the partition</td>
</tr>
<tr>
<td>-connect &lt;TORUS</td>
<td>MESH&gt;</td>
</tr>
<tr>
<td>-enable_tty_reporting</td>
<td>Correctly report tty status to the control system</td>
</tr>
<tr>
<td>-config &lt;path&gt;</td>
<td>Specify mpirun config file path</td>
</tr>
</tbody>
</table>

Creating a partition dynamically

In Example 11-6, a user requests a number (-np) of compute nodes desired for the job. The allocator API searches the machine for free resources and boots the temporary partition if enough resources are found. Upon job completion, mpirun deallocates the partition if the user has not specified -nofree.

Example 11-6  Dynamic allocation
```
dd2sys1fen3:~/bgp/control/mpirun/new> mpirun -np 16 -exe /bin/hostname -verbose 1
```

Chapter 11. mpirun  179
**Using -psets_per_bp**

Example 11-7 illustrates the usage of \texttt{-psets\_per\_bp}. The number of psets per base partition is defined in the db.properties file. The value can be overridden with the \texttt{-psets\_per\_bp} option.

Example 11-7 \texttt{psets\_per\_bp}

```bash
dd2sys1fen3:~/bgp/control/mpirun/new> mpirun -psets_per_bp 16 -shape 4x4x2 -exe /bin/hostname
```

**Using a predefined partition and -np**

Example 11-8 shows a simple script to invoke \texttt{mpirun}.

Example 11-8 \texttt{csh script to invoke mpirun}

```bash
[descartes:/bgusr/cpsosa/pallas] ./run.pallas >& pallas_july06_2007_bgp.out
where the script run.pallas is:
#!/bin/csh
set MPIRUN="mpirun"
set MPIOPT="-np 32"
set MODE="-mode VN"
set PARTITION="-partition N01_32_1"
set WDIR="-cwd /bgusr/cpsosa/pallas"
```
set EXE="/bgusr/cpsosa/pallas/PMB-MPI1"
#
$MPIRUN $PARTITION $MPIOPT $MODE $WDIR $EXE
#
echo "That's all folks!!"

### Using environmental variables

Example 11-9 shows use of -env to define environmental variables.

**Example 11-9 Use of -env**

```
[descartes:/bgusr/cpsosa] mpirun -partition N00_32_1 -np 32 -mode SMP -cwd /bgusr/cpsosa -exe a.out -env "OMP_NUM_THREADS=4"
```

### Using stdin from a terminal

In Example 11-10, the user types the user's name bgp user in response to the job's stdout. After a while, the job is terminated when the user presses Ctrl+C to send `mpirun` a SIGINT.

**Example 11-10 Usage of stdin from a terminal**

```
dd2sys1fen3:~/bgp/control/mpirun/new> mpirun -partition R00-M0-N00 -verbose 0 -exe /BGPhome/stdin.sh -np 1
What's your name?
bgp user
hello bgp user
What's your name?
<Aug 11 15:33:44.021105> FE_MPI (WARN) : SignalHandler() -
<Aug 11 15:33:44.021173> FE_MPI (WARN) : SignalHandler() -
!------------------------------------------------!
<Aug 11 15:33:44.021201> FE_MPI (WARN) : SignalHandler() - ! mpirun is now taking all the
necessary actions !
<Aug 11 15:33:44.021217> FE_MPI (WARN) : SignalHandler() - ! to terminate the job and to free
the resources !
<Aug 11 15:33:44.021233> FE_MPI (WARN) : SignalHandler() - ! occupied by this job. This might
take a while... !
<Aug 11 15:33:44.021261> FE_MPI (WARN) : SignalHandler() -
!------------------------------------------------!
<Aug 11 15:33:44.050365> BE_MPI (WARN) : Received a message from frontend
<Aug 11 15:33:59.532817> FE_MPI (ERROR): Failure list:
<Aug 11 15:33:59.532899> FE_MPI (ERROR): - 1. Execution interrupted by signal (failure #71)
dd2sys1fen3:~/bgp/control/mpirun/new>
```

### Using stdin from a file or pipe

Example 11-11 illustrates the use of stdin from a file or pipe.

**Example 11-11 Usage of stdin from a file or pipe**

```
dd2sys1fen3:~/bgp/control/mpirun/new> cat ~/stdin.cc
#include <iostream>
using namespace std;

int main() {
    unsigned int lineno = 0;
```

---

`Chapter 11. mpirun` 181
while (cin.good()) {
    string line;
    getline(cin, line);
    if (!line.empty()) {
        cout << "line " << ++lineno << ": " << line << endl;
    }
}

d2sys1fen3:~/bgp/control/mpirun/new> cat stdin.txt

d2sys1fen3:~/bgp/control/mpirun/new> cat stdin.txt | mpirun -partition R00-M0-N00 -verbose 0 -exe /BGPhome/stdin_test -np 1

d2sys1fen3:~/bgp/control/mpirun/new> mpirun -partition R00-M0-N00 -verbose 1 -exe /BGPhome/datespinner.sh -np 1 | tee -i datespinner.out
<Aug 12 10:27:10.997374> FE_MPI (Info) : Invoking mpirun backend
<Aug 12 10:27:11.155416> BRIDGE (Info) : rm_set_serial() - The machine serial number (alias) is BGP
<Aug 12 10:27:11.194557> FE_MPI (Info) : Preparing partition
<Aug 12 10:27:11.234550> BE_MPI (Info) : Examining specified partition
<Aug 12 10:27:11.823425> BE_MPI (Info) : Checking partition R00-M0-N00 initial state ...
<Aug 12 10:27:11.823499> BE_MPI (Info) : Partition R00-M0-N00 initial state = READY ('I')
<Aug 12 10:27:11.823516> BE_MPI (Info) : Checking partition owner...
<Aug 12 10:27:11.823532> BE_MPI (Info) : partition R00-M0-N00 owner is 'userX'
<Aug 12 10:27:11.824744> BE_MPI (Info) : Partition owner matches the current user
<Aug 12 10:27:11.824870> BE_MPI (Info) : Done preparing partition
<Aug 12 10:27:11.864539> FE_MPI (Info) : Adding job
<Aug 12 10:27:11.864876> BE_MPI (Info) : No CWD specified ('-cwd' option)
<Aug 12 10:27:11.864903> BE_MPI (Info) : - it will be set to '/BGPhome/usr3/bgp/control/mpirun/new'
<Aug 12 10:27:11.865046> BE_MPI (Info) : Adding job to database...
<Aug 12 10:27:11.944540> FE_MPI (Info) : Job added with the following id: 15
<Aug 12 10:27:11.944593> FE_MPI (Info) : Starting job 15
<Aug 12 10:27:12.004492> FE_MPI (Info) : Waiting for job to terminate
<Aug 12 10:27:12.816792> BE_MPI (Info) : IO - Threads initialized
<Aug 12 10:28:11.159680> FE_MPI (Info) : SignalHandler() -
<Aug 12 10:28:11.159737> FE_MPI (Info) : SignalHandler() - ! Received signal SIGINT
<Aug 12 10:28:11.159760> FE_MPI (WARN) : SignalHandler() -
<Aug 12 10:28:11.159773> FE_MPI (WARN) : SignalHandler() - !------------------------------------------------!
<Aug 12 10:28:11.159788> FE_MPI (WARN) : SignalHandler() - ! mpirun is now taking all the necessary actions!
<Aug 12 10:28:11.159801> FE_MPI (WARN) : SignalHandler() - ! to terminate the job and to free the resources!
<Aug 12 10:28:11.159815> FE_MPI (WARN) : SignalHandler() - ! occupied by this job. This might take a while...!
<Aug 12 10:28:11.159829> FE_MPI (WARN) : SignalHandler() - !------------------------------------------------!
<Aug 12 10:28:11.159842> FE_MPI (WARN) : SignalHandler() -
<Aug 12 10:28:11.201498> FE_MPI (Info) : Termination requested while waiting for backend response
<Aug 12 10:28:11.201534> FE_MPI (Info) : Starting cleanup sequence
<Aug 12 10:28:11.201794> BE_MPI (WARN) : Received a message from frontend
<Aug 12 10:28:11.201863> BE_MPI (WARN) : Execution of the current command interrupted
<Aug 12 10:28:11.201942> BE_MPI (Info) : Starting cleanup sequence
<Aug 12 10:28:11.201986> BE_MPI (Info) : cancel_job() - Cancelling job 15
<Aug 12 10:28:11.204567> BE_MPI (Info) : cancel_job() - Job 15 state is RUNNING ('R')
<Aug 12 10:28:11.230352> BE_MPI (Info) : cancel_job() - Job 15 state is DYING ('D'). Waiting...
<Aug 12 10:28:16.249665> BE_MPI (Info) : cancel_job() - Job 15 has been moved to the history table
<Aug 12 10:28:16.255793> BE_MPI (Info) : cleanupDatabase() - Partition was supplied with READY ('I') initial state
<Aug 12 10:28:16.255996> BE_MPI (Info) : cleanupDatabase() - No need to destroy the partition
<Aug 12 10:28:16.591667> FE_MPI (ERROR) : Failure list:
<Aug 12 10:28:16.591708> FE_MPI (ERROR) : - 1. Execution interrupted by signal (failure #71)
<Aug 12 10:28:16.591722> FE_MPI (Info) : == FE completed ==
<Aug 12 10:28:16.591736> FE_MPI (Info) : == Exit status: 1 ==

dd2sys1fen3:~/bgp/control/mpirun/new> cat datespinner.out
Sun Aug 12 10:28:54 CDT 2007
Sun Aug 12 10:29:04 CDT 2007
Sun Aug 12 10:29:09 CDT 2007
Error when requesting more nodes than the partition size

Example 11-13 shows a case where the user specified a value for -np larger than the number provided in the partition.

Example 11-13 Error due to requesting an -np value greater than the partition size

```
dd2sysfen3:~/bgp/control/mpirun/new> .mpirun -partition R00-M0-N00 -verbose 0 -exe /bin/hostname -np 55
Aug 11 15:28:46.797523> BE_MPI (ERROR): Job execution failed
Aug 11 15:28:46.797634> BE_MPI (ERROR): Job 8 is in state ERROR ('E')
Aug 11 15:28:46.842559> FE_MPI (ERROR): Job execution failed (error code - 50)
Aug 11 15:28:46.842738> FE_MPI (ERROR): - Job execution failed - job switched to an error state
Aug 11 15:28:46.851840> BE_MPI (ERROR): The error message in the job record is as follows:
Aug 11 15:28:46.851900> BE_MPI (ERROR):   "BG_SIZE of 55 is greater than block 'R00-M0-N00' size of 32"
```

Killing a hung job or a running job

`mpirun` has the capability to kill the job and free your partition if it was booted by `mpirun`. To kill your job, we recommend that you send `mpirun` a SIGINT (kill -2) while the job is running or hung. We recommend that you do not use SIGKILL because subsequent jobs might experience problems.

Be aware that using SIGINT is somewhat time consuming depending on the state of the job. Therefore, do not expect it to return control instantaneously. Alternatively, if you do not want to wait, try sending `mpirun` three SIGINTs in succession. In this case, it immediately returns control to your shell. However, as the warning messages indicate, your job, partition, or both might be left in a bad state. Ensure that they are cleaned up correctly before you attempt to use them again. Example 11-14 illustrates this procedure.

Example 11-14 Proper way to kill hung or running jobs

```
From window 2: (open another window to kill a job)
ps -ef | grep cpsosa

cpsosa 23393 23379  0 13:21 pts/13  00:00:00 /bgsys/drivers/ppcfloor/bin/mpirun -partition N04_32_1 -np 32 -mode VN -cwd /bgusr/cpsosa/red/pallas -exe /bgusr/cpsosa/red/pallas/PMB-MPI1

From window 1: (where the job is running)
.
.
32768  1000  95.49  95.49  95.49  654.50
 65536  640 183.20 183.20 183.20 682.31

<Oct 18 13:22:10.804667> FE_MPI (WARN) : SignalHandler() -
<Oct 18 13:22:10.804743> FE_MPI (WARN) : SignalHandler() -
`----------------------------------------`
<Oct 18 13:22:10.804769> FE_MPI (WARN) : SignalHandler() - ! mpirun is now taking all the necessary actions !
<Oct 18 13:22:10.804794> FE_MPI (WARN) : SignalHandler() - ! to terminate the job and to free the resources !
```
11.7 mpirun APIs

When writing programs to the mpirun APIs, you must consider these requirements:

- Currently, SUSE Linux Enterprise Server (SLES) 10 for PowerPC is the only supported platform.
- C and C++ are supported with the GNU gcc 4.1.2 level compilers. For more information and downloads, refer to the following Web address:
- The include file is include/sched_api.h.
- Only support for both 64-bit dynamic libraries is provided, and the 64-bit dynamic library file called by mpirun must be called libsched_if.so.

mpirun can retrieve run-time information directly from the scheduler without using command-line parameters or environment variables. Each time mpirun is invoked, it attempts to load a dynamically loaded library called libsched_if.so. mpirun looks for this library in a set of directories as described by the dlopen command manual pages.

If the plug-in library is found and successfully loaded, mpirun calls the get_parameters() function within that library to retrieve the information from the scheduler. The get_parameters() function returns the information in a data structure of type sched_params. This data structure contains a set of fields that describe the block that the scheduler has allocated the job to run. Each field corresponds to one of the command-line parameters or environment variables.

mpirun complements the information that is retrieved by get_parameters() with values from its command-line parameters and environment variables. It gives precedence to the information that is retrieved by get_parameters() first, then to its command-line parameters, and finally to the environment variables. For example, if the number of processors retrieved by get_parameters() is 256, the -np command-line parameter is set to 512, and the environment variable MPIRUN_NP is set to 448, mpirun runs the job on 256 compute nodes.

The block ID to use for that job can be the one specified by the MPIRUN_PARTITION environment variable, if both the get_parameters() function does not retrieve the block ID and the -partition command-line parameter is not specified.

If mpirun is invoked with the -verbose parameter with a value greater than 0, it displays information that describes the loading of the dynamically loaded library. The message "Scheduler interface library loaded" indicates that mpirun found the library, loaded it, and is using it.
The implementation of the libsched_if.so library is scheduling-system specific. In general, this library should use the scheduler's APIs to retrieve the required information and convert it to the sched_params data type for mpirun to use. The only requirement is that the library interface conform to the definitions in the sched_api.h header file distributed with the mpirun binaries. This interface may be modified with future releases of mpirun.

The mpirun plug-in interface also requires the implementer provide an mpirun_done() function (void mpirun_done(int res);). This function is called by mpirun just before it exits. It is used to signal the plug-in implementer that mpirun is terminating.

You can find more information about the library implementation and data structures in the sched_api.h header file.

The following APIs are supported for mpirun:

- Int get_parameters(sched_params_t *params);
  This function is used to provide input parameters to mpirun from your application. If a value of 1 (failure) is returned on the get_parameters() call, then mpirun proceeds to terminate. Some external resource managers use this technique to prevent standalone mpirun from being used. If the plug-in provider wants mpirun processing to continue, then they must return a 0 (success) value on the get_parameters() call.

- void mpirun_done(int res);
  This function is called by mpirun just before it calls the exit() function. It can be used to signal the scheduler that mpirun is terminating.

- void mpirun_done_enhanced(sched_result_t* res);
  This function is called by mpirun just before it calls the exit() function. It can be used to signal the scheduler that mpirun is terminating. This is an enhanced version of the original mpirun_done() callback and is intended to convey information about boot failures back to a resource scheduler. This is an optional plug-in interface, and it does not have to be included as part of the libsched_if.so shared library.
High-Throughput Computing (HTC) paradigm

In Chapter 7, “Parallel paradigms” on page 67, we described the high-performance computing (HPC) paradigms. Applications that run in an HPC environment make use of the network to share data among MPI tasks. In other words, the MPI tasks are tightly coupled.

In this chapter we describe a paradigm that complements the HPC environment. This mode of running applications emphasizes Blue Gene capacity. An application runs loosely coupled; that is, multiple instances of the applications do not require data communication. The concept of High-Throughput Computing (HTC) has been defined by Condor High Throughput Computing and others (see the following URL):

http://www.cs.wisc.edu/condor/htc.html

In this chapter we cover the implementation of HTC as part of Blue Gene/P functionality. We provide an overview how HTC is implemented and how applications can take advantage of it. We cover the following topics:

- HTC design
- Booting a partition in HTC mode
- Running a job using submit
- Checking HTC mode
- submit API

For a more detailed description of HTC, and how it is integrated into the control system of Blue Gene/P, see IBM System Blue Gene Solution: Blue Gene/P System Administration, SG24-7417.
12.1 HTC design

HTC focuses on pushing a large number of relatively short jobs through the control system. Such a design is significantly different from the traditional High Performance Computing HPC focus of Blue Gene/P where a single job runs the same executable on each node in the partition. In HTC mode, each compute node can run one, two, or four different jobs depending on the mode (SMP, DUAL, or virtual node mode) the partition was booted in. Each job can run under a different user name, with separate stdin, stdout, and stderr. The executables are compiled in the same way that HPC executables are compiled for Blue Gene/P, but they cannot use MPI. Because each node can run a different executable, and jobs start and stop independently of each other, it does not make sense to use MPI in this mode.

12.2 Booting a partition in HTC mode

htcpartition

When booting an HTC partition, you need to specify the job mode at boot time, either SMP, DUAL, or VN mode. This is different from regular MPI partitions where the mode can be given at the run time of the job. To boot a partition in HTC mode, use the htcpartition utility from a FEN or the service node. For more information see Appendix F, “htcpartition” on page 307.

mmcs_db_console

A system administrator can also boot a partition in HTC mode using the mmcs_db_console; see IBM System Blue Gene Solution: Blue Gene/P System Administration, SG24-7417.

12.3 Running a job using submit

The interface to run an HTC job is the submit command. It is similar to mpirun in the sense that it acts as a shadow of the job running on the compute node. It transparently forwards stdin, stdout, stderr, signals, and terminates when the job is complete. However, mpirun can boot either predefined partitions or dynamic partitions based on size and shape requirements of a job. submit performs neither of these tasks; it requires a partition to already be booted in HTC mode prior to submitting the job. Other differences are less noticeable to users, such as the lack of a hybrid front end and back end design, or Bridge API calls in submit that are present in mpirun. The arguments supported by submit are also somewhat different than mpirun. Many of the mpirun arguments are not applicable when running an HTC job compared to a large parallel job.

The syntax for the submit command is as follows:

```
./submit [options] or
./submit [options] binary [arg1 arg2... argn]
```

Table 12-1 contains the available options for the submit command.

<table>
<thead>
<tr>
<th>Job options (and syntax)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-exe &lt;exe&gt;</td>
<td>Executable to run.</td>
</tr>
<tr>
<td>-args “arg1 arg2 ... argn”</td>
<td>Arguments must be enclosed in double quotes.</td>
</tr>
<tr>
<td>-env &lt;env=value&gt;</td>
<td>Used to add an environmental for the job.</td>
</tr>
</tbody>
</table>
Chapter 12. High-Throughput Computing (HTC) paradigm

Environmental variables
Some arguments have a corresponding environment variable. If both an environment variable and an argument are given, precedence is given to the argument.

- --pool SUBMIT_POOL
- --cwd SUBMIT_CWD
- --port SUBMIT_PORT

How to use submit
This section provides selected examples on how to invoke the command and how to use the location and pool arguments.

location argument
The --location argument requests a specific compute core location to run the job; the syntax is in the form of RXX-MX-NXX-JXX-CXX. The rack numbers (RXX) may range between R00 and RFF. The midplane numbers (MX) may range between M0 (bottom) and M1 (top). The node card numbers (NXX) may range between N00-N15. The compute card numbers (JXX) may range between J04 and J35. Note that J00 and J01 are I/O nodes, and J02 and J03 are unused. The compute core numbers (CXX) can range between C00 and C03. Note that C00

<table>
<thead>
<tr>
<th>Job options (and syntax)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-exp_env &lt;env&gt;</td>
<td>Used to export an environmental to the job's environment.</td>
</tr>
<tr>
<td>-env_all</td>
<td>Adds all current environmentalis to the job's environment.</td>
</tr>
<tr>
<td>-cwd &lt;cwd&gt;</td>
<td>Sets the current working directory for the job.</td>
</tr>
<tr>
<td>-timeout &lt;seconds&gt;</td>
<td>Number of seconds to wait for before the job is killed.</td>
</tr>
<tr>
<td>-strace</td>
<td>Run job under system call tracing.</td>
</tr>
</tbody>
</table>

| Resource Options         | |
|--------------------------| |
| -mode <SMP/DUAL/IVN>     | Job mode. |
| -location <Rxx-Mx-Nxx-Jxx-Cxx> | Compute core location (regular expressions are supported). |
| -pool <id>               | Compute node pool ID. |

| Options                  | |
|--------------------------| |
| -port <port>             | Listen port of the submit mux (default = 10246). |
| -trace <0 - 7>           | Tracing level (default = 0). |
| -enable_tty_reporting    | Disable the default line buffering of stdin, stdout, and stderr when input (stdin) or output (stdout/stderr) is not a tty. |
| -raise                   | If a job dies with a signal, submit raises this signal. |
is valid for SMP, DUAL, and VN mode. Core C01 is valid only for VN mode. Core 02 is valid for VN and DUAL modes. Core 03 is valid only for VN mode.

The --location argument is combined with your user ID and --mode argument to find an available location to run the job. If any of these parameters do not match the list of what is available, the job is not started and an error message is returned. See Example 12-1.

It is also possible to omit a portion of the location. If the core is omitted (for example, --location R00-M0-N14-J09), one of the cores in the compute card is chosen. If the compute card is omitted (for example, --location R00-M0-N14), a compute card on the node card is chosen.

**Example 12-1  Requesting specific location**

```bash
bgp6fen:~> submit --cwd /bgusr/tests --location R00-M0-N14-J09-C00 --mode vn --exe hello
hello world
```

If the location you request is busy (job already running), you see an error message (Example 12-2).

**Example 12-2  Job already running**

```bash
bgp6fen:~> submit --cwd /bgusr/tests --exe hello --location R00-M0-N14-J09-C00 --mode vn
```

May 07 15:05:43 (U) [4398046675584] (submit.cc:1237:cleanup) failed to add job: location R00-M0-N14-J09-C00 is not available

If the location you request was booted in a mode different than the --mode argument you give, you see an error message (Example 12-3).

**Example 12-3  Node mode conflict**

```bash
bgp6fen:~> submit --cwd /bgusr/tests --exe hello --location R00-M0-N14-J09-C00 --mode SMP
```

May 07 15:06:50 (U) [4398046675584] (submit.cc:1237:cleanup) failed to add job: location R00-M0-N14-J09-C00 mode (VN) is incompatible with requested mode (SMP)

Similarly, if your user ID does not have permission to run on the location requested, you see an error message (Example 12-4).

**Example 12-4  Permission problem**

```bash
bgp6fen:~> whoami
bgpadmin
```

```bash
bgp6fen:~> submit --cwd /bgusr/tests --exe hello --location R00-M0-N14-J09-C00 --mode vn
```

May 07 15:13:28 (U) [4398046675888] (submit.cc:1237:cleanup) failed to add job: user bgpadmin is not allowed to run at location R00-M0-N14-J09-C00

**pool argument**

A pool is a collection of compute nodes and is represented by an ID just as a partition is. A pool consists of zero or more partitions. By default, each partition's pool ID is its partition ID. Outside the framework of a job scheduler, this should always be the case. Thus,
Example 12-5 shows how to run a job on any available compute node in partition CHEMISTRY.

Example 12-5  Pool argument

```
dd2sys1fen3:~> submit --pool CHEMISTRY--exe hello_world
May 07 16:28:59 (U) [4398046675888] (submit.cc:1237:cleanup) failed to add job: could not find available location matching resource request
dd2sys1fen3:~>
```

If no compute nodes are available, an error message is displayed as shown in Example 12-5.

### 12.4 Checking HTC mode

An application can check whether the node it is running on was booted in HTC mode using the personality information. Example 12-6 illustrates how to use the personality to inquire about the HTC mode.

Example 12-6  Checking whether a node was booted in HTC mode

```
#include <unistd.h>
#include <common/bgp_personality.h>
#include <common/bgp_personality_inlines.h>
#include <spi/kernel_interface.h>

int main()
{
    // get our personality
    _BGP_Personality_t pers;
    if (Kernel_GetPersonality(&pers, sizeof(pers)) == -1) {
        fprintf(stderr, "could not get personality\n");
        exit(EXIT_FAILURE);
    }

    // check HTC mode
    if (pers.Kernel_Config.NodeConfig & _BGP_PERS_ENABLE_HighThroughput) {
        // do something HTC specific
    } else {
        // do something else
    }
}
```
12.5 submit API

When writing programs to the submit APIs, you must consider these requirements:

- Currently, SUSE Linux Enterprise Server (SLES) 10 for PowerPC is the only supported platform.
- C and C++ are supported with the GNU gcc 4.1.2 level compilers. For more information and downloads, refer to the following web address:
  
  http://gcc.gnu.org

- The include file is include/submit_api.h.
- Only 64-bit dynamic libraries are supported.

submit can retrieve and provide run-time information directly from the scheduler without using command-line parameters or environmental variables. Each time submit is invoked, it attempts to load a dynamically loaded library called libsubmit_if.so. submit looks for this library in a series of directories as described by the dlopen manual page. If the plug-in library is found and successfully loaded, submit invokes the following three methods:

- int get_parameters(htc_sched_params *params);
  
  This function is used to provide input parameters to submit from an external scheduler. A non-zero return code is fatal and causes submit to immediately exit without running a job.

- void submit_info(const htc_sched_info* result);
  
  This function is called by submit when the job is started. The htc_sched_info structure contains details about the job being started (pool, job ID, compute node location, partition, and so on). If the job never starts for whatever reason, this function is not called.

- void submit_done(const htc_sched_result *results);
  
  This function is called by submit just before it calls the exit() function. It can be used to signal the scheduler that submit is terminating. The htc_sched_result structure contains job metadata (pool, job ID, compute node location, return code, and so on). The job ID is 0 if the job did not run.

The return code information in the job metadata provides additional details about the submit request.
Job scheduler interfaces

In this part, we provide information about the job scheduler APIs:

- Chapter 13, “Control system (Bridge) APIs” on page 195
- Chapter 14, “Real-time Notification APIs” on page 237
- Chapter 15, “Dynamic Partition Allocator APIs” on page 257
Control system (Bridge) APIs

In this chapter, we define a list of APIs into the Midplane Management Control System (MMCS) that can be used by a job management system. The `mpirun` program that ships with the Blue Gene/P software is an application that uses these APIs to manage partitions, jobs, and other similar aspects of the Blue Gene/P system. You can use these APIs to write applications to manage Blue Gene/P partitions and control Blue Gene/P job execution, as well as other similar administrative tasks.

In this chapter, we present an overview of the support provided by the APIs and discuss the following topics:

- API requirements
- APIs
- Small partition allocation
- API examples
13.1 API requirements

The several requirements for writing programs to the Bridge API are as follows:

- Currently, SUSE Linux Enterprise Server (SLES) 10 for PowerPC is the only supported platform.
- C and C++ are supported with the GNU gcc 4.1.2 level compilers. For more information and downloads, refer to the following Web address:
  
  http://gcc.gnu.org/
- All required include files are installed in the /bgsys/drivers/ppcfloor/include directory. See Appendix B, "Header files and libraries" on page 293 for additional information about include files. The include file for the Bridge API is rm_api.h.
- The Bridge API supports 64-bit applications that use dynamic linking using shared objects. The required library files are installed in the /bgsys/drivers/ppcfloor/lib64 directory.

The shared object for linking to the Bridge API is libbgpbridge.so. The libbgpbridge.so library has dependencies on other libraries that are included with the Blue Gene/P software, including:

- libbgpconfig.so
- libbgpdb.so
- libsaymessage.so
- libtableapi.so

These files are installed with the standard system installation procedure. They are contained in the bgpbase.rpm file.

The requirements for writing programs to the Bridge API are explained in the following sections.

13.1.1 Configuring environment variables

Table 13-1 provides information about the environment variables that are used to control the Bridge API.

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB_PROPERTY</td>
<td>Yes</td>
<td>This variable must be set to the path of the db.properties file with database connection information. For default installation, the path to this file is /bgsys/local/etc/db.properties.</td>
</tr>
<tr>
<td>BRIDGE_CONFIG</td>
<td>Yes</td>
<td>This variable must be set to the path of the bridge.config file that contains the Bridge API configuration values. For a default installation, the path to this file is /bgsys/local/etc/bridge.config.</td>
</tr>
<tr>
<td>BRIDGE_DUMP_XML</td>
<td>No</td>
<td>When set to any value, this variable causes the Bridge API to dump its in-memory XML streams to files in /tmp for debugging. When this variable is not set, the Bridge API does not dump its in-memory XML streams.</td>
</tr>
</tbody>
</table>

For more information about the db.properties and bridge.config files, see IBM System Blue Gene Solution: Blue Gene/P System Administration, SG24-7417.
13.1.2 General comments

All the APIs have general considerations that apply to all calls. In the following list, we highlight the common features:

- All the API calls return a status_t indicating either success or an error code.
- The get APIs that retrieve a compound structure include accessory functions to retrieve relevant nested data.
- The get calls allocate new memory for the structure to be retrieved and return a pointer to the allocated memory in the corresponding argument.
- To add information to MMCS, use new functions as well as rm_set_data(). The new functions allocate memory for new data structures, and the rm_set_data() API is used to fill these structures.
- For each get and new function, a corresponding free function frees the memory allocated by these functions. For instance, rm_get_BG(rm_BG_t **bg) is complemented by rm_free_BG(rm_BG_t *bg).
- The caller is responsible for matching the calls to the get and new allocators to the corresponding free deallocators. Memory leaks result if this is not done.

Memory allocation and deallocation

Some API calls result in memory being allocated on behalf of the user. The user must call the corresponding free function to avoid memory leaks, which can cause the process to run out of memory.

For the rm_get_data() API, see 13.2.8, “Field specifications for the rm_get_data() and rm_set_data() APIs” on page 215 for a complete list of the fields that require calls to free memory.

Avoiding invalid pointers

Some APIs return a pointer to an offset in a data structure, or object, that was previously allocated (based on element in rm_get_data()) - for example, the rm_get_data() API call uses the RM PartListNextPart specification. In this example, element is a partition list, and it returns a pointer to the first or next partition in the list. If the caller of the API frees the memory of the partition list (element) and data is pointing to a subset of that freed memory, the data pointer is invalid. The caller must make sure that no further calls are made against a data structure returned from an rm_get_data() call after it is freed.

First and next calls

Before a next call can be made against a data structure returned from an rm_get_data() call, the first call must have been made. Failure to do so results in an invalid pointer, either pointing at nothing or at invalid data.

Example 13-1 shows correct usage of the first and next API calls. Notice how memory is freed after the list is consumed.

Example 13-1 Correct usage of first and next API calls

```c
status_t stat;
int list_size = 0;
rm_partition_list_t * bgp_part_list = NULL;
rm_partition_t * bgp_part = NULL;

// Get all information on existing partitions
stat = rm_get_partitions_info(PARTITION_ALL_FLAG, &bgp_part_list);
```
if (stat != STATUS_OK) {
    // Do some error handling here...
    return;
}

// How much data (# of partitions) did we get back?
rm_get_data(bgp_part_list, RM_PartListSize, &list_size);

for (int i = 0; i < list_size; i++) {
    // If this is the first time through, use RM_PartListFirstPart
    if (i == 0) {
        rm_get_data(bgp_part_list, RM_PartListFirstPart, &bgp_part);
    }
    // Otherwise, use RM_PartListNextPart
    else {
        rm_get_data(bgp_part_list, RM_PartListNextPart, &bgp_part);
    }
}

// Make sure we free the memory when finished
stat = rm_free_partition_list(bgp_part_list);
if (stat != STATUS_OK) {
    // Do some error handling here...
    return;
}

13.2 APIs

In the following sections, we describe details about the APIs.

13.2.1 API to the Midplane Management Control System

The Bridge API contains an `rm_get_BG()` function to retrieve current configuration and status information about all the physical components of the Blue Gene/P system from the MMCS database. The Bridge API also includes functions that add, remove, or modify information about transient entities, such as jobs and partitions.

The `rm_get_BG()` function returns all the necessary information to define new partitions in the system. The information is represented by three lists: a list of base partitions (BPs), a list of wires, and a list of switches. This representation does not contain redundant data. In general, it allows manipulation of the retrieved data into any desired format. The information is retrieved using a structure called `rm_BG_t`. It includes the three lists that are accessed using iteration functions and the various configuration parameters, for example, the size of a base partition in compute nodes.

All the data retrieved by using the get functions can be accessed using `rm_get_data()` with one of the specifications listed in 13.2.8, “Field specifications for the rm_get_data() and rm_set_data() APIs” on page 215. Additional get functions can retrieve information about the partitions and job entities.

The `rm_add_partition()` and `rm_add_job()` functions add and modify data in the MMCS. The memory for the data structures is allocated by the `new` functions and updated using the `rm_set_data()` function. The specifications that can be set using the `rm_set_data()` function are shown in 13.2.8, “Field specifications for the rm_get_data() and rm_set_data() APIs” on page 215.
13.2.2 Asynchronous APIs

Some APIs that operate on partitions or jobs are documented as being asynchronous. Asynchronous means that control returns to your application before the operation requested is complete.

Before you perform additional operations on the partition or job, make sure that it is in a valid state by using the `rm_get_partition_info()` or `rm_get_job()` APIs to check the current state of the partition or job.

13.2.3 State sequence IDs

For most Blue Gene objects that have a state field, there is a corresponding sequence ID field for the state value. MMCS guarantees that any time the state field changes for a given object, the associated sequence ID is incremented.

The sequence ID fields can be used to determine which state value is more recent. A state value with a higher corresponding sequence ID is the more recent value. This comparison can be helpful for applications that retrieve state information from multiple sources such as the Bridge API and the real-time APIs.

The function to increment sequence IDs only occurs if the real-time APIs are configured for the system. For information about configuring the real-time APIs, see IBM System Blue Gene Solution: Blue Gene/P System Administration, SG24-7417.

13.2.4 Bridge API return codes

When a failure occurs, an API invocation returns an error code. You can use the error code to take corrective actions within your application. In addition, a failure always generates a log message, which provides more information for the possible cause of the problem and an optional corrective action. These log messages are used for debugging and programmed recovery of failures.

The design aims at striking a balance between the number of error codes detected and the different error paths per return code. Thus, some errors have specific return codes, while others have more generic ones. The Bridge API has the following return codes:

- STATUS_OK: The invocation completed successfully.
- PARTITION_NOT_FOUND: The required partition specified by the ID cannot be found in the control system.
- JOB_NOT_FOUND: The required job specified by the ID cannot be found in the control system.
- BP_NOT_FOUND: One or more of the base partitions in the `rm_partition_t` structure do not exist.
- SWITCH_NOT_FOUND: One or more of the switches in the `rm_partition_t` structure do not exist.
- JOB_ALREADY_DEFINED: A job with the same name already exists.
- PARTITION_ALREADY_DEFINED: A partition already exists with the ID specified.
- CONNECTION_ERROR: The connection with the control system has failed or could not be established.
- INVALID_INPUT: The input to the API invocation is invalid, which is due to missing required data, illegal data, and so on.
INCOMPATIBLE_STATE: The state of the partition or job prohibits the specific action. See Figure 13-1 on page 207, Figure 13-2 on page 212, Figure 13-3 on page 213, and Figure 13-4 on page 214 for state diagrams.

INCONSISTENT_DATA: The data retrieved from the control system is not valid.

INTERNAL_ERROR: Such errors do not belong to any of the previously listed categories, such as a memory allocation problem or failures during the manipulation of internal XML streams.

13.2.5 Blue Gene hardware resource APIs

In this section, we describe the APIs that are used to manage the hardware resources in the Blue Gene system:

- `status_t rm_get_BG(rm_BG_t **BG);`
  This function retrieves a snapshot of the Blue Gene/P machine, held in the `rm_BG_t` data structure.

  The following return codes are possible:
  - STATUS_OK
  - CONNECTION_ERROR
  - INCONSISTENT_DATA
    - List of base partitions is empty.
    - Wire list is empty, and the number of base partitions is greater than one.
    - Switch list is empty, and the number of base partitions is greater than one.
  - INTERNAL_ERROR

- `status_t rm_get_data(rm_element_t *rme, enum RMSpecification spec, void *result);`
  This function returns the content of the requested field from a valid `rm_element_t` (Blue Gene object, base partition object, wire object, switch object, and so on). The specifications that are available when using `rm_get_data()` are listed in 13.2.8, “Field specifications for the `rm_get_data()` and `rm_set_data()` APIs” on page 215, and are grouped by the object type that is being accessed.

  The following return codes are possible:
  - STATUS_OK
  - INVALID_INPUT
    - The specification `spec` is unknown.
    - The specification `spec` is illegal (per the “rme” element).
  - INTERNAL_ERROR

- `status_t rm_get_nodecards(rm_bp_id_t bpid, rm_nodecard_list_t **nc_list);`
  This function returns all node cards in the specified base partition.

  The following return codes are possible:
  - STATUS_OK
  - CONNECTION_ERROR
  - INCONSISTENT_DATA
    - The Base Partition was not found.
  - INTERNAL_ERROR
status_t rm_get_serial(rm_serial_t *serial);

This function gets the machine serial number that was set previously by rm_set_serial().

The following return codes are possible:
- STATUS_OK
- INTERNAL_ERROR

status_t rm_set_data(rm_element_t *rme, enum RMSpecification spec, void *result);

This function sets the value of the requested field in the rm_element_t (Blue Gene/P object, base partition object, wire object, switch object, and so on). The specifications, which are available when using rm_set_data(), are listed in 13.2.8, “Field specifications for the rm_get_data() and rm_set_data() APIs” on page 215, and are grouped by the object type that is being accessed.

The following return codes are possible:
- STATUS_OK
- INVALID_INPUT
  - The specification spec is unknown.
  - The specification spec is illegal (per the rme element).
- INTERNAL_ERROR

status_t rm_set_serial(rm_serial_t serial);

This function sets the machine serial number to be used in all the API calls following this call. The database can contain more than one machine. Therefore, it is necessary to specify which machine to work with.

The following return codes are possible:
- STATUS_OK
- INVALID_INPUT
  - The machine serial number serial is NULL.
  - The machine serial number is too long.

13.2.6 Partition-related APIs

In this section, we describe the APIs used to create and manage partitions in the Blue Gene system:

status_t rm_add_partition(rm_partition_t* p);

This function adds a partition record to the database. The partition structure includes an ID field that is filled by the resource manager.

The following return codes are possible:
- STATUS_OK
- CONNECTION_ERROR
- INVALID_INPUT: The data in the rm_partition_t structure is invalid.
  - No base partition nor switch list is supplied.
  - Base partition or switches do not construct a legal partition.
  - No boot images or boot image name is too long.
  - No user or user name is too long.
- BP_NOT_FOUND:
  One or more of the base partitions in the rm_partition_t structure does not exist.
- **SWITCH_NOT_FOUND:**
  One or more of the switches in the `rm_partition_t` structure does not exist.
- **INTERNAL_ERROR**

```c
status_t rm_add_part_user (pm_partition_id_t partition_id, const char *user);
```
This function adds a new user to the partition. If a partition is in "free" state any user can add users. If the partition is in any other state only the partition's owner can add users.

The following return codes are possible:
- **STATUS_OK**
- **CONNECTION_ERROR**
- **INVALID_INPUT**
  - `partition_id` is NULL or the length exceeds the limitations of the control system.
  - `user` is NULL or the length exceeds the limitations of the control system.
  - `user` is already defined as the partition's user.
- **INTERNAL_ERROR**

```c
status_t rm_assign_job(pm_partition_id_t partition_id, db_job_id_t jid);
```
This function assigns a job to a partition. A job can be created and simultaneously assigned to a partition by calling `rm_add_job()` with a partition ID. If a job is created and not assigned to a specific partition, it can be assigned later by calling `rm_assign_job()`.

The following return codes are possible:
- **STATUS_OK**
- **CONNECTION_ERROR**
- **INVALID_INPUT**
  - `partition_id` is NULL or the length exceeds control system limitations.
- **PARTITION_NOT_FOUND**
- **JOB_NOT_FOUND**
- **INCOMPATIBLE_STATE**
  The current state of the partition is not RM_PARTITION_READY ("initialized"), the partition and job owners do not match, or the partition is in HTC mode.
- **INTERNAL_ERROR**

```c
status_t pm_create_partition(pm_partition_id_t partition_id);
```
This function allocates the necessary hardware for a partition, boots the partition, and updates the resulting status in the MMCS database.

**Note:** This API is asynchronous. Control returns to your application before the operation requested is complete.
- PARTITION_NOT_FOUND
- INCOMPATIBLE_STATE
  The current state of the partition prohibits its creation. See Figure 13-1 on page 207.
- INTERNAL_ERROR

```
status_t pm_destroy_partition(pm_partition_id_t partition_id);
```
This function shuts down a currently booted partition and updates the database accordingly.

**Note:** This API is asynchronous. Control returns to your application before the operation requested is complete.

The following return codes are possible:
- STATUS_OK
- CONNECTION_ERROR
- INVALID_INPUT
  - partition_id is NULL or the length exceeds the limitations of the control system.
- PARTITION_NOT_FOUND
- INCOMPATIBLE_STATE
  The state of the partition prohibits its destruction. See Figure 13-1 on page 207.
- INTERNAL_ERROR

```
status_t rm_get_partition(pm_partition_id_t partition_id, rm_partition_t **p);
```
This function retrieves a partition, according to its ID.

The following return codes are possible:
- STATUS_OK
- CONNECTION_ERROR
- INVALID_INPUT
  - partition_id is NULL or the length exceeds the limitations of the control system.
- PARTITION_NOT_FOUND
- INCONSISTENT_DATA
  The base partition or switch list of the partition is empty.
- INTERNAL_ERROR

```
status_t rm_get_partitions(rm_partition_state_t_flag_t flag, rm_partition_list_t **part_list);
```
This function is useful for status reports and diagnostics. It returns a list of partitions whose current state matches the flag. The possible flags are contained in the rm_api.h include file and listed in Table 13-2 on page 204. You can use OR on these values to create a flag for including partitions with different states.
The following return codes are possible:
- STATUS_OK
- CONNECTION_ERROR
- INCONSISTENT_DATA

  At least one of the partitions has an empty base partition list.

- INTERNAL_ERROR

  \texttt{status\_t\ rm\_get\_partitions\_info(rm\_partition\_state\_t\ flag\_t\ flag, \newline \hspace{1em} rm\_partition\_list\_t\ **\ part\_list);}\texttt{;}

  This function is useful for status reports and diagnostics. It returns a list of partitions
  whose current state matches the flag. This function returns the partition information
  without their base partitions, switches, and node cards.

  The possible flags are contained in the \texttt{rm\_api\_h} include file and are listed in Table 13-2.
  You can use OR on these values to create a flag for including partitions with different
  states.

  \textit{Table 13-2 Flags for partition states}

<table>
<thead>
<tr>
<th>Flag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARTITION_FREE_FLAG</td>
<td>0x01</td>
</tr>
<tr>
<td>PARTITION_CONFIGURING_FLAG</td>
<td>0x02</td>
</tr>
<tr>
<td>PARTITION_READY_FLAG</td>
<td>0x04</td>
</tr>
<tr>
<td>PARTITION_DEALLOCATING_FLAG</td>
<td>0x10</td>
</tr>
<tr>
<td>PARTITION_ERROR_FLAG</td>
<td>0x20</td>
</tr>
<tr>
<td>PARTITION_REBOOTING_FLAG</td>
<td>0x40</td>
</tr>
<tr>
<td>PARTITION_ALL_FLAG</td>
<td>0xFF</td>
</tr>
</tbody>
</table>

The following return codes are possible:
- STATUS_OK
- CONNECTION_ERROR
- INCONSISTENT_DATA

  At least one of the partitions has an empty base partition list.

- INTERNAL_ERROR

  \texttt{status\_t\ rm\_modify\_partition(pm\_partition\_id\_t\ partition\_id,\newline \hspace{1em} enum\ rm\_modify\_op \newline \hspace{1em} modify\_option, const\ void\ *value);}\texttt{;}

  This function makes it possible to change a set of fields in an already existing partition.
  The fields that can be modified are owner, description, options, and the partition boot
  images. The \texttt{modify\_option} parameter identifies the field to be modified. To change the
  pool ID, the partition must be in the \texttt{RM\_PARTITION\_READY} ("initialized") state.  The
  other modifiable fields require the partition to be in the \texttt{RM\_PARTITION\_FREE} ("free")
  state.

  The following return codes are possible:
- STATUS_OK
- CONNECTION_ERROR
- INVALID\_INPUT
  - partition\_id\_is NULL, or the length exceeds the limitations of the control system.
  - The value for the \texttt{modify\_option} parameter is not valid.
– PARTITION_NOT_FOUND
– INCOMPATIBLE_STATE

The partition's current state forbids its modification. See Figure 13-1 on page 207.

– INTERNAL_ERROR

status_t pm_reboot_partition(pm_partition_id_t partition_id);

This function sends a request to reboot a partition and update the resulting status in the database.

Note: This API is asynchronous. Control returns to your application before the operation requested is complete.

The following return codes are possible:
– STATUS_OK
– CONNECTION_ERROR
– INVALID_INPUT
  • partition_id is NULL, or the length exceeds the limitations of the control system.
  • This API is not supported for HTC partitions.
– PARTITION_NOT_FOUND
– INCOMPATIBLE_STATE

The partition's current state forbids it to be rebooted. See Figure 13-1 on page 207.

– INTERNAL_ERROR

status_t rm_release_partition(pm_partition_id_t partition_id);

This function is the opposite of rm_assign_job() because it releases the partition from all jobs. Only jobs that are in an RM_JOB_IDLE state have their partition reference removed.

The following return codes are possible:
– STATUS_OK
– CONNECTION_ERROR
– INVALID_INPUT
  partition_id is NULL, or the length exceeds the limitations of the control system (configuration parameter).
– PARTITION_NOT_FOUND
– INCOMPATIBLE_STATE

The current state of one or more jobs assigned to the partition prevents this release. See Figure 13-1 on page 207 and Figure 13-2 on page 212.

– INTERNAL_ERROR

status_t rm_remove_partition(pm_partition_id_t partition_id);

This function removes the specified partition record from MMCS.

The following return codes are possible:
– STATUS_OK
– CONNECTION_ERROR
– INVALID_INPUT
  partition_id is NULL, or the length exceeds the limitations of the control system (configuration parameter).
PARTITION_NOT_FOUND
INCOMPATIBLE_STATE

The partition's current state forbids its removal. See Figure 13-1 on page 207 and Figure 13-2 on page 212.

INTERNAL_ERROR

status_t rm_remove_part_user(pm_partition_id_t partition_id, const char *user);

This function removes a user from a partition. Removing a user from a partition can be done only by the partition owner. A user can be removed from a partition that is in any state. Once a HTC partition is booted, this API can still be used, but the submit server daemon running on the service node ignores any removed users. Those removed users are still allowed to run jobs on the partition.

The following return codes are possible:

STATUS_OK
CONNECTION_ERROR
INVALID_INPUT

partition_id is NULL, or the length exceeds the limitations of the control system (configuration parameter).
user is NULL, or the length exceeds the limitations of the control system.
user is already defined as the partition's user.
Current user is not the partition owner.

INTERNAL_ERROR

status_t rm_set_part_owner(pm_partition_id_t partition_id, const char *user);

This function sets the new owner of the partition. Changing the partition's owner can be done only to a partition in the RM_PARTITION_FREE state.

The following return codes are possible:

STATUS_OK
CONNECTION_ERROR
INVALID_INPUT

partition_id is NULL, or the length exceeds the limitations of the control system (configuration parameter).
owner is NULL, or the length exceeds the limitations of the control system.

INTERNAL_ERROR

status_t rm_get_htc_pool(pm_pool_id_t pid, rm_partition_list_t **p);

This function is useful for status reports and diagnostics. It returns a list of partitions whose HTC pool id matches the parameter.

The following return codes are possible:

STATUS_OK
CONNECTION_ERROR
INCONSISTENT_DATA

At least one of the partitions has an empty base partition list.

INTERNAL_ERROR
State transition diagram for partitions

Figure 13-1 illustrates the states that a partition goes through during its life cycle. For HTC partitions RM_PARTITION_REBOOTING is not a possible state.

![State transition diagram]

13.2.7 Job-related APIs

In this section, we describe the APIs to create and manage jobs in the Blue Gene system:

- `status_t rm_add_job(rm_job_t *job);`

  This function adds a job record to the database. The job structure includes an ID field that will be filled by the resource manager.

**Note:** `rm_add_job` is not supported for HTC jobs.

The following return codes are possible:

- STATUS_OK
- CONNECTION_ERROR
- INVALID_INPUT:
  - Data in the `rm_job_t` structure is invalid.
  - There is no job name, or a job name is too long.
  - There is no user, or the user name is too long.
  - There is no executable, or the executable name is too long.
  - The output or error file name is too long.
– JOB_ALREADY_DEFINED
  A job with the same name already exists.
– INTERNAL_ERROR

> status_t jm_attach_job(jobid);

This function initiates the spawn of debug servers to a job in the RM_JOB_LOADED state.

**Note:** jm_attach_job is not supported for HTC jobs.

The following return codes are possible:

– STATUS_OK
– CONNECTION_ERROR
– JOB_NOT_FOUND
– INCOMPATIBLE_STATE
  The job's state prevents it from being attached. See Figure 13-2 on page 212.
– INTERNAL_ERROR

> status_t jm_begin_job(jobid);

This function begins a job that is already loaded.

**Note:** jm_begin_job is not supported for HTC jobs.

The following return codes are possible:

– STATUS_OK
– CONNECTION_ERROR
– JOB_NOT_FOUND
– INCOMPATIBLE_STATE
  The job's state prevents it from beginning. See Figure 13-2 on page 212.
– INTERNAL_ERROR

> status_t jm_cancel_job(db_job_id_t jid);

This function sends a request to cancel the job identified by the jid parameter.

**Note:** This API is asynchronous. Control returns to your application before the operation requested is complete.

The following return codes are possible:

– STATUS_OK
– CONNECTION_ERROR
– JOB_NOT_FOUND
– INCOMPATIBLE_STATE
  The job's state prevents it from being canceled. See Figure 13-2 on page 212.
– INTERNAL_ERROR
status_t jm_debug_job(jobid);
This function initiates the spawn of debug servers to a job in the RM_JOB_RUNNING state.

**Note:** jm_debug_job is not supported for HTC.

The following return codes are possible:
- STATUS_OK
- CONNECTION_ERROR
- JOB_NOT_FOUND
- INCOMPATIBLE_STATE
  The job's state prevents it from being debugged. See Figure 13-2 on page 212.
- INTERNAL_ERROR

status_t rm_get_job(db_job_id_t jid, rm_job_t **job);
This function retrieves the specified job object.

The following return codes are possible:
- STATUS_OK
- CONNECTION_ERROR
- JOB_NOT_FOUND
- INTERNAL_ERROR

status_t rm_get_jobs(rm_job_state_flag_t flag, rm_job_list_t **job_list);
This function returns a list of jobs whose current state matches the flag.

The possible flags are contained in the rm_api.h include file and are listed in Table 13-3. You can use OR on these values to create a flag for including jobs with different states.

**Table 13-3  Flags for job states**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOB_IDLE_FLAG</td>
<td>0x001</td>
</tr>
<tr>
<td>JOB_STARTING_FLAG</td>
<td>0x002</td>
</tr>
<tr>
<td>JOB_RUNNING_FLAG</td>
<td>0x004</td>
</tr>
<tr>
<td>JOB_TERMINATED_FLAG</td>
<td>0x008</td>
</tr>
<tr>
<td>JOB_ERROR_FLAG</td>
<td>0x010</td>
</tr>
<tr>
<td>JOB_DYING_FLAG</td>
<td>0x020</td>
</tr>
<tr>
<td>JOB_DEBUG_FLAG</td>
<td>0x040</td>
</tr>
<tr>
<td>JOB_LOAD_FLAG</td>
<td>0x080</td>
</tr>
<tr>
<td>JOB_LOADED_FLAG</td>
<td>0x100</td>
</tr>
<tr>
<td>JOB_BEGIN_FLAG</td>
<td>0x200</td>
</tr>
<tr>
<td>JOB_ATTACH_FLAG</td>
<td>0x400</td>
</tr>
<tr>
<td>JOB_KILLED_FLAG</td>
<td>0x800</td>
</tr>
</tbody>
</table>
The following return codes are possible:
- STATUS_OK
- CONNECTION_ERROR
- INTERNAL_ERROR

```c
status_t jm_load_job(jobid);
```
This function sets the job state to LOAD.

**Note:** jm_load_job is not supported for HTC jobs.

The following return codes are possible:
- STATUS_OK
- CONNECTION_ERROR
- JOB_NOT_FOUND
- INCOMPATIBLE_STATE
  The job's state prevents it from being loaded. See Figure 13-2 on page 212.
- INTERNAL_ERROR

```c
status_t rm_query_job(db_job_id_t db_job_id, MPIR_PROCDESC **proc_table, int * proc_table_size);
```
This function fills the proc_table with information about the specified job.

**Note:** rm_query_job is not supported for HTC jobs.

The following return codes are possible:
- STATUS_OK
- CONNECTION_ERROR
- JOB_NOT_FOUND
- INTERNAL_ERROR

```c
status_t rm_remove_job(db_job_id_t jid);
```
This function removes the specified job record from MMCS.

**Note:** rm_remove_job is not supported for HTC jobs.

The following return codes are possible:
- STATUS_OK
- CONNECTION_ERROR
- JOB_NOT_FOUND
- INCOMPATIBLE_STATE
  The job's state prevents its removal. See Figure 13-2 on page 212.
- INTERNAL_ERROR

```c
status_t jm_signal_job(db_job_id_t jid, rm_signal_t signal);
```
This function sends a request to signal the job identified by the `jid` parameter.

**Note:** This API is asynchronous. Control returns to your application before the operation requested is complete.

The following return codes are possible:
- STATUS_OK
- CONNECTION_ERROR
- JOB_NOT_FOUND
- INCOMPATIBLE_STATE
  The job's state prevents it from being signaled.
- INTERNAL_ERROR

```c
status_t jm_start_job(db_job_id_t jid);
```
This function starts the job identified by the `jid` parameter. Note that the partition information is referenced from the job record in MMCS.

**Note:** This API is asynchronous. Control returns to your application before the operation requested is complete.

**Note:** `jm_start_job` is not supported for HTC jobs.

The following return codes are possible:
- STATUS_OK
- CONNECTION_ERROR
- INTERNAL_ERROR

```c
status_t rm_get_filtered_jobs(rm_job_filter_t query_parms, rm_job_list_t **job_list);
```
This function returns a list of jobs whose attributes or states (or both) that match the fields specified in the filter provided in the `rm_job_filter` object.

The following return codes are possible:
- STATUS_OK
- CONNECTION_ERROR
- INTERNAL_ERROR
State transition diagrams for jobs

Figure 13-2 illustrates the states that a job goes through during its life cycle. It also illustrates the order of API calls for creating, running, and canceling a job.
Figure 13-3 illustrates the main states that a job goes through when debugging a new job.
Figure 13-4 illustrates the states a job goes through when debugging an already running job.
Figure 13-5 illustrates the states that a job goes through during its life cycle in HTC mode. It also illustrates that the `submit` command is required.

13.2.8 Field specifications for the `rm_get_data()` and `rm_set_data()` APIs

In this section, we describe all the field specifications that can be used to get and set fields from various objects using the `rm_get_data()` and `rm_set_data()` APIs.

**Blue Gene object**

The Blue Gene/P object (`rm_BG_t`) represents the Blue Gene/P system. You can use this object to retrieve information and status for other components in the system, such as base partitions, node cards, I/O nodes, switches, wires, and port (see Table 13-4 on page 216). The Blue Gene object is retrieved by calling the `rm_get_BG()` API.
Table 13-4  Values retrieved from a Blue Gene object using rm_get_data()

<table>
<thead>
<tr>
<th>Description</th>
<th>Specification</th>
<th>Argument type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of a base partition (in compute nodes)</td>
<td>RM_BPsize</td>
<td>rm_size3D_t *</td>
<td></td>
</tr>
<tr>
<td>in each dimension</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Size of the machine in base partition units</td>
<td>RM_Msize</td>
<td>rm_size3D_t *</td>
<td></td>
</tr>
<tr>
<td>Number of base partitions in the machine</td>
<td>RM_BPNum</td>
<td>int *</td>
<td></td>
</tr>
<tr>
<td>First base partition in the list</td>
<td>RM_FirstBP</td>
<td>rm_BP_t **</td>
<td></td>
</tr>
<tr>
<td>Next base partition in the list</td>
<td>RM_NextBP</td>
<td>rm_BP_t **</td>
<td></td>
</tr>
<tr>
<td>Number of switches in the machine</td>
<td>RM_SwitchNum</td>
<td>int *</td>
<td></td>
</tr>
<tr>
<td>First switch in the list</td>
<td>RM_FirstSwitch</td>
<td>rm_switch_t **</td>
<td></td>
</tr>
<tr>
<td>Next switch in the list</td>
<td>RM_NextSwitch</td>
<td>rm_switch_t **</td>
<td></td>
</tr>
<tr>
<td>Number of wires in the machine</td>
<td>RM_WireNum</td>
<td>int *</td>
<td></td>
</tr>
<tr>
<td>First wire in the list</td>
<td>RM_FirstWire</td>
<td>rm_wire_t **</td>
<td></td>
</tr>
<tr>
<td>Next wire in the list</td>
<td>RM_NextWire</td>
<td>rm_wire_t **</td>
<td></td>
</tr>
</tbody>
</table>

**Base partition object**

The base partition object (rm_BP_t) represents one base partition in the Blue Gene system. The base partition object is retrieved from the Blue Gene object using either the RM_FirstBP or RM_NextBP specification. See Table 13-5.

Table 13-5  Values retrieved from a base partition object using rm_get_data()

<table>
<thead>
<tr>
<th>Description</th>
<th>Specification</th>
<th>Argument type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base partition identifier</td>
<td>RM_BPID</td>
<td>rm_bp_id_t *</td>
<td>free required</td>
</tr>
<tr>
<td>Base partition state</td>
<td>RM_BPState</td>
<td>rm_BP_state_t *</td>
<td></td>
</tr>
<tr>
<td>Sequence ID for the base partition state</td>
<td>RM_BPStateSeqID</td>
<td>rm_sequence_id_t *</td>
<td></td>
</tr>
<tr>
<td>Location of the base partition in the 3D machine</td>
<td>RM_BPLoc</td>
<td>rm_location_t *</td>
<td></td>
</tr>
<tr>
<td>Identifier of the partition associated with the base partition</td>
<td>RM_BPPartID</td>
<td>pm_partition_id_t *</td>
<td>free required. If no partition is associated, NULL is returned.</td>
</tr>
<tr>
<td>State of the partition associated with the base partition</td>
<td>RM_BPPartState</td>
<td>rm_partition_state_t *</td>
<td></td>
</tr>
</tbody>
</table>
Table 13-6 shows the values that are set in the base partition object using `rm_set_data()`.

<table>
<thead>
<tr>
<th>Description</th>
<th>Specification</th>
<th>Argument type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence ID for the state of the partition associated with the base partition</td>
<td><code>RM_BPStateSeqID</code></td>
<td><code>rm_sequence_id_t *</code></td>
<td></td>
</tr>
<tr>
<td>Flag indicating whether this base partition is being used by a small partition (smaller than a base partition)</td>
<td><code>RM_BPSDB</code></td>
<td><code>int *</code></td>
<td></td>
</tr>
<tr>
<td>Flag indicating whether this base partition is being divided into one or more small partitions</td>
<td><code>RM_BPSD</code></td>
<td><code>int *</code></td>
<td></td>
</tr>
<tr>
<td>Compute node memory size for the base partition</td>
<td><code>RM_BPComputeNodeMemory</code></td>
<td><code>rm_BP_computenode_memory_t *</code></td>
<td></td>
</tr>
<tr>
<td>Number of available node cards</td>
<td><code>RM_BPAvailableNodeCards</code></td>
<td><code>int *</code></td>
<td></td>
</tr>
<tr>
<td>Number of available I/O nodes</td>
<td><code>RM_BPNumberIONodes</code></td>
<td><code>int *</code></td>
<td></td>
</tr>
</tbody>
</table>

Table 13-6  Values set in a base partition object using `rm_set_data()`

**Node card list object**

The node card list object (`rm_nodecard_list_t`) contains a list of node card objects. The node card list object is retrieved by calling the `rm_get_nodecards()` API for a given base partition. See Table 13-7.

<table>
<thead>
<tr>
<th>Description</th>
<th>Specification</th>
<th>Argument type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of node cards in the list</td>
<td><code>RM_NodeCardListSize</code></td>
<td><code>int *</code></td>
<td></td>
</tr>
<tr>
<td>First node card in the list</td>
<td><code>RM_NodeCardListFirst</code></td>
<td><code>rm_nodecard_t **</code></td>
<td></td>
</tr>
<tr>
<td>Next node card in the list</td>
<td><code>RM_NodeCardListNext</code></td>
<td><code>rm_nodecard_t **</code></td>
<td></td>
</tr>
</tbody>
</table>

Table 13-7  Values retrieved from a node card list object using `rm_get_data()`

**Node card object**

The node card object (`rm_nodecard_t`) represents a node card within a base partition. The node card object is retrieved from the node card list object using the `RM_NodeCardListFirst` and `RM_NodeCardListNext` specifications. See Table 13-8 on page 218.
Table 13-8  Values retrieved from a node card object using rm_get_data()

<table>
<thead>
<tr>
<th>Description</th>
<th>Specification</th>
<th>Argument type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node card identifier</td>
<td>RM_NodeCardID</td>
<td>rm_nodecard_id_t    *</td>
<td>free required; possible values: N00..N15</td>
</tr>
<tr>
<td>The quadrant of the base partition where this node card is installed</td>
<td>RM_NodeCardQuarter</td>
<td>rm_quarter_t    *</td>
<td></td>
</tr>
<tr>
<td>Node card state</td>
<td>RM_NodeCardState</td>
<td>rm_nodecard_state_t    *</td>
<td></td>
</tr>
<tr>
<td>Sequence ID for the node card state</td>
<td>RM_NodeCardStateSeqID</td>
<td>rm_sequence_id_t    *</td>
<td></td>
</tr>
<tr>
<td>Number of I/O nodes on the node card (can be 0, 1, or 2)</td>
<td>RM_NodeCardIONodes</td>
<td>int    *</td>
<td></td>
</tr>
<tr>
<td>Identifier of the partition associated with the node card</td>
<td>RM_NodeCardPartID</td>
<td>rm_partition_id_t    *</td>
<td>free required. If no partition is associated, NULL is returned.</td>
</tr>
<tr>
<td>State of the partition associated with the node card</td>
<td>RM_NodeCardPartState</td>
<td>rm_partition_state_t    *</td>
<td></td>
</tr>
<tr>
<td>Sequence ID for the state of the partition associated with the node card</td>
<td>RM_NodeCardPartStateSeqID</td>
<td>rm_sequence_id_t    *</td>
<td></td>
</tr>
<tr>
<td>Flag indicating whether the node card is being used by a partition whose size is smaller than a node card</td>
<td>RM_NodeCardSDB</td>
<td>int    *</td>
<td>0=No 1=Yes</td>
</tr>
<tr>
<td>Number of I/O nodes in a list</td>
<td>RM_NodeCardIONodeNum</td>
<td>int    *</td>
<td></td>
</tr>
<tr>
<td>First I/O node in the node card</td>
<td>RM_NodeCardFirstIONode</td>
<td>rm_ionode_t   **</td>
<td></td>
</tr>
<tr>
<td>Next I/O node in the node card</td>
<td>RM_NodeCardNextIONode</td>
<td>rm_ionode_t   **</td>
<td></td>
</tr>
</tbody>
</table>

Table 13-9 shows the values that are set in a node card object when using rm_set_data().

Table 13-9  Values set in a node card object using rm_set_data()

<table>
<thead>
<tr>
<th>Description</th>
<th>Specification</th>
<th>Argument type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node card identifier</td>
<td>RM_NodeCardID</td>
<td>rm_nodecard_id_t</td>
<td></td>
</tr>
<tr>
<td>Number of I/O nodes in list</td>
<td>RM_NodeCardIONodeNum</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>First I/O node in the node card</td>
<td>RM_NodeCardFirstIONode</td>
<td>rm_ionode_t   **</td>
<td></td>
</tr>
<tr>
<td>Next I/O node in the node card</td>
<td>RM_NodeCardNextIONode</td>
<td>rm_ionode_t   **</td>
<td></td>
</tr>
</tbody>
</table>
### I/O node object

The I/O node object (rm_ionode_t) represents an I/O node within a node card. The I/O node object is retrieved from the node card object using the RM_NodeCardFirstIONode and RM_NodeCardNextIONode specifications. See Table 13-10.

<table>
<thead>
<tr>
<th>Description</th>
<th>Specification</th>
<th>Argument type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O node identifier</td>
<td>RM_IONodeID</td>
<td>rm_ionode_id_t *</td>
<td>Possible values: J00, J01; free required</td>
</tr>
<tr>
<td>Node card identifier</td>
<td>RM_IONodeNodeCardID</td>
<td>rm_nodecard_id_t *</td>
<td>Possible values: N00..N15; free required</td>
</tr>
<tr>
<td>IP address</td>
<td>RM_IONodeIPAddress</td>
<td>char **</td>
<td>free required</td>
</tr>
<tr>
<td>MAC address</td>
<td>RM_IONodeMacAddress</td>
<td>char **</td>
<td>free required</td>
</tr>
<tr>
<td>Identifier of the partition associated with the I/O node</td>
<td>RM_IONodePartID</td>
<td>pm_partition_id_t *</td>
<td>free required. If no partition is associated with this I/O node, NULL is returned.</td>
</tr>
<tr>
<td>State of the partition associated with the I/O node</td>
<td>RM_IONodePartState</td>
<td>rm_partition_state_t *</td>
<td></td>
</tr>
<tr>
<td>Sequence ID for the state of the partition associated with the I/O node</td>
<td>RM_IONodePartStateSeqID</td>
<td>rm_sequence_id_t *</td>
<td></td>
</tr>
</tbody>
</table>

Table 13-11 shows the values that are set in an I/O node object by using rm_set_data().

<table>
<thead>
<tr>
<th>Description</th>
<th>Specification</th>
<th>Argument type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O node identifier</td>
<td>RM_IONodeID</td>
<td>rm_ionode_id_t</td>
<td>Possible values: J00, J01</td>
</tr>
</tbody>
</table>

### Switch object

The switch object (rm_switch_t) represents a switch in the Blue Gene system. The switch object is retrieved from the following specifications:

- The Blue Gene object using the RM_FirstSwitch and RM_NextSwitch specifications
- The partition object using the RM_PartitionFirstSwitch and RM_PartitionNextSwitch specifications
Table 13-12 shows the values that are retrieved from a switch object using `rm_get_data()`.

<table>
<thead>
<tr>
<th>Description</th>
<th>Specification</th>
<th>Argument type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switch identifier</td>
<td>RM_SwitchID</td>
<td><code>rm_switch_id_t</code> *</td>
<td>free required</td>
</tr>
<tr>
<td>Identifier of the base partition connected to the switch</td>
<td>RM_SwitchBPID</td>
<td><code>rm_BP_id_t</code> *</td>
<td>free required</td>
</tr>
<tr>
<td>Switch state</td>
<td>RM_SwitchState</td>
<td><code>rm_switch_state_t</code> *</td>
<td></td>
</tr>
<tr>
<td>Sequence ID for the switch state</td>
<td>RM_SwitchStateSeqID</td>
<td><code>rm_sequence_id_t</code> *</td>
<td></td>
</tr>
<tr>
<td>Switch dimension</td>
<td>RM_SwitchDim</td>
<td><code>rm_dimension_t</code> *</td>
<td>Values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>▶ RM_DIM_X</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>▶ RM_DIM_Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>▶ RM_DIM_Z</td>
</tr>
<tr>
<td>Number of connections in the switch</td>
<td>RM_SwitchConnNum</td>
<td><code>int</code> *</td>
<td>A connection is a pair of ports that are connected internally in the switch.</td>
</tr>
<tr>
<td>First connection in the list</td>
<td>RM_SwitchFirstConnection</td>
<td><code>rm_connection_t</code> *</td>
<td></td>
</tr>
<tr>
<td>Next connection in the list</td>
<td>RM_SwitchNextConnection</td>
<td><code>rm_connection_t</code> *</td>
<td></td>
</tr>
</tbody>
</table>

Table 13-13 shows the values that are set in a switch object using `rm_set_data()`.

<table>
<thead>
<tr>
<th>Description</th>
<th>Specification</th>
<th>Argument type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switch identifier</td>
<td>RM_SwitchID</td>
<td><code>rm_switch_id_t</code> *</td>
<td></td>
</tr>
<tr>
<td>Number of connections in the switch</td>
<td>RM_SwitchConnNum</td>
<td><code>int</code> *</td>
<td>A connection is a pair of ports that are connected internally in the switch.</td>
</tr>
<tr>
<td>First connection in the list</td>
<td>RM_SwitchFirstConnection</td>
<td><code>rm_connection_t</code> *</td>
<td></td>
</tr>
<tr>
<td>Next connection in the list</td>
<td>RM_SwitchNextConnection</td>
<td><code>rm_connection_t</code> *</td>
<td></td>
</tr>
</tbody>
</table>

**Wire object**

The wire object (`rm_wire_t`) represents a wire in the Blue Gene/P system. The wire object is retrieved from the Blue Gene/P object using the `RM_FirstWire` and `RM_NextWire` specifications. See Table 13-14 on page 221.
Table 13-14  Values retrieved from a wire object using rm_get_data()

<table>
<thead>
<tr>
<th>Description</th>
<th>Specification</th>
<th>Argument type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wire identifier</td>
<td>RM_WireID</td>
<td>rm_wire_id_t *</td>
<td>free required.</td>
</tr>
<tr>
<td>Wire state</td>
<td>RM_WireState</td>
<td>rm_wire_state_t *</td>
<td>The state can be UP or DOWN.</td>
</tr>
<tr>
<td>Source port</td>
<td>RM_WireFromPort</td>
<td>rm_port_t **</td>
<td></td>
</tr>
<tr>
<td>Destination port</td>
<td>RM_WireToPort</td>
<td>rm_port_t **</td>
<td></td>
</tr>
<tr>
<td>Identifier of the partition associated with the wire</td>
<td>RM_WirePartID</td>
<td>pm_partition_id_t *</td>
<td>free required. If no partition is associated, NULL is returned.</td>
</tr>
<tr>
<td>State of the partition associated with the wire</td>
<td>RM_WirePartState</td>
<td>rm_partition_state_t *</td>
<td></td>
</tr>
<tr>
<td>Sequence ID for the state of the partition associated with the wire</td>
<td>RM_WirePartStateSeqID</td>
<td>rm_sequence_id_t *</td>
<td></td>
</tr>
</tbody>
</table>

**Port object**

The port object (rm_port_t) represents a port for a switch in the Blue Gene System. The port object is retrieved from the wire object using the RM_WireFromPort and RM_WireToPort specifications. See Table 13-15.

Table 13-15  Values retrieved from a port object using rm_get_data()

<table>
<thead>
<tr>
<th>Description</th>
<th>Specification</th>
<th>Argument type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier of the base partition or switch associated with the port</td>
<td>RM_PortComponentID</td>
<td>rm_component_id_t *</td>
<td>free required</td>
</tr>
<tr>
<td>Port identifier</td>
<td>RM_PortID</td>
<td>rm_port_id_t *</td>
<td>Possible values for base partitions: plus_x minus_x, plus_y minus_y, plus_z minus_z. Possible values for switches: s0...S5</td>
</tr>
</tbody>
</table>

**Partition list object**

The partition list object (rm_partition_list_t) contains a list of partition objects. The partition list object is retrieved by calling the rm_get_partitions() or rm_get_partitions_info() APIs. See Table 13-16.

Table 13-16  Values retrieved from a partition list object using rm_get_data()

<table>
<thead>
<tr>
<th>Description</th>
<th>Specification</th>
<th>Argument type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of partitions in the list</td>
<td>RM_PartListSize</td>
<td>int *</td>
<td></td>
</tr>
<tr>
<td>First partition in the list</td>
<td>RM_PartListFirstPart</td>
<td>rm_partition_t **</td>
<td></td>
</tr>
<tr>
<td>Next partition in the list</td>
<td>RM_PartListNextPart</td>
<td>rm_partition_t **</td>
<td></td>
</tr>
</tbody>
</table>

**Partition object**

The partition object (rm_partition_t) represents a partition that is defined in the Blue Gene system. The partition object is retrieved from the partition list object using the RM_PartListFirstPart and RM_PartListNextPart specifications. A new partition object is created using the rm_new_partition() API. After setting the appropriate fields in a new partition object, the partition can be added to the system using the rm_add_partition() API. See Table 13-17 on page 222.
<table>
<thead>
<tr>
<th>Description</th>
<th>Specification</th>
<th>Argument type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition identifier</td>
<td>RM_PartitionID</td>
<td>pm_partition_id_t *</td>
<td>free required</td>
</tr>
<tr>
<td>Partition state</td>
<td>RM_PartitionState</td>
<td>rm_partition_state_t *</td>
<td></td>
</tr>
<tr>
<td>Sequence ID for the partition state</td>
<td>RM_PartitionStateSeqID</td>
<td>rm_sequence_id_t *</td>
<td></td>
</tr>
<tr>
<td>Connection type of the partition</td>
<td>RM_PartitionConnection</td>
<td>rm_connection_type_t *</td>
<td>Values: TORUS or MESH</td>
</tr>
<tr>
<td>Partition description</td>
<td>RM_PartitionDescription</td>
<td>char **</td>
<td>free required</td>
</tr>
<tr>
<td>Flag indicating whether this partition is a</td>
<td>RM_PartitionSmall</td>
<td>int *</td>
<td>0=No&lt;br&gt;1=Yes</td>
</tr>
<tr>
<td>partition smaller than the base partition</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of used processor sets (psets) per base</td>
<td>RM_PartitionPsetsPerBP</td>
<td>int *</td>
<td></td>
</tr>
<tr>
<td>Job identifier of the current job</td>
<td>RM_PartitionJobID</td>
<td>int *</td>
<td>If no job is currently on the partition, 0 is returned; for HTC partitions it always returns 0 even when HTC jobs are running.</td>
</tr>
<tr>
<td>Partition owner</td>
<td>RM_PartitionUserName</td>
<td>char **</td>
<td>free required</td>
</tr>
<tr>
<td>Partition options</td>
<td>RM_PartitionOptions</td>
<td>char **</td>
<td>free required</td>
</tr>
<tr>
<td>File name of the machine loader image</td>
<td>RM_PartitionMloaderImg</td>
<td>char **</td>
<td>free required</td>
</tr>
<tr>
<td>Comma-separated list of images to load on the</td>
<td>RM_PartitionCnloadImg</td>
<td>char **</td>
<td>free required</td>
</tr>
<tr>
<td>compute nodes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comma-separated list of images to load on the I/O</td>
<td>RM_PartitionIoloadImg</td>
<td>char **</td>
<td>free required</td>
</tr>
<tr>
<td>nodes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of base partitions in the partition</td>
<td>RM_PartitionBPNum</td>
<td>int *</td>
<td></td>
</tr>
<tr>
<td>First base partition in the partition</td>
<td>RM_PartitionFirstBP</td>
<td>rm_BP_t **</td>
<td></td>
</tr>
<tr>
<td>Next base partition in the partition</td>
<td>RM_PartitionNextBP</td>
<td>rm_BP_t **</td>
<td></td>
</tr>
<tr>
<td>Number of switches in the partition</td>
<td>RM_PartitionSwitchNum</td>
<td>int *</td>
<td></td>
</tr>
<tr>
<td>First switch in the partition</td>
<td>RM_PartitionFirstSwitch</td>
<td>rm_switch_t **</td>
<td></td>
</tr>
<tr>
<td>Next switch in the partition</td>
<td>RM_PartitionNextSwitch</td>
<td>rm_switch_t **</td>
<td></td>
</tr>
<tr>
<td>Number of node cards in the partition</td>
<td>RM_PartitionNodeCardNum</td>
<td>int *</td>
<td></td>
</tr>
<tr>
<td>First node card in the partition</td>
<td>RM_PartitionFirstNodeCard</td>
<td>rm_nodecard_t **</td>
<td></td>
</tr>
<tr>
<td>Next node card in the partition</td>
<td>RM_PartitionNextNodeCard</td>
<td>rm_nodecard_t **</td>
<td></td>
</tr>
<tr>
<td>Number of users of the partition</td>
<td>RM_PartitionUsersNum</td>
<td>int *</td>
<td></td>
</tr>
<tr>
<td>First user name for the partition</td>
<td>RM_PartitionFirstUser</td>
<td>char **</td>
<td>free required</td>
</tr>
</tbody>
</table>
Table 13-18 shows the values that are set in a partition object using `rm_set_data()`.

<table>
<thead>
<tr>
<th>Description</th>
<th>Specification</th>
<th>Argument type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Next user name for the partition</td>
<td><code>RM_PartitionNextUser</code></td>
<td><code>char **</code></td>
<td>free required</td>
</tr>
<tr>
<td>HTC pool identifier</td>
<td><code>RM_PartitionHTCPoolID</code></td>
<td><code>pm_pool_id_t</code></td>
<td>Value will be <code>NULL</code> for a HPC partition. free required</td>
</tr>
<tr>
<td>Partition size in compute nodes</td>
<td><code>RM_PartitionSize</code></td>
<td><code>int *</code></td>
<td></td>
</tr>
</tbody>
</table>

**Table 13-18  Values set in a partition object using `rm_set_data()`**

<table>
<thead>
<tr>
<th>Description</th>
<th>Specification</th>
<th>Argument type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition identifier</td>
<td><code>RM_PartitionID</code></td>
<td><code>pm_partition_id_t</code></td>
<td></td>
</tr>
<tr>
<td>Connection type of the partition</td>
<td><code>RM_PartitionConnection</code></td>
<td><code>rm_connection_type_t</code></td>
<td>Values: TORUS or MESH</td>
</tr>
<tr>
<td>Partition description</td>
<td><code>RM_PartitionDescription</code></td>
<td><code>char *</code></td>
<td></td>
</tr>
<tr>
<td>Flag indicating whether this partition is</td>
<td><code>RM_PartitionSmall</code></td>
<td><code>int *</code></td>
<td>0=No 1=Yes</td>
</tr>
<tr>
<td>a partition smaller than the base partition</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of used processor sets (psets) per base partition</td>
<td><code>RM_PartitionPsetsPerBP</code></td>
<td><code>int *</code></td>
<td></td>
</tr>
<tr>
<td>Partition owner</td>
<td><code>RM_PartitionUserName</code></td>
<td><code>char *</code></td>
<td></td>
</tr>
<tr>
<td>File name of the machine loader image</td>
<td><code>RM_PartitionMloaderImg</code></td>
<td><code>char *</code></td>
<td></td>
</tr>
<tr>
<td>Comma-separated list of images to load on the compute nodes</td>
<td><code>RM_PartitionCnloadImg</code></td>
<td><code>char *</code></td>
<td></td>
</tr>
<tr>
<td>Comma-separated list of images to load on the I/O nodes</td>
<td><code>RM_PartitionIoloadImg</code></td>
<td><code>char *</code></td>
<td></td>
</tr>
<tr>
<td>Number of base partitions in the partition</td>
<td><code>RM_PartitionBPNum</code></td>
<td><code>int *</code></td>
<td></td>
</tr>
<tr>
<td>First base partition in the partition</td>
<td><code>RM_PartitionFirstBP</code></td>
<td><code>rm_BP_t</code></td>
<td></td>
</tr>
<tr>
<td>Next base partition in the partition</td>
<td><code>RM_PartitionNextBP</code></td>
<td><code>rm_BP_t</code></td>
<td></td>
</tr>
<tr>
<td>Number of switches in the partition</td>
<td><code>RM_PartitionSwitchNum</code></td>
<td><code>int *</code></td>
<td></td>
</tr>
<tr>
<td>First switch in the list in the partition</td>
<td><code>RM_PartitionFirstSwitch</code></td>
<td><code>rm_switch_t</code></td>
<td></td>
</tr>
<tr>
<td>Next switch in the partition</td>
<td><code>RM_PartitionNextSwitch</code></td>
<td><code>rm_switch_t</code></td>
<td></td>
</tr>
</tbody>
</table>
Job list object
The job list object (rm_job_list_t) contains a list of job objects. The job list object is retrieved by calling the rm_get_jobs() API. See Table 13-19.

<table>
<thead>
<tr>
<th>Description</th>
<th>Specification</th>
<th>Argument type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of node cards in the</td>
<td>RM_PartitionNodeCardNum</td>
<td>int *</td>
<td></td>
</tr>
<tr>
<td>partition</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>First node card in the partition</td>
<td>RM_PartitionFirstNodecard</td>
<td>rm_nodecard_t *</td>
<td></td>
</tr>
<tr>
<td>Next node card in the partition</td>
<td>RM_PartitionNextNodecard</td>
<td>rm_nodecard_t *</td>
<td></td>
</tr>
</tbody>
</table>

Table 13-19  Values retrieved from a job list object using rm_get_data()

Job object
The job object (rm_job_t) represents a job defined in the Blue Gene system. The job object is retrieved from the job list object using the RM_JobListFirstJob and RM_JobListNextJob specifications. A new job object is created using the rm_new_job() API. After setting the appropriate fields in a new job object, the job can be added to the system using the rm_add_job() API. See Table 13-20.

<table>
<thead>
<tr>
<th>Description</th>
<th>Specification</th>
<th>Argument type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job identifier</td>
<td>RM_JobID</td>
<td>rm_job_id_t *</td>
<td>free required</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Identifier is unique across all jobs on the system.</td>
</tr>
<tr>
<td>Identifier of the partition</td>
<td>RM_JobPartitionID</td>
<td>pm_partition_id_t *</td>
<td>free required</td>
</tr>
<tr>
<td>assigned for the job</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Job state</td>
<td>RM_JobState</td>
<td>rm_job_state_t *</td>
<td>free required</td>
</tr>
<tr>
<td>Sequence ID for the job state</td>
<td>RM_JobStateSeqID</td>
<td>rm_sequence_id_t *</td>
<td>free required</td>
</tr>
<tr>
<td>Executable file name for the</td>
<td>RM_JobExecutable</td>
<td>char **</td>
<td>free required</td>
</tr>
<tr>
<td>job</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name of the user who submitted</td>
<td>RM_JobUserName</td>
<td>char **</td>
<td>free required</td>
</tr>
<tr>
<td>the job</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer containing the ID</td>
<td>RM_JobDBJobID</td>
<td>db_job_id_t *</td>
<td>free required</td>
</tr>
<tr>
<td>given to the job by the</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>database</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Job output file name</td>
<td>RM_JobOutFile</td>
<td>char **</td>
<td>free required</td>
</tr>
<tr>
<td>Job error file name</td>
<td>RM_JobErrFile</td>
<td>char **</td>
<td>free required</td>
</tr>
<tr>
<td>Job output directory name</td>
<td>RM_JobOutDir</td>
<td>char **</td>
<td>free required</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>This directory contains the output files if a full path is not given.</td>
</tr>
<tr>
<td>Description</td>
<td>Specification</td>
<td>Argument type</td>
<td>Notes</td>
</tr>
<tr>
<td>-----------------------------------------------------------------------------</td>
<td>----------------------</td>
<td>----------------------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>Error text returned from the control daemons</td>
<td>RM_JobErrText</td>
<td>char **</td>
<td>free required</td>
</tr>
<tr>
<td>Arguments for the job executable</td>
<td>RM_JobArgs</td>
<td>char **</td>
<td>free required</td>
</tr>
<tr>
<td>Environment parameter needed for the job</td>
<td>RM_JobEnvs</td>
<td>char **</td>
<td>free required</td>
</tr>
<tr>
<td>Flag indicating whether the job was retrieved from the history table</td>
<td>RM_JobInHist</td>
<td>int *</td>
<td>0=No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1=Yes</td>
</tr>
<tr>
<td>Job mode</td>
<td>RM_JobMode</td>
<td>rm_job_mode_t *</td>
<td>Indicates virtual node, SMP, or dual mode</td>
</tr>
<tr>
<td>System call trace indicator for compute nodes</td>
<td>RM_JobStrace</td>
<td>rm_job_strace_t *</td>
<td></td>
</tr>
<tr>
<td>Job start time</td>
<td>RM_JobStartTime</td>
<td>char **</td>
<td>free required</td>
</tr>
<tr>
<td>The format is yyyy-mm-dd-hh.mm.ss.nnnnnn. If the job never goes to running</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>state, it will be an empty string. Data is only valid for completed jobs.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The rm_get_data() specification RM_JobInHist can be used to determine</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>whether a job has completed. If the job is an active job, then the value</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>returned is meaningless.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Job end time</td>
<td>RM_JobEndTime</td>
<td>char **</td>
<td>free required</td>
</tr>
<tr>
<td>Format is yyyy-mm-dd-hh.mm.ss.nnnnnn. Data is valid only for completed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jobs. The rm_get_data() specification RM_JobInHist can be used to determine</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>whether a job has completed. If the job is an active job, the value</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>returned is meaningless.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Job run time in seconds</td>
<td>RM_JobRunTime</td>
<td>rm_job_runtime_t *</td>
<td></td>
</tr>
<tr>
<td>Data is only valid for completed jobs. The rm_get_data() specification</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RM_JobInHist can be used to determine whether a job has completed. If the</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>job is an active job, the value returned is meaningless.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of compute nodes used by the job</td>
<td>RM_JobComputeNodesUsed</td>
<td>rm_job_computenodes_used_t</td>
<td></td>
</tr>
</tbody>
</table>
Table 13-21 shows the values that are set in a job object using `rm_set_data()`.

<table>
<thead>
<tr>
<th>Description</th>
<th>Specification</th>
<th>Argument type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job exit status</td>
<td><code>RM_JobExitStatus</code></td>
<td><code>rm_job_exitstatus_t *</code></td>
<td>Data is only valid for completed jobs. The <code>rm_get_data()</code> specification <code>RM_JobInHist</code> can be used to determine whether a job has completed. If the job is an active job, the value returned is meaningless.</td>
</tr>
<tr>
<td>User UID</td>
<td><code>RM_JobUserUid</code></td>
<td><code>rm_job_user_uid_t *</code></td>
<td>Zero is returned when querying existing jobs.</td>
</tr>
<tr>
<td>User GID</td>
<td><code>RM_JobUserGid</code></td>
<td><code>rm_job_user_gid_t *</code></td>
<td>Zero is returned when querying existing jobs.</td>
</tr>
<tr>
<td>Job location</td>
<td><code>RM_JobLocation</code></td>
<td><code>rm_job_location_t *</code></td>
<td>If NULL value, then job is HPC job. Non-NULL value indicates the location of the HTC job and is of the form Rxx-Mx-Nxx-Jxx-Cxx (where C-xx is the processor core). Free required.</td>
</tr>
<tr>
<td>Pool ID assigned for the job</td>
<td><code>RM_JobPooID</code></td>
<td><code>pm_pool_id_t *</code></td>
<td>If NULL value then job is HPC job. Non-NULL value indicates the partition pool that the job is assigned to. Free required.</td>
</tr>
<tr>
<td>Job identifier</td>
<td><code>RM_JobID</code></td>
<td><code>rm_job_id_t</code></td>
<td>This must be unique across all jobs on the system; if not, return code <code>JOB_ALREADY_DEFINED</code> is returned.</td>
</tr>
<tr>
<td>Partition identifier assigned for the job</td>
<td><code>RM_JobPartitionID</code></td>
<td><code>pm_partition_id_t</code></td>
<td>This field can be left blank when adding a new job to the system.</td>
</tr>
<tr>
<td>Executable file name for the job</td>
<td><code>RM_JobExecutable</code></td>
<td><code>char *</code></td>
<td></td>
</tr>
<tr>
<td>Name of the user who submitted the job</td>
<td><code>RM_JobUserName</code></td>
<td><code>char *</code></td>
<td></td>
</tr>
<tr>
<td>Job output file name</td>
<td><code>RM_JJobOutFile</code></td>
<td><code>char *</code></td>
<td></td>
</tr>
<tr>
<td>Job error file name</td>
<td><code>RM_JJobErrFile</code></td>
<td><code>char *</code></td>
<td></td>
</tr>
<tr>
<td>Job output directory</td>
<td><code>RM_JJobOutDir</code></td>
<td><code>char *</code></td>
<td>This directory contains the output files if a full path is not given.</td>
</tr>
<tr>
<td>Arguments for the job executable</td>
<td><code>RM_JJobArgs</code></td>
<td><code>char *</code></td>
<td></td>
</tr>
</tbody>
</table>
**Job filter object**

The job filter object (rm_job_filter_t) represents a filter for job+s defined in the Blue Gene system. The job filter object is passed as a parameter to the rm_get_filtered_jobs() API. The jobs returned match all of the specified filter fields. See Table 13-22.

<table>
<thead>
<tr>
<th>Description</th>
<th>Specification</th>
<th>Argument type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Environment parameter needed for the job</td>
<td>RM_JobEnvs</td>
<td>char *</td>
<td></td>
</tr>
<tr>
<td>Job mode</td>
<td>RM_JobMode</td>
<td>rm_job_mode_t *</td>
<td>Possible values: virtual node, SMP, or dual mode.</td>
</tr>
<tr>
<td>System call trace indicator for compute nodes</td>
<td>RM_JobStrace</td>
<td>rm_job_strace_t *</td>
<td></td>
</tr>
<tr>
<td>User UID</td>
<td>RM_JobUserUid</td>
<td>rm_job_user_uid_t *</td>
<td>This value can be set when adding a job.</td>
</tr>
<tr>
<td>User GID</td>
<td>RM_JobUserGid</td>
<td>rm_job_user_gid_t *</td>
<td>This value can be set when adding a job.</td>
</tr>
</tbody>
</table>

**Table 13-22: Job filter object description for rm_job_filter_t**

<table>
<thead>
<tr>
<th>Description</th>
<th>Specification</th>
<th>Argument Type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job identifier</td>
<td>RM_JobFilterID</td>
<td>rm_job_id_t</td>
<td>ID is unique across all jobs on the system. free required</td>
</tr>
<tr>
<td>Partition identifier assigned for the job</td>
<td>RM_JobFilterPartitionID</td>
<td>pm_partition_id_t *</td>
<td>free required.</td>
</tr>
<tr>
<td>Job state</td>
<td>RM_JobFilterState</td>
<td>rm_job_state_t *</td>
<td></td>
</tr>
<tr>
<td>Executable file name for the job</td>
<td>RM_JobFilterExecutable</td>
<td>char**</td>
<td>free required.</td>
</tr>
<tr>
<td>Name of the user who submitted the job</td>
<td>RM_JobFilterUserName</td>
<td>char**</td>
<td>free required.</td>
</tr>
<tr>
<td>Integer containing the ID given to the job by the database</td>
<td>RM_JobFilterDBJobID</td>
<td>db_job_id_t</td>
<td></td>
</tr>
<tr>
<td>Job output directory name</td>
<td>RM_JobFilterOutDir</td>
<td>char**</td>
<td>This directory contains the output files if a full path is not given. free required.</td>
</tr>
<tr>
<td>Job mode</td>
<td>RM_JobFilterMode</td>
<td>rm_job_mode_t *</td>
<td>Indicates Virtual node, SMP, or dual mode.</td>
</tr>
<tr>
<td>Job start time</td>
<td>RM_JobFilterStartTime</td>
<td>char**</td>
<td>free required.</td>
</tr>
</tbody>
</table>

Format is yyyy-mm-dd-hh.mm.ss.nnnn.
Table 13-23 shows the Job modes.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RM_SMP_MODE</td>
<td>0x0000</td>
</tr>
<tr>
<td>RM_DUAL_MODE</td>
<td>0x0001</td>
</tr>
<tr>
<td>RM_VIRTUAL_NODE_MODE</td>
<td>0x0002</td>
</tr>
</tbody>
</table>

Table 13-24 shows Type modes.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOB_TYPE_HPC_FLAG</td>
<td>0x0001</td>
</tr>
<tr>
<td>JOB_TYPE_HTC_FLAG</td>
<td>0x0002</td>
</tr>
<tr>
<td>JOB_TYPE_ALL_FLAG</td>
<td>0x0003</td>
</tr>
</tbody>
</table>

### 13.2.9 Object allocator APIs

In this section, we describe the APIs used to allocate memory for objects used with other API calls:

- `status_t rm_new_BP(rm_BP_t **bp);`
  Allocates storage for a new base partition object.
- `status_t rm_new_ionode(rm_ionode_t **io);`
  Allocates storage for a new I/O node object.
- `status_t rm_new_job(rm_job_t **job);`
  Allocates storage for a new job object.
13.2.10 Object deallocator APIs

In this section, we describe the APIs used to deallocate memory for objects that are created by other API calls:

- `status_t rm_free_BG(rm_BG_t *bg);`
  Frees storage for a Blue Gene object.
- `status_t rm_free_BP(rm_BP_t *bp);`
  Frees storage for a base partition object.
- `status_t rm_free_ionode(rm_ionode_t *io);`
  Frees storage for an I/O node object.
- `status_t rm_free_job(rm_job_t *job);`
  Frees storage for a job object.
- `status_t rm_free_job_list(rm_job_list_t *job_list);`
  Frees storage for a job list object.
- `status_t rm_free_nodecard(rm_nodecard_t *nc);`
  Frees storage for a node card object.
- `status_t rm_free_nodecard_list(rm_nodecard_list_t *nc_list);`
  Frees storage for a node card list object.
- `status_t rm_free_partition(rm_partition_t *partition);`
  Frees storage for a partition object.
- `status_t rm_free_partition_list(rm_partition_list_t *part_list);`
  Frees storage for a partition list object.
- `status_t rm_free_switch(rm_switch_t *switch);`
  Frees storage for a switch object.
- `status_t rm_free_job_filter(rm_job_filter_t *jobfilter);`
  Frees storage for a job filter object.
13.2.11 Messaging APIs

In this section, we describe the set of thread-safe messaging APIs. These APIs are used by the Bridge as well as by other components of the job management system, such as the `mpirun` program that ships with the Blue Gene software. Each message is written using the following format:

<Timestamp> Component (Message type): Message text

Here is an example:

<Mar 9 04:24:30> BRIDGE (Debug): rm_get_BG()- Completed Successfully

The message can be one of the following types:

- **MESSAGE_ERROR**: Error messages
- **MESSAGE_WARNING**: Warning messages
- **MESSAGE_INFO**: Informational messages
- **MESSAGE_DEBUG1**: Basic debug messages
- **MESSAGE_DEBUG2**: More detailed debug messages
- **MESSAGE_DEBUG3**: Very detailed debug messages

The following verbosity levels, to which the messaging APIs can be configured, define the policy:

- Level 0: Only error or warning messages are issued.
- Level 1: Level 0 messages and informational messages are issued.
- Level 2: Level 1 messages and basic debug messages are issued.
- Level 3: Level 2 messages and more debug messages are issued.
- Level 4: The highest verbosity level. All messages that will be printed are issued.

By default, only error and warning messages are written. To have informational and minimal debug messages written, set the verbosity level to 2. To obtain more detailed debug messages, set the verbosity level to 3 or 4.

In the following list, we describe the messaging APIs:

- **int isSayMessageLevel(message_type_t m_type);**
  Tests the current messaging level. Returns 1 if the specified message type is included in the current messaging level; otherwise returns 0.

- **void closeSayMessageFile();**
  Closes the messaging log file.

- **int sayFormattedMessage(FILE * curr_stream, const void * buf, size_t bytes);**
  Logs a preformatted message to the messaging output without a time stamp.

- **void sayMessage(const char * component, message_type_t m_type, const char * curr_func, const char * format, ...);**
  Logs a message to the messaging output.
  The format parameter is a format string that specifies how subsequent arguments are converted for output. This value must be compatible with printf format string requirements.

- **int sayPlainMessage(FILE * curr_stream, const char * format, ...);**
  Logs a message to the messaging output without a time stamp.

Note: Any messaging output after calling this method is sent to stderr.
The format parameter is a format string that specifies how subsequent arguments are converted for output. This value must be compatible with the printf format string requirements.

- void setSayMessageFile(const char* oldfilename, const char* newfilename);
  Opens a new file for message logging.

*Note:* This method can be used to atomically rotate log files.

- void setSayMessageLevel(unsigned int level);
  Sets the messaging verbose level.

- void setSayMessageParams(FILE * stream, unsigned int level);
  Uses the provided file for message logging and sets the logging level.

*Note:* This method has been deprecated in favor of the setSayMessageFile() and setSayMessageLevel() methods.

### 13.3 Small partition allocation

The base allocation unit in the Blue Gene/P system is a base partition. Partitions are composed of whole numbers of base partitions, except in two special cases concerning small partitions. A small partition is a partition that is comprised of a fraction of a base partition. Small partitions can be created in the following sizes:

- **16 compute nodes**
  A 16-node partition is comprised of 16 compute nodes from a single node card. The node card must have two installed I/O nodes in order to be used for a 16-node partition.

- **32 compute nodes**
  A 32-node partition is comprised of all the compute nodes in a single node card. The node card must have at least one installed I/O node in order to be used for a 32-node partition.

- **64 compute nodes**
  A 64-node partition is comprised of two adjacent node cards beginning with N00, N02, N04, N06, N08, N10, N12, or N14. The first node card in the pair must have at least one installed I/O node in order to be used for a 64-node partition.

- **128 compute nodes**
  A 128-node partition is comprised of set of four adjacent node cards beginning with N00, N04, N08, or N12. The first node card in the set must have at least one installed I/O node in order to be used for a 128-node partition.

- **256 compute nodes**
  A 256-node partition is comprised of a set of eight adjacent node cards beginning with N00 or N08. The first node card in the set must have at least one installed I/O node in order to be used for a 256-node partition.
13.3.1 Subdivided busy base partitions

It is important that you understand the concept of subdivided busy base partitions when working with small partitions. A base partition is considered subdivided busy if at least one partition, defined for a subset of its node cards, is busy. A partition is busy if its state is not free (RM_PARTITION_FREE).

A base partition that is subdivided busy cannot be booted as a whole because some of its hardware is unavailable. A base partition can have small partitions and full midplane partitions (multiples of 512 compute nodes) defined for it in the database. If the base partition has small partitions defined, they do not have to be in use, and a full midplane partition can use the actual midplane. In this case, the partition name that is using the base partition is returned on the RM_BPPartID specification.

For small partitions, multiple partitions can use the same base partition. This is the subdivided busy (SDB) example. In this situation, the value returned for the RM_BPPartID specification is meaningless. You must use the RM_BPSDB specification to determine whether the base partition is subdivided busy (small partition in use).

13.4 API examples

In this section, we provide example API calls for several common situations.

13.4.1 Retrieving base partition information

The code in Example 13-2 retrieves the Blue Gene/P hardware information and prints some information about each base partition in the system.

Example 13-2 Retrieving base partition information

```c
#include "rm_api.h"
int main(int argc, char *argv[]) {
    status_t rmrc;
    rm_BG_t *rmbg;
    int bpNum;
    enum rm_specification getOption;
    rm_BP_t *rmbp;
    rm_bp_id_t bpid;
    rm_BP_state_t state;
    rm_location_t loc;
    rmrc = rm_set_serial("BGP");
    rmrc = rm_get_BG(&rmbg);
    if (rmrc) {
        printf("Error occured calling rm_get_BG: %d\n", rmrc);
        return -1;
    }
    rm_get_data(rmbg, RM_BPNum, &bpNum);
    printf("Number of base partitions: %d\n", bpNum);
    getOption = RM_FirstBP;
    for (int ii = 0; ii < bpNum; ++ii) {
        rm_get_data(rmbg, getOption, &rmbp);
        rm_get_data(rmbp, RM_BPID, &bpid);
        rm_get_data(rmbp, RM_BPState, &state);
        rm_get_data(rmbp, RM_BPLoc, &loc);
        printf(" BP %s with state %d at location <%d,%d,%d>\n", bpid, state, loc.X, loc.Y, loc.Z);
        free(bpid);
```
getOption = RM_NextBP;
}
rm_free_BG(rmbg);  // Deallocate memory from rm_get_BG()
}

The example code can be compiled and linked with the commands shown in Figure 13-6.

```
g++ -m64 -pthread -I/bgsys/drivers/ppcfloor/include -c sample1.cc -o sample1.o_64
g++ -m64 -pthread -o sample1 sample1.o_64 -L/bgsys/drivers/ppcfloor/lib64 -lbgpbridge
```

**Figure 13-6  Example compile and link commands**

### 13.4.2 Retrieving node card information

The code in Example 13-3 shows how to retrieve information about the node cards for a base partition. The `rm_get_nodecards()` function retrieves a list of all the node cards in a base partition. The list always contains exactly 16 node cards.

**Example 13-3 Retrieving node card information**

```c
int getNodeCards(rm_bp_id_t bpid) {
    int rmrc;
    rm_nodecard_list_t *ncList;
    int ncNum;
    enum rm_specification getOption;
    rm_nodecard_t *rmnc;
    rm_nodecard_id_t ncid;
    rm_nodecard_state_t ncState;
    int ioNum;
    rmrc = rm_get_nodecards(bpid, &ncList);
    if (rmrc) {
        printf("Error occurred calling rm_get_nodecards: %d\n", rmrc);
        return -1;
    }
    rmrc = rm_get_data(ncList, RM_NodeCardListSize, &ncNum);
    printf("Base partition %s has %d nodecards\n", bpid, ncNum);
    getOption = RM_NodeCardListFirst;
    for (int ii = 0; ii < ncNum; ++ii) {
        rmrc = rm_get_data(ncList, getOption, &rmnc);
        rmrc = rm_get_data(rmnc, RM_NodeCardID, &ncid);
        rmrc = rm_get_data(rmnc, RM_NodeCardState, &ncState);
        rmrc = rm_get_data(rmnc, RM_NodeCardIONodes, &ioNum);
        printf("Node card %s with state %d has %d I/O nodes\n", ncid, ncState, ioNum);
        free(ncid);
        getOption = RM_NodeCardListNext;
    }
    rm_free_nodecard_list(ncList);
}
```
13.4.3 Defining a new small partition

Example 13-4 contains pseudo code that shows how to allocate a new small partition.

Example 13-4 Allocating a new small partition

```c
int isSmall = 1;
rm_new_partition(&newpart); //Allocate space for new partition

// Set the descriptive fields
rm_set_data(newpart, RM_PartitionUserName, username);
rm_set_data(newpart, RM_PartitionMloaderImg, BGP_MLOADER_IMAGE);
rm_set_data(newpart, RM_PartitionCnloadImg, BGP_CNLOAD_IMAGE);
rm_set_data(newpart, RM_PartitionIoloadImg, BGP_IOLOAD_IMAGE);
rm_set_data(newpart, RM_PartitionSmall, &isSmall); // Mark partition as a small partition

// Add a single BP
rm_new_BP(rm_BP_t **BP);
r_set_data(BP, RM_BPID, "R01-M0");
r_set_data(newpart, RM_PartitionFirstBP, BP);

// Add the node card(s) comprising the partition
ncNum = 4; // The number of node cards is 4 for 128 compute nodes
rm_set_data(newpart, RM_PartitionNodeCardNum, &ncNum); // Set the number of node cards
for (1 to ncNum) {
    // all four node cards must belong to same quarter!
    rm_new_nodecard(rm_nodecard_t **nc); // Allocate space for new node card
    rm_set_data(nc, RM_NodeCardID, ncid);
r_set_data(newpart, RM_PartitionFirstNodeCard, nc); // Add the node card to the
    or
    rm_set_data(newpart, RM_PartitionNextNodeCard, nc);
    rm_free_nodecard(nc);
}
rm_add_partition(newpart);
```

13.4.4 Querying a small partition

Example 13-5 contains pseudo code that shows how to query a small partition for its node cards.

Example 13-5 Querying a small partition

```c
rm_get_partition(part_id, &mypart); // Get the partition
rm_get_data(mypart, RM_PartitionSmall, &small); // Check if this is a “small” partition
if (small) {
    rm_get_data(mypart, RM_PartitionFirstBP, &BP); // Get the First (and only) BP
    rm_get_data(mypart, RM_PartitionNodeCardNum, &nc_num); // Get the number of node cards
    for (1 to nc_num) {
        rm_get_data(mypart, RM_PartitionFirstNodeCard, &nc); // Get the id
        or
        rm_get_data(nc, RM_NodeCardID, &ncid); // Get the quarter
        rm_get_data(nc, RM_NodeCardState, &state); // Get the state
```
rm_get_data(nc, RM_NodeCardIONodes, &ionodes); // Get num of I/O nodes
rm_get_data(nc, RM_NodeCardPartID, &partid); // Get the partition ID
rm_get_data(nc, RM_NodeCardPartState, &partstate); // Get the partition state

print node card information
}
Real-time Notification APIs

With the Blue Gene/P system, two programming models can handle state transitions for jobs, blocks, base partitions, switches, and node cards. The first model is based on a polling model, where the Bridge API caller is responsible for the continuous polling of state information. The second model consists of Real-time Notification APIs that allow callers to register for state transition event notifications.

The Real-time Notification APIs are designed to eliminate the need for a resource management system to constantly have to read in all of the machine state in order to detect changes. The APIs enable the caller to be notified in real time of state changes to jobs, blocks, and hardware, such as base partitions, switches, and node cards. After a resource management application has obtained an initial snapshot of the machine state using the Bridge APIs, the Bridge APIs can then determine to be notified only of changes, and the Real-time Notification APIs provide that mechanism.

In this chapter, we describe the thread-safe Real-time Notification APIs for the Blue Gene/P system that can be used by a resource management application. We discuss the following specific topics:

- API support
- Real-time callback functions
- Real-time Notification API status codes
- Sample real-time application code
14.1 API support overview

In the following sections, we present an overview of the support provided by the APIs.

Note: Real-time Notification APIs are not support in HTC mode

14.1.1 Requirements

The several requirements for writing programs to the Real-time Notification APIs are as follows:

- Currently, SUSE Linux Enterprise Server (SLES) 10 for PowerPC is the only supported platform.
- When the application calls `rt_init`, the API looks for the `DB_PROPERTY` environment variable. The corresponding `db.properties` file indicates the port on which the real-time server is listening and that the real-time client will use to connect to the server. The environment variable should be set to point to the actual `db.properties` file location as follows:
  - On a bash shell
    ```
    export DB_PROPERTY=/bgsys/drivers/ppcfloor/bin/db.properties
    ```
  - On a csh shell
    ```
    setenv DB_PROPERTY /bgsys/drivers/ppcfloor/bin/db.properties
    ```
- `C` and `C++` are supported with the GNU gcc 4.1.2 level compilers. For more information and downloads, refer to the following Web address:
  ```
  http://gcc.gnu.org/
  ```
- The include file is `/bgsys/drivers/ppcfloor/include/rt_api.h`.
- Only 64-bit shared library support is provided. Link your real-time application with the file `/bgsys/drivers/ppcfloor/lib64/libbgrealtime.so`.
  
  Both the include and shared library files are installed as part of the standard system installation. They are contained in the bgpbase.rpm file.

Makefile excerpt

Example 14-1 shows a possible excerpt from a makefile that you might want to create to help automate builds of your application. This sample is shipped in the `/bgsys/drivers/ppcfloor/doc/realtime/simple/Makefile` directory. In this makefile, the program that is being built is `rt_sample_app`, and the source is in the `rt_sample_app.cc` file.

Example 14-1 Makefile excerpt

```makefile
... 
ALL_APPS = rt_sample_app
CXXFLAGS += -w -Wall -g -m64 -pthread
CXXFLAGS += -I/bgsys/drivers/ppcfloor/include
LDFLAGS += -L/bgsys/drivers/ppcfloor/lib64 -lbgrealtme
LDFLAGS += -pthread

.PHONY: all clean default distclean

default: $(ALL_APPS)
```
Real-time server
Before using these functions, the Blue Gene/P administrator must start the real-time server. Otherwise the \texttt{rt\_init} API returns an \texttt{RT\_CONNECTION\_ERROR} status code. Configuring and starting the real-time server is documented in \textit{IBM System Blue Gene Solution: Blue Gene/P System Administration}, SG24-7417.

14.1.2 General comments
All the real-time APIs have general considerations that apply to all calls. We highlight the common features here:

- All the API calls return an \texttt{rt\_status\_t}, which indicates either success or a status code. Successful status codes are non-negative, where failure status codes are negative.
- All the API calls take a pointer to \texttt{rt\_handle\_t}, which is an opaque structure that represents a stream of real-time messages.
- The real-time APIs use \texttt{sayMessage} APIs for printing debug and error messages. The application should set the \texttt{sayMessage} logging level before calling the real-time APIs.

Blocking mode versus non-blocking mode
A real-time handle can be in \texttt{blocking} or \texttt{non-blocking} mode. In blocking mode, the \texttt{rt\_request\_realtime} API blocks until it can send the request, and the \texttt{rt\_read\_msgs} API blocks until there is an event to receive. In non-blocking mode, the \texttt{rt\_request\_realtime} API returns \texttt{RT\_WOULD\_BLOCK} if it cannot send the request. If you get this return code from \texttt{rt\_request\_realtime}, you must call it again until it returns \texttt{RT\_FINISHED\_PREV}. In non-blocking mode, the \texttt{rt\_read\_msgs} API returns \texttt{RT\_NO\_REALTIME\_MSG\_S} immediately if no real-time event is ready to be processed.

The \texttt{rt\_get\_socket\_descriptor} API can be used to get a file descriptor that can be used with a \texttt{select()}-type function to wait for a real-time event when a handle is in blocking mode.

The initial blocking or non-blocking mode is set using the \texttt{rt\_init} API. An initialized handle can be set to blocking mode by using the \texttt{rt\_set\_blocking} API or set to non-blocking mode by using the \texttt{rt\_set\_nonblocking} API.

Filtering events
A real-time handle can be configured so that only partition events that affect certain partitions, job events, or both, which affect certain jobs, are passed to the application.

Setting the partition filter is done by using the \texttt{rt\_set\_filter} API with \texttt{RT\_PARTITION} as the \texttt{filter\_type} parameter. The \texttt{filter\_names} parameter can specify one or more partition IDs separated by spaces. When the \texttt{rt\_get\_msgs} API is called, partition events are delivered only to the application if the partition ID matches any of the partition IDs in the filter. If the \texttt{filter\_names} parameter is set to \texttt{NULL}, then the partition filter is removed, and all partition...
events are delivered to the application. An example of the value to use for the filter_names parameter for partition IDs R00-M0 and R00-M1 is R00-M0 R00-M1.

Setting the job filter is done by using the rt_set_filter API with RT_JOB as the filter_type parameter. The filter_names parameter can specify one or more job IDs (as strings) separated by spaces. When the rt_get_msgs API is called, job events are delivered only to the application if the job ID matches any of the job IDs in the filter. If the filter_names parameter is set to NULL, the job filter is removed, and all job events are delivered to the application. An example of the value to use for the filter_names parameter for job IDs 10030 and 10031 is 10030 10031.

The other use of the rt_set_filter API is to remove both types of filter by passing RT_CLEAR_ALL in the filter_type parameter.

### 14.2 Real-time Notification APIs

In this section, we describe the Real-time Notification APIs, which are as follows:

- **rt_status_t rt_init(rt_handle_t **handle_out, rt_block_flag_t blocking flag, rt_callbacks_t* callbacks);**
  - Initializes a real-time handle. This function gets the port of the real-time server from the db.properties file. The name of the db.properties file must be in the DB_PROPERTY environment variable, or RT_DB_PROPERTY_ERROR is returned.
  - If this function is successful, *handle_out is set to a valid handle connected to the real-time server. The blocking state for the handle is set based on the blocking flag parameter. The callbacks for the handle are set to the callbacks parameter. If this function is not successful and handle_out is not NULL, then *handle_out is set to NULL.

- **rt_status_t rt_close(rt_handle_t **handle);**
  - Closes a real-time handle. The handle must not be used after calling this function.

- **rt_status_t rt_set_blocking(rt_handle_t **handle);**
  - Sets a real-time handle to blocking mode.

- **rt_status_t rt_set_nonblocking(rt_handle_t **handle);**
  - Sets a real-time handle to non-blocking mode.

- **rt_status_t rt_set_filter(rt_handle_t **handle, rt_filter_type_t filter_type, const char* filter_names);**
  - Sets the filter on a real-time handle. The filter names consists of a C-style string that contains a space-separated list of names to filter on. If removing filter entries, set this to NULL. For filtering on partition names, consider this example of R01-M0 R02-M1 R03.

- **rt_status_t rt_request_realtime(rt_handle_t **handle);**
  - Requests real-time events for this handle. If this function returns RT_WOULD_BLOCK, the request has not been sent. Call this function again until it returns RT_FINISHED_PREV, which indicates that the previous request has been sent.
  - If this function returns RT_FINISHED_PREV, then a new request was not sent.

- **rt_status_t rt_get_socket_descriptor(rt_handle_t **handle, int *sd_out);**
  - Gets the socket descriptor used by the real-time APIs. You can use this socket descriptor with the select() or poll() Linux APIs to wait until a real-time message is ready to be read. Other file or socket descriptor APIs, such as close(), should not be used on the socket descriptor returned by this API.
Chapter 14. Real-time Notification APIs

14.3 Real-time callback functions

Developers who use the Real-time Notification APIs must write functions that are called when real-time events are received. These functions are callback functions because the application calls the rt_read_msgs API, which then calls the function supplied by the application.

Pointers to the callback functions must be set in an rt_callbacks_t structure. When a real-time event is received, the corresponding function is called using that pointer. The application passes its rt_callbacks_t into rt_init, which is stored for use when rt_read_msgs is called. If the pointer to the callback function in the rt_callbacks_t structure is NULL, then the event is discarded.

In addition to setting the callback functions in the rt_callbacks_t structure, the application must also set the version field to RT_CALLBACK_VERSION_0. With a later version of the real-time APIs, you can allow different callbacks and provide a different version for this field.

From inside your callback function, you cannot call a real-time API using the same handle on which the event occurred. Otherwise your application deadlocks.

The return type of the callback functions is an indicator of whether rt_read_msgs should continue to attempt to receive another real-time event on the handle or whether it should stop. If the callback function returns RT_CALLBACK_CONTINUE, rt_read_msgs continues to attempt to receive real-time events. If the callback function returns RT_CALLBACK_QUIT, rt_read_msgs does not attempt to receive another real-time event but returns RT_STATUS_OK.

Sequence identifiers (IDs) are associated with the state of each partition, job, base partition, node card, and switch. A state with a higher sequence ID is newer. If your application gets the state for an object from the Bridge APIs in addition to the real-time APIs, you must discard any state that has a lower sequence ID for the same object.

These APIs provide the raw state for partitions, jobs, base partitions, node cards, and switches in addition to providing the state. The raw state is the status value stored in the Blue Gene database as a single character, rather than the state enumeration that the Bridge APIs use. Several raw state values map to a single state value, so that your application might receive real-time event notifications where the state does not change but the raw state does. For example, the partition raw states of “A” (allocating), “C” (configuring), and “B” (booting) all map to the Bridge enumerated state of RM_PARTITION_CONFIGURING.

Real-time callback structure

In this section, we describe each of the callbacks available to applications in the rt_callbacks_t structure. We list each field of the structure along with the following information:

- The description of the event that causes the callback to be invoked

- rt_status_t rt_read_msgs(rt_handle_t **handle, void* data);
  Receives real-time events on a handle. If the handle is blocking, this function blocks as long as no events are waiting. If the handle is non-blocking, the function returns immediately with RT_NO_REALTIME_MSGS if no events are waiting. If an event is waiting to be processed, the callback associated with the event type is called. If the callback returns RT_CALLBACK_CONTINUE, events continue to be processed.
The signature of the callback function

Your function must match the signature. Otherwise your program fails to compile.

A description of each argument to the callback function

**Field end_cb**

The field `end_cb` callback function is called when a real-time ended event occurs. Your application does not receive any more real-time events on this handle until you request real-time events from the server again by calling the `rt_request_realtime` API.

The function uses the following signature:

```c
cb_ret_t my_rt_end(rt_handle_t **handle, void* extended_args, void* data);
```

Table 14-1 lists the arguments to the field `end_cb` function.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Real-time handle on which the event occurred</td>
</tr>
<tr>
<td>extended_args</td>
<td>Not used; is NULL for now</td>
</tr>
<tr>
<td>data</td>
<td>Application data forwarded by <code>rt_read_msgs</code></td>
</tr>
</tbody>
</table>

**Field partition_added_cb**

The field `partition_added_cb` function is called when a partition added event occurs.

The function uses the following signature:

```c
cb_ret_t my_rt_partition_added(rt_handle_t **handle, rm_sequence_id_t seq_id, pm_partition_id_t partition_id, rm_partition_state_t partition_new_state, rt_raw_state_t partition_raw_new_state, void* extended_args, void* data);
```

Table 14-2 lists the arguments to the field `partition_added_cb` function.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Real-time handle on which the event occurred</td>
</tr>
<tr>
<td>seq_id</td>
<td>Sequence ID for this partition’s state</td>
</tr>
<tr>
<td>partition_id</td>
<td>The partition’s ID</td>
</tr>
<tr>
<td>partition_new_state</td>
<td>The partition’s new state</td>
</tr>
<tr>
<td>partition_raw_new_state</td>
<td>The partition’s new raw state</td>
</tr>
<tr>
<td>extended_args</td>
<td>Not used; NULL for now</td>
</tr>
<tr>
<td>data</td>
<td>Application data forwarded by <code>rt_read_msgs</code></td>
</tr>
</tbody>
</table>

**Field partition_state_changed_cb**

The field `partition_state_changed_cb` function is called when a partition state changed event occurs.

The function uses the following signature:

```c
cb_ret_t my_rt_partition_state_changed(rt_handle_t **handle, rm_sequence_id_t seq_id, rm_sequence_id_t previous_seq_id, pm_partition_id_t partition_id, pm_partition_state_t partition_new_state, rt_raw_state_t partition_raw_new_state, void* extended_args, void* data);
```
rm_partition_state_t partition_new_state, rm_partition_state_t partition_old_state, rt_raw_state_t partition_raw_new_state, rt_raw_state_t partition_raw_old_state, void* extended_args, void* data);

Table 14-3 lists the arguments to the field partition_state_changed_cb function.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Real-time handle on which the event occurred</td>
</tr>
<tr>
<td>seq_id</td>
<td>Sequence ID for this partition's new state</td>
</tr>
<tr>
<td>previous_seq_id</td>
<td>Sequence ID for this partition's old state</td>
</tr>
<tr>
<td>partition_id</td>
<td>The partition's ID</td>
</tr>
<tr>
<td>partition_new_state</td>
<td>The partition's new state</td>
</tr>
<tr>
<td>partition_old_state</td>
<td>The partition's old state</td>
</tr>
<tr>
<td>partition_raw_new_state</td>
<td>The partition's new raw state</td>
</tr>
<tr>
<td>partition_raw_old_state</td>
<td>The partition's old raw state</td>
</tr>
<tr>
<td>extended_args</td>
<td>Not used; NULL for now</td>
</tr>
<tr>
<td>data</td>
<td>Application data forwarded by rt_read_msgs</td>
</tr>
</tbody>
</table>

Field partition_deleted_cb
The field partition_deleted_cb is called when a partition deleted event occurs.

The function uses the following signature:

```
cb_ret_t my_rt_partition_deleted(rt_handle_t **handle, rm_sequence_id_t previous_seq_id, pm_partition_id_t partition_id, void* extended_args, void* data);
```

Table 14-4 lists the arguments to the field partition_deleted_cb function.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Real-time handle on which the event occurred</td>
</tr>
<tr>
<td>previous_seq_id</td>
<td>Sequence ID for this partition's state when removed</td>
</tr>
<tr>
<td>partition_id</td>
<td>The partition's ID</td>
</tr>
<tr>
<td>extended_args</td>
<td>Not used; NULL for now</td>
</tr>
<tr>
<td>data</td>
<td>Application data forwarded by rt_read_msgs</td>
</tr>
</tbody>
</table>

Field job_added_cb
The field job_added_cb function is called when a job added event occurs.

The function uses the following signature:

```
cb_ret_t my_rt_job_added(rt_handle_t **handle, rm_sequence_id_t seq_id, db_job_id_t job_id, pm_partition_id_t partition_id, rm_job_state_t job_new_state, rt_raw_state_t job_raw_new_state, void* extended_args, void* data);
```
Table 14-5 lists the arguments to the field `job_added_cb` function.

**Table 14-5  Field job_added_cb**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Real-time handle on which the event occurred</td>
</tr>
<tr>
<td>seq_id</td>
<td>Sequence ID for the job's state</td>
</tr>
<tr>
<td>job_id</td>
<td>The new job's ID</td>
</tr>
<tr>
<td>partition_id</td>
<td>ID of the partition to which the job is assigned</td>
</tr>
<tr>
<td>job_new_state</td>
<td>The job's new state</td>
</tr>
<tr>
<td>job_raw_new_state</td>
<td>The job's new raw state</td>
</tr>
<tr>
<td>extended_args</td>
<td>Not used; NULL for now</td>
</tr>
<tr>
<td>data</td>
<td>Application data forwarded by rt_read_msgs</td>
</tr>
</tbody>
</table>

**Field job_state_changed_cb**
The field `job_state_changed_cb` function is called when a job state changed event occurs.

The function uses the following signature:

```c
cb_ret_t my_rt_job_state_changed(rt_handle_t **handle, rm_sequence_id_t seq_id, rm_sequence_id_t previous_seq_id, db_job_id_t job_id, pm_partition_id_t partition_id, rm_job_state_t job_new_state, rm_job_state_t job_old_state, rt_raw_state_t job_raw_new_state, rt_raw_state_t job_raw_old_state, void* extended_args, void* data);
```

Table 14-6 lists the arguments to the field `job_state_changed_cb` function.

**Table 14-6  Field job_state_changed_cb**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Real-time handle on which the event occurred</td>
</tr>
<tr>
<td>seq_id</td>
<td>Sequence ID for the job's new state</td>
</tr>
<tr>
<td>previous_seq_id</td>
<td>Sequence ID of the job's previous state</td>
</tr>
<tr>
<td>job_id</td>
<td>The job's ID</td>
</tr>
<tr>
<td>partition_id</td>
<td>ID of the partition to which the job is assigned</td>
</tr>
<tr>
<td>job_new_state</td>
<td>The job's new state</td>
</tr>
<tr>
<td>job_old_state</td>
<td>The job's old state</td>
</tr>
<tr>
<td>job_raw_new_state</td>
<td>The job's new raw state</td>
</tr>
<tr>
<td>job_raw_old_state</td>
<td>The job's old raw state</td>
</tr>
<tr>
<td>extended_args</td>
<td>Not used; NULL for now</td>
</tr>
<tr>
<td>data</td>
<td>Application data forwarded by rt_read_msgs</td>
</tr>
</tbody>
</table>
**Field job_deleted_cb**

The field `job_deleted_cb` function is called when a job-deleted event occurs.

The function uses the following signature:

```c
cb_ret_t my_rt_job_deleted(rt_handle_t **handle, rm_sequence_id_t previous_seq_id,
                           db_job_id_t job_id, pm_partition_id_t partition_id, void* extended_args, void* data);
```

Table 14-7 lists the arguments to the field `job_deleted_cb` function.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Real-time handle on which the event occurred</td>
</tr>
<tr>
<td>previous_seq_id</td>
<td>Sequence ID of the job's previous state</td>
</tr>
<tr>
<td>job_id</td>
<td>The deleted job's ID</td>
</tr>
<tr>
<td>partition_id</td>
<td>ID of the partition to which the job was assigned</td>
</tr>
<tr>
<td>extended_args</td>
<td>Not used; NULL for now</td>
</tr>
<tr>
<td>data</td>
<td>Application data forwarded by rt_read_msgs</td>
</tr>
</tbody>
</table>

**Field bp_state_changed_cb**

The field `bp_state_changed_cb` is called when a base partition state changed event occurs.

The function uses the following signature:

```c
cb_ret_t (*rt_BP_state_changed_fn_p)(rt_handle_t **handle, rm_sequence_id_t seq_id,
                                     rm_sequence_id_t previous_seq_id, rm_bp_id_t bp_id, rm_BP_state_t
                                     BP_new_state, rm_BP_state_t BP_old_state, rt_raw_state_t BP_raw_new_state,
                                     rt_raw_state_t BP_raw_old_state, void* extended_args, void* data);
```

Table 14-8 lists the arguments to the field `bp_state_changed_cb` function.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Real-time handle on which the event occurred</td>
</tr>
<tr>
<td>seq_id</td>
<td>Sequence ID of the base partition's new state</td>
</tr>
<tr>
<td>previous_seq_id</td>
<td>Sequence ID of the base partition's previous state</td>
</tr>
<tr>
<td>bp_id</td>
<td>The base partition's ID</td>
</tr>
<tr>
<td>BP_new_state</td>
<td>The base partition's new state</td>
</tr>
<tr>
<td>BP_old_state</td>
<td>The base partition's old state</td>
</tr>
<tr>
<td>BP_raw_new_state</td>
<td>The base partition's new raw state</td>
</tr>
<tr>
<td>BP_raw_old_state</td>
<td>The base partition's old raw state</td>
</tr>
<tr>
<td>extended_args</td>
<td>Not used; NULL for now</td>
</tr>
<tr>
<td>data</td>
<td>Application data forwarded by rt_read_msgs</td>
</tr>
</tbody>
</table>
**Field switch_state_changed_cb**
The field `switch_state_changed_cb` is called when a switch state changed event occurs.

The function uses the following signature:

```c
cb_ret_t my_rt_switch_state_changed(rt_handle_t **handle, rm_sequence_id_t seq_id, rm_sequence_id_t previous_seq_id, rm_switch_id_t switch_id, rm_bp_id_t bp_id, rm_switch_state_t switch_new_state, rm_switch_state_t switch_old_state, rt_raw_state_t switch_raw_new_state, rt_raw_state_t switch_raw_old_state, void* extended_args, void* data);
```

Table 14-9 lists the arguments to the field `switch_state_changed_cb` function.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Real-time handle on which the event occurred</td>
</tr>
<tr>
<td>seq_id</td>
<td>Sequence ID for the switch's new state</td>
</tr>
<tr>
<td>previous_seq_id</td>
<td>Sequence ID of the switch's previous state</td>
</tr>
<tr>
<td>switch_id</td>
<td>The switch's ID</td>
</tr>
<tr>
<td>bp_id</td>
<td>The switch's base partition's ID</td>
</tr>
<tr>
<td>switch_new_state</td>
<td>The switch's new state</td>
</tr>
<tr>
<td>switch_old_state</td>
<td>The switch's old state</td>
</tr>
<tr>
<td>switch_raw_new_state</td>
<td>The switch's new raw state</td>
</tr>
<tr>
<td>switch_raw_old_state</td>
<td>The switch's old raw state</td>
</tr>
<tr>
<td>extended_args</td>
<td>Not used; NULL for now</td>
</tr>
<tr>
<td>data</td>
<td>Application data forwarded by <code>rt_read_msgs</code></td>
</tr>
</tbody>
</table>

**Field nodecard_state_changed_cb**
The field `nodecard_state_changed_cb` is called when a node card state changed event occurs.

The function uses the following signature:

```c
b_ret_t my_rt_nodecard_state_changed(rt_handle_t **handle, rm_sequence_id_t seq_id, rm_sequence_id_t previous_seq_id, rm_nodecard_id_t nodecard_id, rm_bp_id_t bp_id, rm_nodecard_state_t nodecard_new_state, rm_nodecard_state_t nodecard_old_state, rt_raw_state_t nodecard_raw_new_state, rt_raw_state_t nodecard_raw_old_state, void* extended_args, void* data);
```
Table 14-10 lists the arguments to the field `nodecard_state_changed_cb` function.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>Real-time handle on which the event occurred</td>
</tr>
<tr>
<td>seq_id</td>
<td>Sequence ID for the node card's new state</td>
</tr>
<tr>
<td>previous_seq_id</td>
<td>Sequence ID of the node card's previous state</td>
</tr>
<tr>
<td>nodecard_id</td>
<td>The node card's ID</td>
</tr>
<tr>
<td>bp_id</td>
<td>The node card's base partition's ID</td>
</tr>
<tr>
<td>nodecard_new_state</td>
<td>The node card's new state</td>
</tr>
<tr>
<td>nodecard_old_state</td>
<td>The node card's old state</td>
</tr>
<tr>
<td>nodecard_raw_new_state</td>
<td>The node card's new raw state</td>
</tr>
<tr>
<td>nodecard_raw_old_state</td>
<td>The node card's old raw state</td>
</tr>
<tr>
<td>extended_args</td>
<td>Not used; NULL for now</td>
</tr>
<tr>
<td>data</td>
<td>Application data forwarded by <code>rt_read_msgs</code></td>
</tr>
</tbody>
</table>

### 14.4 Real-time Notification API status codes

When a failure occurs, an API invocation returns a status code. This status code helps apply automatic corrective actions within the resource management application. In addition, a failure always generates a log message, which provides more information for the possible cause of the problem and any corrective action. These log messages are used for debugging and non-automatic recovery of failures.

The design aims at striking a balance between the number of status codes detected and the different error paths per status code. Thus, some errors have specific status codes, while others have more generic ones.

The Real-time Notification API uses the following status codes:

- **RT_STATUS_OK**: API call completed successfully.
- **RT_NO_REALTIME_MSGS**: No events available.
- **RT_WOULD_BLOCK**: In non-blocking mode and request would block.
- **RT_FINISHED_PREV**: Previous request completed.
- **RT_CONNECTION_ERROR**: Connection to the real-time server failed.
- **RT_INTERNAL_ERROR**: Unexpected internal error. No recovery possible.
- **RT_INVALID_INPUT_ERROR**: The input to the API is bad due to missing required data, illegal data, and so on.
- **RT_DB_PROPERTY_ERROR**: Error trying to read the `db.properties` file.
- **RT_PROTOCOL_ERROR**: An incorrect message was received from the real-time server.
- **RT_HANDLE_CLOSED**: The handle passed to the API was previously closed.
14.4.1 Status code specification

The various API functions have the following status codes:

- `rt_status_t rt_init(rt_handle_t **handle_out, rt_block_flag_t blocking_flag, rt_callbacks_t* callbacks);`
  
  This function initializes a real-time handle.

  The status codes are:
  - RT_STATUS_OK: The handle is initialized.
  - RT_INVALID_INPUT_ERROR: One or more of the parameters are not valid.
  - RT_CONNECTION_ERROR: Failed to connect to the real-time server.
  - RT_INTERNAL_ERROR: There was an unexpected internal error in setting blocking or non-blocking mode on socket.
  - RT_DB_PROPERTY_ERROR: Problem accessing the db.properties file.

- `rt_status_t rt_close(rt_handle_t **handle);`
  
  This function closes a real-time handle.

  The status codes are:
  - RT_STATUS_OK: The handle was closed.
  - RT_INVALID_INPUT_ERROR: The handle is not valid.
  - RT_INTERNAL_ERROR: There was an unexpected internal error in closing the handle.

- `rt_status_t rt_set_blocking(rt_handle_t **handle);`
  
  This function sets a real-time handle to blocking mode.

  The status codes are:
  - RT_STATUS_OK: Blocking mode was set for the handle.
  - RT_INVALID_INPUT_ERROR: The handle is not valid.
  - RT_INTERNAL_ERROR: There was an unexpected internal error in setting blocking mode.

- `rt_status_t rt_set_nonblocking(rt_handle_t **handle);`
  
  This function sets a real-time handle to non-blocking mode.

  The status codes are:
  - RT_STATUS_OK: Non-blocking mode was set for the handle.
  - RT_INVALID_INPUT_ERROR: The handle is not valid.
  - RT_INTERNAL_ERROR: There was an unexpected internal error in setting non-blocking mode.

- `rt_status_t rt_set_filter(rt_handle_t **handle, rt_filter_type_t filter_type, const char* filter_names);`
  
  This function sets the filter on a real-time handle.

  The status codes are:
  - RT_STATUS_OK: Filtering was set successfully.
  - RT_INVALID_INPUT_ERROR: The handle is not valid.
  - RT_INTERNAL_ERROR: There was an unexpected internal error in setting the filter.

- `rt_status_t rt_request_realtime(rt_handle_t **handle);`
  
  This function requests real-time events for this handle.

  The status codes are:
  - RT_STATUS_OK: Request to start real-time updates was successful.
  - RT_INVALID_INPUT_ERROR: The handle is not valid.
  - RT_CONNECTION_ERROR: The connection to the server was lost.
RT_WOULD_BLOCK: The handle is non-blocking and this request would block.

RT_FINISHED_PREV: A previous request finished.

- rt_status_t rt_get_socket_descriptor(rt_handle_t **handle, int *sd_out);
  This function gets the socket descriptor used by the real-time APIs.
  
  The status codes are:
  - RT_STATUS_OK: Socket descriptor was retrieved successfully.
  - RT_INVALID_INPUT_ERROR: The handle is not valid.

- rt_status_t rt_read_msgs(rt_handle_t **handle, void* data);
  This function receives real-time events.
  
  The status codes are:
  - RT_STATUS_OK: Message or messages were read successfully.
  - RT_INVALID_INPUT_ERROR: The handle is not valid.
  - RT_CONNECTION_ERROR: The connection to the server was lost.
  - RT_INTERNAL_ERROR: There was an unexpected internal error when reading messages.
  - RT_HANDLE_CLOSED: The real-time handle was closed.
  - RT_NO_REALTIME_MSGS: Non-blocking mode and no messages to receive.

### 14.5 Sample real-time application code

Example 14-2 shows basic example code for calling the real-time APIs and programming the callback functions.

```c
#include <rt_api.h>
#include <sayMessage.h>
#include <stdio.h>
#include <unistd.h>
#include <iostream>
#include <sstream>

using namespace std;
```

Title: Blue Gene Real-time Notification Interface - Sample Program

The environment must have DB_PROPERTY set for the realtime apis.

```c
#include <rt_api.h>
#include <sayMessage.h>

#include <getopt.h>
#include <stdio.h>
#include <unistd.h>

#include <iostream>
#include <sstream>

using namespace std;
```
// Converts partition state enum to character string for messages
string partition_state_to_msg(rm_partition_state_t state)
{
    switch(state) {
        case RM_PARTITION_FREE:
            return "Free";
        case RM_PARTITION_CONFIGURING:
            return "Configuring";
        case RM_PARTITION_READY:
            return "Ready";
        case RM_PARTITION_DEALLOCATING:
            return "Deallocating";
        case RM_PARTITION_ERROR:
            return "Error";
        case RM_PARTITION_NAV:
            return "Not a value (NAV)";
    }
    return "Unknown";
}

// Converts job state enum to character string for messages
string job_state_to_msg(rm_job_state_t state)
{
    switch(state) {
        case RM_JOB_IDLE:
            return "Queued/Idle";
        case RM_JOB_STARTING:
            return "Starting";
        case RM_JOB_RUNNING:
            return "Running";
        case RM_JOB_TERMINATED:
            return "Terminated";
        case RM_JOB_ERROR:
            return "Error";
        case RM_JOB_DYING:
            return "Dying";
        case RM_JOB_DEBUG:
            return "Debug";
        case RM_JOB_LOAD:
            return "Load";
        case RM_JOB_LOADED:
            return "Loaded";
        case RM_JOB_BEGIN:
            return "Begin";
        case RM_JOB_ATTACH:
            return "Attach";
        case RM_JOB_NAV:
            return "Not a value (NAV)";
    }
    return "Unknown";
}

// Converts BP state enum to character string for messages
string BP_state_to_msg(rm_BP_state_t state)
{
    switch(state) {
        case RM_BP_UP:
            return "Available/Up";
        case RM_BP_MISSING:
            return "Missing";
    }
}
case RM_BP_ERROR:
    return "Error";
    case RM_BP_DOWN:
    return "Service/Down";
    case RM_BP_NAV:
    return "Not a value (NAV)";
}
    return "Unknown";
}

// Converts switch state enum to character string for messages
string switch_state_to_msg(rm_switch_state_t state)
{
    switch(state) {
        case RM_SWITCH_UP:
            return "Available/Up";
        case RM_SWITCH_MISSING:
            return "Missing";
        case RM_SWITCH_ERROR:
            return "Error";
        case RM_SWITCH_DOWN:
            return "Service/Down";
        case RM_SWITCH_NAV:
            return "Not a value (NAV)";
    }
    return "Unknown";
}

// Converts nodecard state enum to character string for messages
string nodecard_state_to_msg(rm_nodecard_state_t state)
{
    switch(state) {
        case RM_NODECARD_UP:
            return "Available/Up";
        case RM_NODECARD_MISSING:
            return "Missing";
        case RM_NODECARD_ERROR:
            return "Error";
        case RM_NODECARD_DOWN:
            return "Service/Down";
        case RM_NODECARD_NAV:
            return "Not a value (NAV)";
    }
    return "Unknown";
}

/* Definitions of the Real-time callback functions. */

cb_ret_t rt_end_callback(
    rt_handle_t **handle,
    void* extended_args,
    void* data)
{
    cout << "Received real-time end message." << endl;
    return RT_CALLBACK_QUIT;
}

cb_ret_t rt_partition_added_callback(

rt_handle_t **handle,
rm_sequence_id_t seq_id,
pm_partition_id_t partition_id,
rm_partition_state_t partition_new_state,
rt_raw_state_t partition_raw_new_state,
void* extended_args,
void* data)
{
    cout << "Received callback for add partition " << partition_id
    << " state of partition is " << partition_state_to_msg(partition_new_state) <<
endl
    << "Raw state=" << partition_raw_new_state << " sequence ID=" << seq_id << endl;
return RT_CALLBACK_CONTINUE;
}

cb_ret_t rt_partition_state_changed_callback(
    rt_handle_t **handle,
    rm_sequence_id_t seq_id,
    rm_sequence_id_t prev_seq_id,
    pm_partition_id_t partition_id,
    rm_partition_state_t partition_new_state,
    rm_partition_state_t partition_old_state,
    rt_raw_state_t partition_raw_new_state,
    rt_raw_state_t partition_raw_old_state,
    void* extended_args,
    void* data)
{
    cout << "Received callback for partition " << partition_id
    << " state change, old state is " << partition_state_to_msg(partition_old_state)
    << ", new state is " << partition_state_to_msg(partition_new_state) << endl
    << "Raw old state=" << partition_raw_old_state
    << " Raw new state=" << partition_raw_new_state
    << " New sequence ID=" << seq_id << " Previous sequence ID=" << prev_seq_id <<
endl;
return RT_CALLBACK_CONTINUE;
}

cb_ret_t rt_partition_deleted_callback(
    rt_handle_t **handle,
    rm_sequence_id_t prev_seq_id,
    pm_partition_id_t partition_id,
    void* extended_args,
    void* data)
{
    cout << "Received callback for delete on partition " << partition_id
    << " Previous sequence ID=" << prev_seq_id << endl;
return RT_CALLBACK_CONTINUE;
}

cb_ret_t rt_job_added_callback(
    rt_handle_t **handle,
    rm_sequence_id_t seq_id,
    db_job_id_t job_id,
    pm_partition_id_t partition_id,
    rm_job_state_t job_new_state,
    rt_raw_state_t job_raw_new_state,
    void* extended_args,
cb_ret_t rt_job_state_changed_callback(
    rt_handle_t **handle,
    rm_sequence_id_t seq_id,
    rm_sequence_id_t prev_seq_id,
    db_job_id_t job_id,
    pm_partition_id_t partition_id,
    rm_job_state_t job_new_state,
    rm_job_state_t job_old_state,
    rt_raw_state_t job_raw_new_state,
    rt_raw_state_t job_raw_old_state,
    void* extended_args,
    void* data)
{
    cout << "Received callback for job " << job_id
        << " on partition " << partition_id << ", state of job is "
        << job_state_to_msg(job_new_state) << endl
        << "Raw new state=" << job_raw_new_state << " New sequence ID=" << seq_id << endl;
    return RT_CALLBACK_CONTINUE;
}

cb_ret_t rt_job_deleted_callback(
    rt_handle_t **handle,
    rm_sequence_id_t prev_seq_id,
    db_job_id_t job_id,
    pm_partition_id_t partition_id,
    void* extended_args,
    void* data)
{
    cout << "Received callback for delete of job " << job_id
        << " on partition " << partition_id << " Previous sequence ID=" << prev_seq_id << endl;
    return RT_CALLBACK_CONTINUE;
}

cb_ret_t rt_BP_state_changed_callback(
    rt_handle_t **handle,
    rm_sequence_id_t seq_id,
    rm_sequence_id_t prev_seq_id,
    rm_bp_id_t bp_id,
    rm_BP_state_t BP_new_state,
    rm_BP_state_t BP_old_state,
    rt_raw_state_t BP_raw_new_state,
    rt_raw_state_t BP_raw_old_state,
    void* extended_args,
    void* data)
cb_ret_t rt_switch_state_changed_callback(
    rt_handle_t **handle,
    rm_sequence_id_t seq_id,
    rm_sequence_id_t prev_seq_id,
    rm_switch_id_t switch_id,
    rm_bp_id_t bp_id,
    rm_switch_state_t switch_new_state,
    rm_switch_state_t switch_old_state,
    rt_raw_state_t switch_raw_new_state,
    rt_raw_state_t switch_raw_old_state,
    void* extended_args,
    void* data)
{
    cout << "Received callback for switch " << switch_id
     << " state change on BP " << bp_id
     << " old state is " << state_to_msg(switch_old_state)
     << " new state is " << state_to_msg(switch_new_state) << endl
     << " Raw old state=" << switch_raw_old_state << " Raw new state="
     << switch_raw_new_state
     << " New sequence ID=" << seq_id << " Previous sequence ID=" << prev_seq_id <<
     << " New sequence ID=" << seq_id << " Previous sequence ID=" << prev_seq_id <<
    return RT_CALLBACK_CONTINUE;
}

cb_ret_t rt_nodecard_state_changed_callback(
    rt_handle_t **handle,
    rm_sequence_id_t seq_id,
    rm_sequence_id_t prev_seq_id,
    rm_nodecard_id_t nodecard_id,
    rm_bp_id_t bp_id,
    rm_nodecard_state_t nodecard_new_state,
    rm_nodecard_state_t nodecard_old_state,
    rt_raw_state_t nodecard_raw_new_state,
    rt_raw_state_t nodecard_raw_old_state,
    void* extended_args,
    void* data)
{
    cout << "Received callback for node card " << nodecard_id
     << " state change on BP " << bp_id
     << " old state is " << nodecard_state_to_msg(nodecard_old_state)
     << " new state is " << nodecard_state_to_msg(nodecard_new_state) << endl
     << " Raw old state=" << nodecard_raw_old_state
     << " Raw new state=" << nodecard_raw_new_state
     << " New sequence ID=" << seq_id << " Previous sequence ID=" << prev_seq_id <<
    return RT_CALLBACK_CONTINUE;
}
/* Program entry point */

int main( int argc, char *argv[] ) {
    string job_filter, *job_filter_p(0);
    string partition_filter, *partition_filter_p(0);
    int verbose(0);

    const int JOB_FILTER_PARAM_IND = 0;
    const int PARTITION_FILTER_PARAM_IND = 1;
    const int VERBOSE_PARAM_IND = 2;

    struct option long_options[] = {
        { "job_filter", 1, 0, JOB_FILTER_PARAM_IND },
        { "partition_filter", 1, 0, PARTITION_FILTER_PARAM_IND },
        { "verbose", 1, 0, VERBOSE_PARAM_IND },
        { 0, 0, 0, 0 }
    };

    int option_index = 0;

    while ( 1 ) {
        int getopt_ret = getopt_long(argc, argv, "", long_options, &option_index);
        if (-1 == getopt_ret) {
            break;
        }
        switch ( getopt_ret ) {
            case JOB_FILTER_PARAM_IND:
                job_filter = optarg;
                job_filter_p = &job_filter;
                break;
            case PARTITION_FILTER_PARAM_IND:
                partition_filter = optarg;
                partition_filter_p = &partition_filter;
                break;
            case VERBOSE_PARAM_IND:
                { 
                    istringstream iss( optarg );
                    iss >> verbose;
                }
                break;
        }
    }

    setSayMessageParams(stdout, verbose);

    rt_handle_t  *rt_handle;
    rt_callbacks_t rt_callbacks;

    rt_callbacks.version = RT_CALLBACK_VERSION_0;
    rt_callbacks.end_cb = &rt_end_callback;
    rt_callbacks.partition_added_cb = &rt_partition_added_callback;
    rt_callbacks.partition_state_changed_cb = &rt_partition_state_changed_callback;
    rt_callbacks.partition_deleted_cb = &rt_partition_deleted_callback;
    rt_callbacks.job_added_cb = &rt_job_added_callback;
    rt_callbacks.job_state_changed_cb = &rt_job_state_changed_callback;
    rt_callbacks.job_deleted_cb = &rt_job_deleted_callback;
    rt_callbacks.bp_state_changed_cb = &rt_BP_state_changed_callback;
    rt_callbacks.switch_state_changed_cb = &rt_switch_state_changed_callback;
rt_callbacks.nodecard_state_changed_cb = &rt_nodecard_state_changed_callback;

// Get a handle, set socket to block, and setup callbacks
if (rt_init(&rt_handle, RT_BLOCKING, &rt_callbacks) != RT_STATUS_OK)
{
    cout << "Failed on real-time initialize (rt_init), exiting program." << endl;
    return -1;
}

// Set the job filter if requested.
if (job_filter_p != 0) {
    if (rt_set_filter(&rt_handle, RT_JOB, job_filter_p->c_str()) != RT_STATUS_OK) {
        cout << "Failed to set job filter." << endl;
        rt_close(&rt_handle);
        return -1;
    }
}

// Set the partition filter if requested.
if (partition_filter_p != 0) {
    if (rt_set_filter(&rt_handle, RT_PARTITION, partition_filter_p->c_str()) != RT_STATUS_OK) {
        cout << "Failed to set partition filter." << endl;
        rt_close(&rt_handle);
        return -1;
    }
}

// Tell real-time server we are ready to handle messages
if (rt_request_realtime(&rt_handle) != RT_STATUS_OK) {
    cout << "Failed to connect to real-time server, exiting program." << endl;
    rt_close(&rt_handle);
    return -1;
}

// Read messages
if (rt_read_msgs(&rt_handle, NULL) != RT_STATUS_OK) {
    cout << "rt_read_msgs failed" << endl;
    rt_close(&rt_handle);
    return -1;
}

// Close the handle
rt_close(&rt_handle);
return 0;
} // main()
Dynamic Partition Allocator APIs

The Dynamic Partition Allocator APIs provide an easy-to-use interface for the dynamic creation of partitions. This API inspects the current state of the Blue Gene/P machine and attempts to create a partition based on available resources. If no resources are available that match the partition requirements, the partition is not created. It is expected that any job scheduler that uses the partition allocator does so from a centralized process to avoid conflicts in finding free resources to build the partition. Dynamic Partition Allocator APIs are thread safe. Only 64-bit shared libraries are provided.

In this chapter, we define a list of APIs into the Midplane Management Control System (MMCS) Dynamic Partition Allocator. See Chapter 13, “Control system (Bridge) APIs” on page 195, for details about the Bridge API.

We discuss the following specific topics in this chapter and provide a sample program:

- API support
- API details
15.1 Overview of API support

In the following sections, we provide an overview of the support provided by the APIs.

15.1.1 Requirements

When writing programs to the Dynamic Partition Allocator APIs, you must meet the following requirements:

► Operating system supported

Currently, SUSE Linux Enterprise Server (SLES) 10 for PowerPC is the only supported platform.

► Languages supported

C and C++ are supported with the GNU gcc 4.1.2 level compilers. For more information and downloads, refer to the following Web address:

http://gcc.gnu.org/

► Include files

All required include files are installed in the /bgsys/drivers/ppcfloor/include directory. The include file for the dynamic allocator API is allocator_api.h.

► Library files

The Dynamic Partition Allocator APIs support 64-bit applications using dynamic linking with shared objects.

Sixty-four bit libraries: The required library files are installed in the /bgsys/drivers/ppcfloor/lib64 directory. The shared object for linking to the Bridge API is libbgpallocator.so.

The libbgpallocator.so library has dependencies on other libraries included with the Blue Gene software, including the following objects:

- libbgpbridge.so
- libbgpconfig.so
- libbgpdb.so
- libsaymessage.so
- libtableapi.so

These files are installed with the standard system installation procedure. They are contained in the bgpbase.rpm file.
Configuring environment variables

The environment variables in Table 15-1 are used to control the dynamic allocator and Bridge APIs.

### Table 15-1 Environment variables that control the Bridge APIs

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB_PROPERTY</td>
<td>Yes</td>
<td>This variable must be set to the path of the db.properties file with database connection information. For a default installation, the path to this file is /bgsys/local/etc/db.properties.</td>
</tr>
<tr>
<td>BRIDGE_CONFIG</td>
<td>Yes</td>
<td>This variable must be set to the path of the bridge.config file that contains the Bridge API configuration values. For a default installation, the path to this file is /bgsys/local/etc/bridge.config.</td>
</tr>
<tr>
<td>ALLOCATOR_DRAIN_LIST</td>
<td>No</td>
<td>This variable can be set to the path of the base partition drain list to be used if one is not specified on the call to rm_init_allocator(). When this variable is not set, the file /etc/allocator_drain.lst is used as a default if it exists.</td>
</tr>
<tr>
<td>BRIDGE_DUMP_XML</td>
<td>No</td>
<td>When set to any value, this variable causes the Bridge API to dump its in-memory XML streams to files in /tmp for debugging. When this variable is not set, the Bridge API does not dump its in-memory XML streams.</td>
</tr>
</tbody>
</table>

### 15.2 API details

In this section, we provide details about the APIs and return codes for dynamic partition allocation.

#### 15.2.1 APIs

The following APIs are used for dynamic partition allocation and are all thread safe:

- **BGALLOC_STATUS** rm_init_allocator(const char * caller_desc, const char * drain_list);

  A program should call rm_init_allocator() and pass a description that will be used as the text description for all partitions used by subsequent rm_allocate_partition() calls. For example, passing in *ABC job scheduler* causes any partitions that are created by rm_allocate_partition() to have *ABC job scheduler* as the partition description.

  The caller can also optionally specify a drain list file name that identifies the base partitions (midplanes) that will be excluded from the list of resources to consider when allocating new partitions. If NULL is passed in for the drain list file name, a default drain list is set first from the following locations:
  - The path in the environment variable ALLOCATOR_DRAIN_LIST if it exists
  - The /etc/allocator_drain.lst file if it exists

  If no drain list file is established, no base partitions are excluded. If an invalid file name is passed in, the call fails. For example, a drain list file with the following content excludes base partitions R00-M0, R00-M1, and R01-M0 when allocating resources for a partition:

  R00-M0
  R00-M1
  R01-M0
The list of resources can contain items separated by any white-space character (space, tab, new line, vertical tab, or form feed). Items found that do not match an existing resource are ignored, but an error message is logged.

```
BGALLOC_STATUS rm_allocate_partition(
    const rm_size_t size,
    const rm_connection_type_t conn,
    const rm_size3D_t shape,
    const rm_job_mode_t mode,
    const rm_psetsPerBP_t psetsPerBP,
    const char * user_name,
    const char * caller_desc,
    const char * boot_options,
    const char * ignoreBPs,
    const char * partition_id,
    char ** newpartition_id);
```

The caller to `rm_allocate_partition()` provides input parameters that describe the characteristics of the partition that should be created from available Blue Gene/P machine resources. If resources are available that match the requirements, a partition is created and allocated, and the partition name is returned to the caller along with a return code of `BGALLOC_OK`.

If both size and shape values are provided, the allocation is based on the shape value only.

The `user_name` parameter is required.

If the `caller_desc` value is NULL, the caller description specified on the call to `rm_init_allocator` is used.

The `boot_options` parameter is optional and can be NULL.

If the `ignoreBPs` parameter is not NULL, it must be a string of blank-separated base partition identifiers to be ignored. The base partitions listed in the parameter are ignored as though the partitions were included in the drain list file currently in effect.

If the `partition_id` parameter is not NULL, it can specify one of the following options:

- The name of the new partition
  
The name can be from 1 to 32 characters. Valid characters are a...z, A...Z, 0...9, - (hyphen), and _ (underscore).

- The prefix to be used for generating a unique partition name
  
The prefix can be from 1 to 16 characters, followed by an asterisk (*). Valid characters are the same as those for a new partition name. For example, if `ABC-Scheduler*` is specified as a prefix, the resulting unique partition name can be `ABC-Scheduler-27Sep1519514155`.

**Important:** The returned char * value for `newpartition_id` should be freed by the caller when it is no longer needed to avoid memory leaks.
The caller to `rm_allocate_htc_pool()` provides input parameters that describe the characteristics of the pool of HTC partitions that should be created from available Blue Gene/P machine resources. If resources are available that match the requirements, a pool of partitions is created and allocated, and a return code of BGALLOC_OK is returned.

The size parameter specifies the total number of nodes to allocate for the pool. The mode parameter is required and must be one of the modes in Table 13-23 on page 228. The psetsPerBP specifies the number of psets per base partition to be used. By specifying fewer psets per base partition, the IO ratio for the allocated partition can be effectively increased. If zero is specified, use all IO nodes available. This will be the partition size used or 32, whichever is greater.

The user_name parameter is required. If the caller_desc value is NULL, the caller description specified on the call to `rm_init_allocator` is used. If the ignoreBPs parameter is not NULL, it must be a string of blank-separated base partition identifiers to be ignored. The base partitions listed in the parameter are ignored as though the partitions were included in the drain list file currently in effect.

The `pool_id` is used as a prefix for generating unique partition names. It must be from 1 to 32 characters. Valid characters are a...z, A...Z, 0...9, -(hyphen), and _ (underscore). If the user_list parameter is not NULL, the user IDs specified are permitted to run jobs in the pool.

Multiple calls may be made to `rm_allocate_htc_pool` with the same pool id; these allocate additional resources to the pool.

```c
BGALLOC_STATUS rm_deallocate_htc_pool(
    const unsigned int in_removed,
    const char * pool_id,
    unsigned * num_removed);
```

This API deallocates the specified number of nodes from a pool.

The `in_removed` parameter specifies the minimum number of nodes to remove from the pool. If the number of nodes to remove is not a multiple of the size of the partitions in the pool, or this number is greater than the number of nodes available to be removed, less than this number may be removed. If zero is specified, the entire pool and all its resources are deallocated.

The `pool_id` parameter specifies the name of the pool.

The value returned in `num_removed` is the actual number of nodes removed from the pool. This number may be less than the number of nodes specified by `in_removed`. 
15.2.2 Return codes

When a failure occurs, the API invocation returns an error code. In addition, a failure always
generates a log message, which provides more information about the possible cause of the
problem and an optional corrective action. These log messages are used for debugging and
non-automatic recovery of failures.

The BGALLOC_STATUS return codes for the Dynamic Partition Allocator can be one of the
following types:

- **BGALLOC_OK**: Invocation completed successfully.
- **BGALLOC_ILLEGAL_INPUT**: The input to the API invocation is invalid. This result is due to
  missing required data, illegal data, and similar problems.
- **BGALLOC_ERROR**: An error occurred, such as a memory allocation problem or failure on a
  low-level call.
- **BGALLOC_NOT_FOUND**: The request to dynamically create a partition failed because required
  resources are not available.
- **BGALLOC_ALREADY_EXISTS**: A partition already exists with the name specified. This error
  occurs only when the caller indicates a specific name for the new partition.

15.3 Sample program

The sample program in Example 15-1 shows how to allocate a partition from resources on
base partition R001.

**Example 15-1  Sample allocator API program**

```cpp
#include <iostream>
#include <sstream>
#include <cstring>
#include "allocator_api.h"

using std::cout;
using std::cerr;
using std::endl;

int main() {
    rm_size3D_t shape;
    rm_connection_type_t conn = RM_MESH;
    char * ignoreBPs = "R00-M0";
    char* new_partition_id;
    shape.X = 0;
    shape.Y = 0;
    shape.Z = 0;
    BGALLOC_STATUS alloc_rc;

    //set lowest level of verbosity
    setSayMessageParams(stderr, MESSAGE_DEBUG1);
    alloc_rc = rm_init_allocator("test", NULL);
    alloc_rc = rm_allocate_partition(256, conn, shape, RM_SMP_MODE, 0,
        "user1",
        "New partition description",
        ignoreBPs,
        "",
```
"ABC-Scheduler",
&new_partition_id);

if (alloc_rc == BGALLOC_OK) {
    cout << "successfully allocated partition: " << new_partition_id << endl;
    free(new_partition_id);
} else {
    cerr << "could not allocate partition: " << endl;
    if (alloc_rc == BGALLOC_ILLEGAL_INPUT) {
        cerr << "illegal input" << endl;
    } else if (alloc_rc == BGALLOC_ERROR) {
        cerr << "unknown error" << endl;
    } else if (alloc_rc == BGALLOC_NOT_FOUND) {
        cerr << "not found" << endl;
    } else if (alloc_rc == BGALLOC_ALREADY_EXISTS) {
        cerr << "partition already exists" << endl;
    } else {
        cerr << "internal error" << endl;
    }
}

Example 15-2 shows the commands used to compile and link the sample program.

Example 15-2  The compile and link commands

```bash

```
g++ -m64 -pthread -I/bgsys/drivers/ppcfloor/include -c sample1.cc -o sample1.o_64
g++ -m64 -pthread -o sample1 sample1.o_64 -L/bgsys/drivers/ppcfloor/lib64 -lbgpallocator
```

"ABC-Scheduler",
&new_partition_id);

if (alloc_rc == BGALLOC_OK) {
    cout << "successfully allocated partition: " << new_partition_id << endl;
    free(new_partition_id);
} else {
    cerr << "could not allocate partition: " << endl;
    if (alloc_rc == BGALLOC_ILLEGAL_INPUT) {
        cerr << "illegal input" << endl;
    } else if (alloc_rc == BGALLOC_ERROR) {
        cerr << "unknown error" << endl;
    } else if (alloc_rc == BGALLOC_NOT_FOUND) {
        cerr << "not found" << endl;
    } else if (alloc_rc == BGALLOC_ALREADY_EXISTS) {
        cerr << "partition already exists" << endl;
    } else {
        cerr << "internal error" << endl;
    }
}

Example 15-2 shows the commands used to compile and link the sample program.

Example 15-2  The compile and link commands

```bash

```
g++ -m64 -pthread -I/bgsys/drivers/ppcfloor/include -c sample1.cc -o sample1.o_64
g++ -m64 -pthread -o sample1 sample1.o_64 -L/bgsys/drivers/ppcfloor/lib64 -lbgpallocator
```

"ABC-Scheduler",
&new_partition_id);

if (alloc_rc == BGALLOC_OK) {
    cout << "successfully allocated partition: " << new_partition_id << endl;
    free(new_partition_id);
} else {
    cerr << "could not allocate partition: " << endl;
    if (alloc_rc == BGALLOC_ILLEGAL_INPUT) {
        cerr << "illegal input" << endl;
    } else if (alloc_rc == BGALLOC_ERROR) {
        cerr << "unknown error" << endl;
    } else if (alloc_rc == BGALLOC_NOT_FOUND) {
        cerr << "not found" << endl;
    } else if (alloc_rc == BGALLOC_ALREADY_EXISTS) {
        cerr << "partition already exists" << endl;
    } else {
        cerr << "internal error" << endl;
    }
}
In this part, we discuss applications that are being used on the Blue Gene/L or Blue Gene/P system. This part includes Chapter 16, “Performance overview of engineering and scientific applications” on page 267.
Performance overview of engineering and scientific applications

In this chapter, we briefly describe a series of scientific and engineering applications that are currently being used on either the Blue Gene/L or Blue Gene/P system. For a comprehensive list of applications, refer to the IBM Blue Gene Web page at:

http://www-03.ibm.com/servers/deepcomputing/bluegene/siapps.html

The examples in this chapter emphasize the benefits of using the Blue Gene supercomputer as a highly scalable parallel system. They present results for running applications in various modes that exploit the architecture of the system. We discuss the following topics:

- Blue Gene/P system from an applications perspective
- Chemistry and life sciences applications
16.1 Blue Gene/P system from an applications perspective

This book has been dedicated to describing the Blue Gene/P massively parallel supercomputer from IBM. In this section, we summarize the benefits of the Blue Gene/P system from an applications point of view. We recall that at the core of the system is the IBM PowerPC (PowerPC 450) processor with the addition of two floating-point units (FPU). This system uses a distributed memory, message-passing programming model.

To achieve a high level of integration and quantity of micro-processors with low power consumption, the machine was developed based on a processor with moderate frequency. The Blue Gene/P system uses system-on-a-chip (SoC) technology to allow a high level of integration, low power, and low design cost. Each processor core runs at a frequency of 850 MHz giving a theoretical peak performance of 3.4 gigaflops/core or 13.6 gigaflops/chip. The chip constitutes the compute node.

The next building blocks are the compute and I/O cards. A single compute node attached to a processor card with 2 GB of memory (RAM) creates the compute and I/O cards. The compute cards and I/O cards are plugged into a node card. Two rows of sixteen compute cards are on the node card. There can be up to two I/O cards per node card.

A midplane consists of 16 node cards stacked in a rack. A rack holds two midplanes, for a total of 32 node cards. A system with 72 racks consisting of 294,912 processor cores.

In 2005, running a real application on the Blue Gene/L system broke the barrier of 100 teraflops/second (TF/s), sustaining performance using the domain decomposition molecular-dynamics code (ddcMD) from the Lawrence Livermore National Laboratory. In 2006, the first system to break the barrier of 200 TF/s was Qbox running at 207.3 TF/s. Real applications are currently achieving two orders of magnitude higher performance than previously possible. Successful scaling has pushed from $O(1000)$ processors to $O(100,000)$ processors by the Gordon Bell Prize finalists at Supercomputing 2006. Out of six finalists, three ran on the Blue Gene/L system.

In silico experimentation plays a crucial role in many scientific disciplines. It provides a fingerprint for experimentation. In engineering applications, such as automotive crash studies, numerical simulation is much cheaper than physical experimentation. In other applications, such as global climate change where experiments are impossible, simulations are used to explore the fundamental scientific issues. This is certainly true in life sciences as well as in materials science. Figure 16-1 on page 269 illustrates a landscape of a few selected areas and techniques where high-performance computing is important to carry out simulations.
In the rest of this chapter, we summarize the performance that has been recorded in the literature for a series of applications in life sciences and materials science. A comprehensive list of applications is available for the Blue Gene/L and Blue Gene/P systems. For more information, see the IBM Blue Gene Applications Web page at:
http://www-03.ibm.com/servers/deepcomputing/bluegene/siapps.html

16.2 Chemistry and life sciences applications

In this section, we provide a brief overview of the performance characteristics of a selected set of chemistry and life sciences applications. In particular, we focus on what is known as computational chemistry. However, as other disciplines in sciences that traditionally relied almost exclusively on experimental observation began to fully incorporate Information Technology (IT) as one of their tools, the area of computational chemistry has expanded to new disciplines such as bioinformatics, systems biology, and several other areas that have emerged after the post-genomic era.

To understand or define the kind of molecular systems that can be studied with these techniques, Figure 16-2 on page 270 defines the computational chemistry landscape as a function of the size of the systems and the methodology. It illustrates that Classical Molecular Mechanics/Molecular Dynamics (MM/MD) are commonly used to simulate large biomolecules that cannot be treated with more accurate methods. The next level corresponds to semi-empirical methods. Finally Ab Initio methods (also called electronic structure methods)
provide a more accurate description of the system, but the computational demands in terms of compute cycles increase rapidly.

Alternatively, bioinformatics techniques rely mainly on string manipulations in an effort to carry out data mining of large databases. These applications tend to be data intensive.

Although Density Functional Theory-based approaches are not fully represented in Figure 16-2, nowadays these types of methods are being used to simulate biologically important systems. These techniques allow for the calculation of larger systems. In this chapter, we briefly describe Car-Parrinello Molecular Dynamics (CPMD). In the same vein, use of mixed Quantum Mechanical/Molecular Mechanical (QM/MM) methods can simulate larger systems.

16.2.1 Classical molecular mechanics and molecular dynamics applications

Applications in such areas as chemistry and life sciences can benefit from the type of architecture used in the Blue Gene supercomputer. In particular, software packages based on molecular dynamics have been considered good candidates for the Blue Gene architecture. Classical MD simulations compute atomic trajectories by solving equations of motion numerically by using empirical force fields. The overall MD energy equation is broken into three components: bonded, van der Waals, and electrostatic. The first two components are local in nature and therefore do not make a significant contribution to the overall running time.

The quadratic scaling of the electrostatics force terms, however, requires a high level of optimization of the MD application. To improve performance on simulations in which the solvent is modeled at the atomic level (that is, explicit solvent modeling), the four Blue Gene MD applications of AMBER, Blue Matter, LAMMPS, and NAMD employ a reciprocal-space technique called Ewald sums, which enables the evaluation of long-range electrostatic forces to a preselected level of accuracy. In addition to the particle mesh Ewald (PME) method, LAMMPS offers the particle particle/particle-mesh (PPPM) technique with characteristics that make it scale well on massively parallel processing (MPP) machines such as the Blue Gene system.
AMBER
AMBER\textsuperscript{16} is the collective name for a suite of programs that are developed by the Scripps Research Institute. With these programs, users can carry out molecular dynamics simulations, particularly on biomolecules. The primary AMBER module, called \textit{sander}, was designed to run on parallel systems and provides direct support for several force fields for proteins and nucleic acids. AMBER includes an extensively modified version of \textit{sander}, called \textit{pmemd} (particle mesh). For complete information about AMBER as well as benchmarks, refer to the AMBER Web site at:

http://amber.scripps.edu/

For implicit solvent (continuum) models, which rely on variations of the Poisson equation of classical electrostatics, AMBER offers the Generalized Born (GB) method. This method uses an approximation to the Poisson equation that can be solved analytically and allows for good scaling. In Figure 16-3, the experiment is with an implicit solvent (GB) model of 120,000 atoms (Aon benchmark).

![Figure 16-3  Parallel scaling of AMBER on the Blue Gene/L system](image-url)
AMBER also incorporates the PME algorithm, which takes the full electrostatic interactions into account to improve the performance of electrostatic force evaluation (see Figure 16-4). In Figure 16-4, the experiment is with an explicit solvent (PME) model of 290,000 atoms (Rubisco).

**Blue Matter**

Blue Matter\(^47\) is a classical molecular dynamics application that has been under development as part of the IBM Blue Gene project. The effort serves two purposes:

- Enables scientific work in the area of biomolecular simulation that IBM announced in December 1999.
- Acts as an experimental platform for the exploration of programming models and algorithms for massively parallel machines in the context of a real application.

Blue Matter has been implemented via spatial-force decomposition for N-body simulations using the PME method for handling electrostatic interactions. The Ewald summation method and particle mesh techniques are approximated by a finite range cut-off and a reciprocal space portion for the charge distribution. This is done in Blue Matter via the Particle-Particle-Particle-Mesh (P3ME) method.\(^48\)

The results presented by Fitch et al.\(^49\) show impressive scalability on the Blue Gene/L system. Figure 16-5 on page 273 shows scalability as a function of the number of nodes. It illustrates that the performance in time/time step as a function of the number of processors for β-Hairpin contains a total of 5,239 atoms. SOPE contains 13,758 atoms. In this case, the timings that are reported here correspond to a size of 64\(^3\) FFT. Rhodopsin contains 43,222 atoms, and ApoA1 contains 92,224 atoms. All runs were carried out using the P3ME method, which was implemented in Blue Matter at constant particle number, volume, and energy (NVE).\(^51\)
LAMMPS

Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS)\textsuperscript{52} is an MD program from Sandia National Laboratories that is designed specifically for MPP. LAMMPS is implemented in C++ and is distributed freely as open-source software under the GNU Public License (GPL).\textsuperscript{53} LAMMPS can model atomic, polymeric, biological, metallic, or granular systems using a variety of force fields and boundary conditions. The parallel efficiency of LAMMPS varies from the size of the benchmark data and the number of steps being simulated. In general, LAMMPS can scale to more processors on larger systems (see Figure 16-6).

For a one-million atom system, LAMMPS can scale up to 4096 nodes. For a larger system, such as a four-million atom system, LAMMPS can scale up to 4096 nodes as well. As the size of the system increases, scalability increases as well.
NAMD
NAMD is a parallel molecular dynamics application that was developed for high-performance calculations of large biological molecular systems. NAMD supports the force fields used by AMBER, CHARMM, and X-PLOR and is also file compatible with these programs. This commonality allows simulations to migrate between these four programs. The C++ source for NAMD and Charm++ are freely available from UIUC. For additional information about NAMD, see the official NAMD Web site at:

http://www.ks.uiuc.edu/Research/namd/

NAMD incorporates the PME algorithm, which takes the full electrostatic interactions into account and reduces computational complexity. To further reduce the cost of the evaluation of long-range electrostatic forces, a multiple time step scheme is employed. The local interactions (bonded, van der Waals, and electrostatic interactions within a specified distance) are calculated at each time step. The longer range interactions (electrostatic interactions beyond the specified distance) are computed less often. An incremental load balancer monitors and adjusts the load during the simulation.

Due to the good balance of network and processor speed of the Blue Gene system, NAMD is able to scale to large processor counts (see Figure 16-7). While scalability is affected by many factors, many simulations can make use of multiple Blue Gene racks. Work by Kumar et al. has reported scaling up to 8192 processors. Timing comparisons often use the “benchmark time” metric instead of wall clock time to completion. The benchmark time metric omits setup, I/O, and load balance overhead. While benchmark scaling can be considered a guide to what is possible, ideal load balance and I/O parameters for each case must be found for the wall clock time to scale similarly. Careful consideration of these parameters might be necessary to achieve the best scalability.

![Figure 16-7  Parallel speedup on the Blue Gene/L system for the NAMD standard apoA1 benchmark](image)

16.2.2 Molecular docking applications

Applications in the area of molecular docking are becoming important in high-performance computing. In particular, in silico screening using molecular docking has been recognized as an approach that benefits from high-performance computing to identify novel small molecules that can then be used for drug design. This process consists of the identification or selection of compounds that show activity against a biomolecule that is of interest as a drug target.

Docking programs place molecules into the active site of the receptor (or target biomolecule) in a noncovalent fashion and then rank them by the ability of the small molecules to interact
with the receptor. An extensive family of molecular docking software packages is available.

**DOCK6**

DOCK is an open-source molecular docking software package that is frequently used in structure-based drug design. The computational aspects of this program can be divided into two parts. The first part consists of the *ligand atoms* located inside the cavity or binding pocket of a receptor, which is a large biomolecule. This step is carried out by a search algorithm. The second part corresponds to scoring or identifying the most favorable interactions, which is normally done by means of a scoring function.

The latest version of the DOCK software package is Version 6.1. However, in our work, we used Version 6.0. This version is written in C++ to exploit code modularity and has been parallelized using the Message Passing Interface (MPI) paradigm. DOCK V6.0 is parallelized using a master-worker scheme. The master handles I/O and tasks management, while each worker is given an individual molecule to perform simultaneous independent docking.

Recently, Peters, et al. have shown that DOCK6 is well suited for doing virtual screening on the Blue Gene/L or Blue Gene/P system. Figure 16-8 shows the receptor HIV-1 reverse transcriptase in complex with nevirapine as used and described in the Official UCSF DOCK Web site. The ligand library corresponds to a subset of 27,005 drug-like ligands from the ZINC database. The scalability of the parallel version of the code is illustrated by constructing a set of ligands with 128,000 copies of nevirapine as recommended in the Official UCSF DOCK Web site to remove dependence on the order and size of the compound. You can find this Web site at: [http://dock.compbio.ucsf.edu](http://dock.compbio.ucsf.edu)

In Figure 16-8, the original code is the dark bar. Sorting by total number of atoms per ligand is represented by the bar with horizontal lines. Sorting by total number of rotatable bonds per ligand is represented by the white bar.

![Figure 16-8](image_url) *Figure 16-8  The effect of load-balancing optimization for 27,005 ligands on 2048 processors*
16.2.3 Electronic structure (Ab Initio) applications

Electronic structure calculations, such as the Hartree-Fock (HF) method, represent one of the simplest techniques in this area. However, even this first approximation tends to be computationally demanding. Many types of calculations begin with a Hartree-Fock calculation and subsequently correct for electron-electron repulsion, which is also referred to as electronic correlation. The Möller-Plesset perturbation theory (MPn) and coupled cluster theory (CC) are examples of these post-Hartree-Fock methods.\textsuperscript{70}

A common characteristic of these techniques is that they are used to accurately compute molecular properties. As such, they tend to be widely available in high-performance computing. However, in addition to traditional electronic structure methods, Density Functional Theory-based methods have proven to be an attractive alternative to include correction effects and still treat large systems.

CPMD

The CPMD code is based on the original computer code written by Car and Parrinello.\textsuperscript{71} It was developed first at the IBM Research Zurich laboratory, in collaboration with many groups worldwide. It is a production code with many unique features written in Fortran and has grown from its original size of approximately 10,000 lines to currently close to 200,000 lines of code. Since January 2002, the program has been freely available for noncommercial use.\textsuperscript{72}

The basics of the implementation of the Kohn-Sham method using a plane-wave basis set and pseudopotentials are described in several review articles,\textsuperscript{73} and the CPMD code follows them closely. All standard gradient-corrected density functionals are supported, and preliminary support for functionals that depend on the kinetic energy density is available. Pseudopotentials used in CPMD are either of the norm-conserving or the ultra-soft type.\textsuperscript{74} Norm-conserving pseudopotentials have been the default method in CPMD, and only some of the rich functionality has been implemented for ultra-soft pseudopotentials.

The emphasis of CPMD on MD simulations of complex structures and liquids led to the optimization of the code for large supercells and a single k-point (the k = 0 point) approximation. Therefore, many features have only been implemented for this special case. CPMD has a rich set of features, many of which are unique. For a complete overview, refer to the CPMD manual.\textsuperscript{75} The basic electronic structure method implemented uses fixed occupation numbers, either within a spin-restricted or an unrestricted scheme. For systems with a variable occupation number (small gap systems and metals), the free energy functional\textsuperscript{3} can be used together with iterative diagonalization methods.

16.2.4 Bioinformatics applications

The list of molecular biology databases is constantly increasing, and more scientists rely on this information. The NAR Molecular Biology Database collection reported an increase of 139 more databases for 2006 compared to the previous year. enBank doubles its size approximately every 18 months. However, the increase in microprocessor clock speed is not changing at the same rate. Therefore, scientists try to leverage the use of multiple processors. In this section, we introduce some of the applications currently running on the Blue Gene supercomputer.
HMMER

For a complete discussion of hidden Markov models, refer to the work by Krogh et al.\textsuperscript{76} HMMER V2.3.2 consists of nine different programs: hmmalign, hmmbuild, hmmcalibrate, hmmconvert, hmmemit, hmmfetch, hmmindex, hmmpfam, and hmmsearch.\textsuperscript{77} Out of these nine programs, hmmcalibrate, hmmpfam, and hmmsearch have been parallelized. hmmcalibrate is used to identify statistical significance parameters for profile HMM. hmmpfam is used to search a profile HMM database, and hmmsearch is used to carry out sequence database searches.\textsuperscript{78}

The first module tested corresponds to hmmcalibrate. Figure 16-9 summarizes the performance of this module up to 2048 nodes.\textsuperscript{79} Although this module was not optimized, the parallel efficiency is still 75% on 2048 nodes. The graph in Figure 16-9 illustrates the performance of hmmcalibrate using only the first 327 entries in the Pfam database.\textsuperscript{80}

Figure 16-9 .hmmcalibrate parallel performance using the first 327 entries of the Pfam database

Figure 16-10 on page 278 illustrates the work presented by Jiang, et al.\textsuperscript{79} for optimizing hmmsearch parallel performance using 50 proteins of the globin family from different organisms and the UniProt release 8 database. For each processor count, the left bar shows the original PVM to MPI port. Notice scaling stops at 64 nodes. The second bar shows the multiple master implementation. The third bar shows the dynamic data collection implementation, and the right bar shows the load balancing implementation.

Figure 16-10 .hmmsearch parallel performance using 50 proteins of the globin family from different organisms and the UniProt release 8 database
mpiBLAST-PIO

mpiBLAST is an open-source parallelization of BLAST that uses MPI.\textsuperscript{82} One of the key features of the initial parallelization of mpiBLAST is its ability to fragment and distribute databases.

Thorsen et al.\textsuperscript{83} have compared the query Arabidopsis thaliana, a model organism for studying plant genetics. This query was further subdivided into small, medium, and large query sets that contain 200, 1168, and 28014 sequences, respectively.

Figure 16-11 on page 279 illustrates the results of comparing three queries of three different sizes. We labeled them “small,” “medium,” and “large.” The database corresponds to NR. This figure shows that scalability is a function of the query size. The small query scales to approximately 1024 nodes in coprocessor mode with a parallel efficiency of 72% where the large query scales to 8,192 nodes with a parallel efficiency of 74%.
From the top of Figure 16-11, the thick solid line corresponds to ideal scaling. The thin solid line corresponds to the large query. The dashed line corresponds to the medium query. The dotted line corresponds to the small query.

![Figure 16-11](Scaling chart for queries run versus the nr database)

### 16.2.5 Performance kernel benchmarks

Communication performance is an important aspect when running parallel applications, particularly, when running on a distributed-memory system such as the Blue Gene/P system. On both the Blue Gene/L and Blue Gene/P systems, instead of implementing a single type of network capable of transporting all protocols needed, these two systems have separate networks for different types of communications.

Usually two measurements provide information about the network and can be used to look at the parallel performance of applications:

- **Bandwidth**: The number of MB of data that can be sent from a node to another node in one second
- **Latency**: The amount of time it takes for the first byte sent from one node to reach its target node

These two values provide information about communication. In this section, we illustrate two simple cases. The first case corresponds to a benchmark that involves a single transfer. The second case corresponds to a collective as defined in the Intel MPI Benchmarks. Intel MPI Benchmarks is formerly known as “Pallas MPI Benchmarks” - PMB-MPI1 (for MPI1 standard functions only). Intel MPI Benchmarks - MPI1 provides a set of elementary MPI benchmark kernels.

For more details, see the product documentation included in the package that you can download from the Web at:

Intel MPI Benchmarks

The Intel MPI Benchmarks kernel or elementary set of benchmarks was reported as part of *Unfolding the IBM eServer Blue Gene Solution*, SG24-6686. Here we describe and perform the same benchmarks. You can run all of the supported benchmarks, or just a subset, specified through the command line. The rules, such as time measurement, message lengths, selection of communicators to run a particular benchmark, are program parameters. For more information, see the product documentation that is included in the package, which you can download from the Web at:

http://www.intel.com/software/products/cluster/mpi/mpi_benchmarks_lic.htm

This set of benchmarks has the following objectives:

- Provide a concise set of benchmarks targeted at measuring important MPI functions: point-to-point message-passing, global data movement and computation routines, and one-sided communications and file I/O
- Set forth precise benchmark procedures: run rules, set of required results, repetition factors, and message lengths
- Avoid imposing an interpretation on the measured results: execution time, throughput, and global operations performance

16.2.6 MPI point-to-point

In the Intel MPI Benchmarks, single transfer corresponds to PingPong and PingPing benchmarks. Here we illustrate a comparison between the Blue Gene/L and Blue Gene/P system for the case of PingPong. This benchmark illustrates a single message that was transferred between two MPI tasks, which in our case, is on two different nodes.

To run this benchmark, we used the Intel MPI Benchmark Suite Version 2.3, MPI-1 part. On the Blue Gene/L system, the benchmark was run in coprocessor mode, which is defined in *Unfolding the IBM eServer Blue Gene Solution*, SG24-6686. On the Blue Gene/P system, we used the SMP node mode.

Example 16-1 shows how *mpirun* was invoked on the Blue Gene/L system.

**Example 16-1 mpirun on the Blue Gene/L system**

```
mpirun -nofree -timeout 120 -verbose 1 -mode CO -env "BGL_APP_L1_WRITE_THROUGH=0 BGL_APP_L1_SWOA=0" -partition R000 -cwd /bglscratch/pallas -exe /bglscratch/pallas/IMB-MPI1.4MB.perf.rts -args ":msglen 4194304.txt -npmin 512 PingPong" | tee IMB-MPI1.4MB.perf.PingPong.4194304.512.out) >> run.IMB-MPI1.4MB.perf.PingPong.4194304.512.out 2>&1
```

Example 16-2 shows how *mpirun* was invoked on the Blue Gene/P system.

**Example 16-2 mpirun on the Blue Gene/P system**

```
mpirun -nofree -timeout 300 -verbose 1 -np 512 -mode SMP -partition R01-M1 -cwd /bgusr/BGTH_BGP/test512nDD2BGP/pallas/pal1512DD2SMP/bgpdd2sys1-R01-M1 -exe /bgusr/BGTH_BGP/test512nDD2BGP/pallas/pal1512DD2SMP/bgpdd2sys1-R01-M1/IMB-MPI1.4MB.perf.rts -args "-msglen 4194304.txt -npmin 512 PingPong" | tee IMB-MPI1.4MB.perf.PingPong.4194304.512.out) >> run.IMB-MPI1.4MB.perf.PingPong.4194304.512.out 2>&1
```
Figure 16-12 shows the bandwidth on the torus network as a function of the message size, for one simultaneous pair of nearest neighbor communications. The protocol switch from short to eager is visible in these two cases, where the eager to rendezvous switch is most pronounced on the Blue Gene/L system. This figure also shows the improved performance on the Blue Gene/P system. Notice also in Figure 16-12 that the diamonds correspond to the Blue Gene/P system and the asterisks (*) correspond to the Blue Gene/L system.

**Figure 16-12  Bandwidth versus message size**

**MPI collective benchmarks**

In the Intel MPI Benchmarks, collective benchmarks correspond to Bcast, Allgather, Allgatherv, Alltoall, Alltoallv, Reduce, Reduce_scatter, Allreduce, and Barrier benchmarks. Here we illustrate a comparison between the Blue Gene/L and Blue Gene/P system for the case of Allreduce, which is a popular collective used in certain scientific applications. These benchmarks measure the message-passing power of a system as well as the quality of the implementation.

To run this benchmark, we used the Intel MPI Benchmark Suite Version 2.3, MPI-1 part. On the Blue Gene/P system, the benchmark was run in coprocessor mode, which is defined in *Unfolding the IBM eServer Blue Gene Solution*, SG24-6686. On the Blue Gene/P system, we used SMP node mode.

Example 16-3 shows how `mpirun` was invoked on the Blue Gene/L system.

**Example 16-3  `mpirun` on the Blue Gene/L system**

```
mpirun -nofree -timeout 120 -verbose 1 -mode CO -env "BGL_APP_L1_WRITE_THROUGH=0 BGL_APP_L1_SWOA=0" -partition RO00 -cwd /bglscratch/8GTH/testsmall1512nodeBGL/pallas -exe /bglscratch/8GTH/testsmall1512nodeBGL/pallas/IMB-MPI1.4MB.perf.rts -args "-msglen 4194304.txt -npmin 512 Allreduce" | tee IMB-MPI1.4MB.perf.Allreduce.4194304.512.out) >> run.IMB-MPI1.4MB.perf.Allreduce.4194304.512.out 2>&1
```
Example 16-4 shows how `mpirun` was invoked on the Blue Gene/P system.

Example 16-4  mpirun on the Blue Gene/P system

```bash
mpirun -nofree -timeout 300 -verbose 1 -np 512 -mode SMP -partition R01-M1 -cwd /bgusr/BGTH_BGP/test512nDD2BGP/pallas/pali512DD2SMP/bgpdd2sys1-R01-M1 -exe /bgusr/BGTH_BGP/test512nDD2BGP/pallas/pali512DD2SMP/bgpdd2sys1-R01-M1/IMB-MPI1.4MB .perf.rts -args "-msglen 4194304.txt -npmin 512 Allreduce" | tee IMB-MPI1.4MB.perf.Allreduce.4194304.512.out) >> run.IMB-MPI1.4MB.perf.Allreduce.4194304.512.out 2>&1
```

Collective operations are more efficient on the Blue Gene/P system. You should try to use these operations instead of point-to-point communication wherever possible. The overhead for point-to-point communications is much larger than those for collectives. Unless all your point-to-point communication is purely the nearest neighbor, it is also difficult to avoid network congestion on the torus network.

Alternatively, collective operations can use the barrier (global interrupt) network or the torus network. If they run over the torus network, they can still be optimized by using specially designed communication patterns that achieve optimum performance. Doing this manually with point-to-point operations is possible in theory, but in general, the implementation in the Blue Gene/P MPI library offers superior performance.

With point-to-point communication, the goal of reducing the point-to-point Manhattan distances necessitates a good mapping of MPI tasks to the physical hardware. For collectives, mapping is equally important because most collective implementations prefer certain communicator shapes to achieve optimum performance. The technique of mapping is illustrated in Appendix E, “Mapping” on page 303.

Similar to point-to-point communications, collective communications also works best if you do not use complicated derived data types and if your buffers are aligned to 16-byte boundaries.

While the MPI standard explicitly allows for MPI collective communications to occur at the same time as point-to-point communications (on the same communicator), we generally do not recommend that you allow this to happen for performance reasons.

Table 16-1 summarizes the MPI collectives that have been optimized on the Blue Gene/P system, together with their performance characteristics when executed on the various networks of the Blue Gene/P system.

<table>
<thead>
<tr>
<th>MPI routine</th>
<th>Condition</th>
<th>Network</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>(MPI_BARRIER)</td>
<td>MPI_COMM_WORLD</td>
<td>Barrier (global interrupt) network</td>
<td>1.2 μs</td>
</tr>
<tr>
<td>(MPI_BARRIER)</td>
<td>Any communicator</td>
<td>Torus network</td>
<td>30 μs</td>
</tr>
<tr>
<td>(MPI_BROADCAST)</td>
<td>MPI_COMM_WORLD</td>
<td>Collective network</td>
<td>817 MBps</td>
</tr>
<tr>
<td>(MPI_BROADCAST)</td>
<td>Rectangular communicator</td>
<td>Torus network</td>
<td>934 MBps</td>
</tr>
<tr>
<td>(MPI_ALLREDUCE)</td>
<td>MPI_COMM_WORLD fixed-point</td>
<td>Collective network</td>
<td>778 MBps</td>
</tr>
<tr>
<td>(MPI_ALLREDUCE)</td>
<td>MPI_COMM_WORLD floating point</td>
<td>Collective network</td>
<td>98 MBps</td>
</tr>
</tbody>
</table>
Figure 16-13 shows a comparison between the Blue Gene/L and Blue Gene/P systems for the MPI_Allreduce() type of communication.

<table>
<thead>
<tr>
<th>MPI routine</th>
<th>Condition</th>
<th>Network</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Alltoall[v]</td>
<td>Any communicator</td>
<td>Torus network</td>
<td>84-97% peak</td>
</tr>
<tr>
<td>MPI_Allgatherv</td>
<td>N/A</td>
<td>Torus network</td>
<td>Same as broadcast</td>
</tr>
</tbody>
</table>

Figure 16-13  MPI_Allreduce() performance on 512 nodes

Figure 16-14 illustrates the performance of the barrier on Blue Gene/P for up to 32 nodes.

Figure 16-14  Barrier performance on the Blue Gene/P system
Appendixes

In this part, we provide additional information about system administration for the Blue Gene/P system. This part includes the following appendixes:

- Appendix A, “Blue Gene/P hardware-naming conventions” on page 287
- Appendix B, “Header files and libraries” on page 293
- Appendix C, “Files on architectural features” on page 297
- Appendix D, “Porting applications” on page 301
- Appendix E, “Mapping” on page 303
- Appendix F, “htcpartition” on page 307
Blue Gene/P hardware-naming conventions

In this appendix, we present an overview of how the Blue Gene/P hardware locations are assigned. These naming conventions are used consistently throughout both hardware and software.
Figure A-1 shows the conventions used when assigning locations to all hardware except the various cards in a Blue Gene/P system. Using the charts and diagrams that follow, consider an example where you have an error in the fan named R23-M1-A3-0. This naming convention tells you where to look for the error. In the upper-left corner of Figure A-1, you see that racks use the convention Rxx. Looking at our error message, we can see that the rack involved is R23. From the chart in Figure A-1, we see that R23 is the fourth rack in row two. (Remember that all numbering starts with 0). The bottom midplane of any rack is 0. Therefore, we are dealing with the top midplane (R23-M1).

In the chart, you can see in the fan assemblies description that assemblies 0-4 are on the front of the rack, bottom to top, respectively. Therefore, we check for an attention light (Amber LED) on the fan assembly second from the top, because the front-most fan is the one that is causing the error message to surface. Service, link, and node cards use a similar form of addressing.
Figure A-2 shows the conventions used for the various card locations.

Table A-1 contains examples of various hardware conventions. The figures that follow the table provide illustrations of the actual hardware.

Table A-1  Examples of hardware-naming conventions

<table>
<thead>
<tr>
<th>Card</th>
<th>Element</th>
<th>Name</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute</td>
<td>Card</td>
<td>J04 through J35</td>
<td>R23-M10-N02-J09</td>
</tr>
<tr>
<td>I/O</td>
<td>Card</td>
<td>J00 through J01</td>
<td>R57-M1-N04-J00</td>
</tr>
<tr>
<td>I/O &amp; Compute</td>
<td>Module</td>
<td>U00</td>
<td>R23-M0-N13-J08-U00</td>
</tr>
<tr>
<td>Link</td>
<td>Module</td>
<td>U00 through U05 (00 leftmost, 05 rightmost)</td>
<td>R32-M0-L2_U03</td>
</tr>
<tr>
<td>Link</td>
<td>Port</td>
<td>TA through TF</td>
<td>R01-M0-L1-U02-TC</td>
</tr>
<tr>
<td>Link data cable</td>
<td>Connector</td>
<td>J00 through J15 (as labeled on link card)</td>
<td>R21-M1-L2-J13</td>
</tr>
<tr>
<td>Node Ethernet</td>
<td>Connector</td>
<td>EN0, EN1</td>
<td>R16-M1-N14-EN1</td>
</tr>
<tr>
<td>Service</td>
<td>Connector</td>
<td>Control FPGA, control network, Clock R, Clock B</td>
<td>R05-M0-S-Control FPGA</td>
</tr>
<tr>
<td>Clock</td>
<td>Connector</td>
<td>Input, Output 0 through Output 9</td>
<td>R13-K-Output 3</td>
</tr>
</tbody>
</table>
Figure A-3 shows the layout of a 64-rack system.

![Figure A-3 - Rack numbering](image)

**Note:** The fact that Figure A-3 shows numbers 00 through 77 does not imply that this configuration is the largest possible. The largest configuration possible is 256 racks numbered 00 through FF.

Figure A-4 identifies each of the cards in a single midplane.

![Figure A-4 - Positions of the node, link, and service cards](image)

**Note:** N00-J23 is torus position 0,0,0.
Figure A-5 shows a diagram of a node card. On the front of the card are Ethernet ports EN0 and EN1. The first nodes behind the Ethernet ports are the I/O nodes. In this diagram, the node card is fully populated with I/O nodes, meaning that it has two I/O nodes. Behind the I/O nodes are the compute nodes.

Figure A-6 is an illustration of a service card.
Figure A-7 shows the link card. The locations identified as J00 through J15 are the link card connectors. The link cables are routed from one link card to another to form the torus network between the midplanes.

Figure A-8 shows the clock card. If the clock is a secondary or tertiary clock, a cable comes to the input connector on the far right. Next to the input (just to the left) is the master and worker toggle switch. All clock cards are built with the capability of filling either role. If the clock is a secondary or tertiary clock, this must be set to worker. Output zero through nine can be used to send signals to midplanes throughout the system.
Header files and libraries

In this appendix, we provide information about selected header files and libraries for the Blue Gene/P system. Directories that contain header files and libraries for the Blue Gene/P system are under the main system path in the /bgsys/drivers/ppcfloor directory.
Blue Gene/P applications

Blue Gene/P applications run on the Blue Gene/P compute or I/O nodes. Table B-1 describes the header files in the /bgsys/drivers/ppcfloor/comm/include directory.

**Table B-1  Header files in /bgsys/drivers/ppcfloor/comm/include**

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcmf.h</td>
<td>Common BGP message layer interface</td>
</tr>
<tr>
<td>dcmf_collectives.h</td>
<td>Common BGP message layer interface for general collectives</td>
</tr>
<tr>
<td>mpe_thread.h</td>
<td>Multi-processing environment (MPE) routines</td>
</tr>
<tr>
<td>mpicxx.h</td>
<td>MPI GCC script routine naming</td>
</tr>
<tr>
<td>mpi.h</td>
<td>MPI Fortran parameters</td>
</tr>
<tr>
<td>mpi.h</td>
<td>MPI C defines</td>
</tr>
<tr>
<td>mpiio.h</td>
<td>MPI I/O Fortran programs</td>
</tr>
<tr>
<td>mpiio.h</td>
<td>MPI I/O C includes</td>
</tr>
<tr>
<td>mpix.h</td>
<td>Blue Gene/P extensions to the MPI specifications</td>
</tr>
</tbody>
</table>

Table B-2 describes the header files in the /bgsys/drivers/ppcfloor/arch/include/common directory.

**Table B-2  Header files in /bgsys/drivers/ppcfloor/arch/include/common**

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bgp_personality.h</td>
<td>Defines personality</td>
</tr>
<tr>
<td>bgp_personality_inlines.h</td>
<td>Static inline for personality</td>
</tr>
<tr>
<td>bgp_personalityP.h</td>
<td>Defines personality processing</td>
</tr>
</tbody>
</table>

Table B-3 describes the 32-bit static and dynamic libraries in the /bgsys/drivers/ppcfloor/comm/lib directory.

**Table B-3  32-bit static and dynamic libraries in /bgsys/drivers/ppcfloor/comm/lib/**

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>libdcmf.cnk.a,</td>
<td>Common BGP message layer interface in C</td>
</tr>
<tr>
<td>libdcmf.cnk.so</td>
<td></td>
</tr>
<tr>
<td>libdcmfcoll.cnk.a,</td>
<td>Common BGP message layer interface for general collectives in C</td>
</tr>
<tr>
<td>libdcmfcoll.cnk.so</td>
<td></td>
</tr>
<tr>
<td>libmpich.cnk.a,</td>
<td>C bindings for MPI</td>
</tr>
<tr>
<td>libmpich.cnk.so</td>
<td></td>
</tr>
<tr>
<td>libcxxmpich.cnk.a,</td>
<td>C++ bindings for MPI</td>
</tr>
<tr>
<td>libcxxmpich.cnk.so</td>
<td></td>
</tr>
<tr>
<td>libfmpich.cnk.a,</td>
<td>Fortran bindings for MPI</td>
</tr>
<tr>
<td>libfmpich.cnk.so</td>
<td></td>
</tr>
<tr>
<td>libfmpich_.cnk.a</td>
<td>Fortran bindings for MPI with extra underscoring</td>
</tr>
</tbody>
</table>
Resource management APIs

Blue Gene/P resource management applications run on the service node. Table B-4 describes the header files used by resource management applications. They are located in the /bgsys/drivers/ppcfloor/include directory.

Table B-4  Header files for resource management APIs

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allocator_api.h</td>
<td>Available for applications using the Dynamic Partition Allocator APIs</td>
</tr>
<tr>
<td>attach_bg.h</td>
<td>The Blue Gene/P version of attach.h, which is described in the Message</td>
</tr>
<tr>
<td></td>
<td>Passing Interface (MPI) debug specification</td>
</tr>
<tr>
<td>rm_api.h</td>
<td>Available for applications that use Bridge APIs</td>
</tr>
<tr>
<td>rt_api.h</td>
<td>Available for applications that use Real-time Notification APIs</td>
</tr>
<tr>
<td>sayMessage.h</td>
<td>Available for applications that use sayMessage APIs</td>
</tr>
<tr>
<td>sched_api.h</td>
<td>Available for applications that use the mpirun plug-in interface</td>
</tr>
<tr>
<td>submit_api.h</td>
<td>Available for applications that use the submit plug-in interface</td>
</tr>
</tbody>
</table>

Table B-5 describes the 64-bit dynamic libraries available to resource management applications. They are located in the /bgsys/drivers/ppcfloor/lib64 directory.

Table B-5  64-bit dynamic libraries for resource management APIs

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>libbgpallocator.so</td>
<td>Required when using the Dynamic Partition Allocator APIs</td>
</tr>
<tr>
<td>libbgrealtime.so</td>
<td>Required when using the Real-time Notification APIs</td>
</tr>
<tr>
<td>libbgpbridge.so</td>
<td>Required when using the Bridge APIs</td>
</tr>
<tr>
<td>libsaymessage.so</td>
<td>Required when using the sayMessage APIs</td>
</tr>
</tbody>
</table>
Files on architectural features

System calls that provide access to certain hardware or system features can be accessed by applications. In this appendix, we illustrate how to obtain hardware-related information.
Personality of Blue Gene/P

The personality of a Blue Gene/P node is static data given to every compute node and I/O node at boot time by the control system. This data contains information that is specific to the node, with respect to the block that is being booted.

The personality is a set of C language structures that contain such items as the node’s coordinates on the torus network. This kind of information can be useful if the application programmer wants to determine, at run time, where the tasks of the application are running. It can also be used to tune certain aspects of the application at run time, such as determining which set of tasks share the same I/O node and then optimizing the network traffic from the compute nodes to that I/O node.

Example of running personality on Blue Gene/P

Example C-1 illustrates how to invoke and print selected hardware features.

Example: C-1  personal.c architectural features program

```c
/*  --------------------------------------------------------------- */
/* Example: architectural features                                */
/* Written by: Bob Walkup                                           */
/* IBM Watson, Yorktown, NY                                       */
/* September 17, 2007                                              */
/*  --------------------------------------------------------------- */

#include <mpi.h>
#include <stdio.h>
#include <spi/kernel_interface.h>
#include <common/bgp_personality.h>
#include <common/bgp_personality_inlines.h>

int main(int argc, char * argv[]) {
  int taskid, ntasks;
  int memory_size_MBytes;
  _BGP_Personality_t personality;
  int torus_x, torus_y, torus_z;
  int pset_size, pset_rank, node_config;
  int xsize, ysize, zsize, procid;
  char location[128];

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
  MPI_Comm_size(MPI_COMM_WORLD, &ntasks);

  Kernel_GetPersonality(&personality, sizeof(personality));

  if (taskid == 0) {
    memory_size_MBytes = personality.DDR_Config.DDRSizeMB;
    printf("Memory size = %d MBytes\n", memory_size_MBytes);
    node_config = personality.Kernel_Config.ProcessConfig;
  }
```

if (node_config == _BGP_PERS_PROCESSCONFIG_SMP) printf("SMP mode\n");
else if (node_config == _BGP_PERS_PROCESSCONFIG_VNM) printf("Virtual-node mode\n");
else if (node_config == _BGP_PERS_PROCESSCONFIG_2x2) printf("Dual mode\n");
else printf("Unknown mode\n");

printf("number of MPI tasks = %d\n", ntasks);

xsize = personality.Network_Config.Xnodes;
ysize = personality.Network_Config.Ynodes;
zsize = personality.Network_Config.Znodes;

pset_size = personality.Network_Config.PSetSize;
pset_rank = personality.Network_Config.RankInPSet;

printf("number of processors in the pset = %d\n", pset_size);
printf("torus dimensions = <%d,%d,%d>\n", xsize, ysize, zsize);
}

torus_x = personality.Network_Config.Xcoord;
torus_y = personality.Network_Config.Ycoord;
torus_z = personality.Network_Config.Zcoord;

BGP_Personality_getLocationString(&personality, location);

procid = Kernel_PhysicalProcessorID();

/*-----------------------------------------------*/
/* print torus coordinates and the node location */
/*-----------------------------------------------*/
printf("MPI rank %d has torus coords <%d,%d,%d> cpu = %d, location = %s\n", taskid, torus_x, torus_y, torus_z, procid, location);

MPI_Finalize();
return 0;
}

Example C-2 illustrates the makefile that is used to build personality.c. This particular file uses the GNU compiler.

Example: C-2  Makefile to build the personality.c program

BGP_FLOOR   = /bgsys/drivers/ppcfloor
BGP_IDIRS   = -I$(BGP_FLOOR)/arch/include
CC          = /bgsys/drivers/ppcfloor/comm/bin/mpicc
EXE         = personality
OBJ         = personality.o
SRC         = personality.c
FLAGS       =
FLD         =

$(EXE): $(OBJ)
  $(CC) $(FLAGS) -o $(EXE) $(OBJ) $(BGP_LIBS)
$(OBJ): $(SRC)
```
$(CC) $(FLAGS) $(BGP_IDIRS) -c $(SRC)
```

```
clean:
  rm personality.o personality
```

Example C-3 shows a section of the output that is generated after running `personality` using TXYZ mapping. (See Appendix E, “Mapping” on page 303.) Notice that the output has been ordered by MPI rank for readability.

**Example: C-3  Output generated with TXYZ mapping**

```
/bgsys/drivers/ppcfloor/bin/mpirun -partition N04_32_1 -label -env "BG_MAPPING=TXYZ" -mode VN -np 8 -cwd `pwd` -exe personality | tee personality_VN_8_TXYZ.out
```

Memory size = 2048 MBytes
Virtual-node mode
number of MPI tasks = 128
number of processors in the pset = 32
torus dimensions = <4,4,2>

<table>
<thead>
<tr>
<th>MPI rank</th>
<th>has torus coords</th>
<th>cpu</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&lt;0,0,0&gt;</td>
<td>0</td>
<td>R00-M0-N04-J23</td>
</tr>
<tr>
<td>1</td>
<td>&lt;0,0,0&gt;</td>
<td>1</td>
<td>R00-M0-N04-J23</td>
</tr>
<tr>
<td>2</td>
<td>&lt;0,0,0&gt;</td>
<td>2</td>
<td>R00-M0-N04-J23</td>
</tr>
<tr>
<td>3</td>
<td>&lt;0,0,0&gt;</td>
<td>3</td>
<td>R00-M0-N04-J23</td>
</tr>
<tr>
<td>4</td>
<td>&lt;1,0,0&gt;</td>
<td>0</td>
<td>R00-M0-N04-J04</td>
</tr>
<tr>
<td>5</td>
<td>&lt;1,0,0&gt;</td>
<td>1</td>
<td>R00-M0-N04-J04</td>
</tr>
<tr>
<td>6</td>
<td>&lt;1,0,0&gt;</td>
<td>2</td>
<td>R00-M0-N04-J04</td>
</tr>
<tr>
<td>7</td>
<td>&lt;1,0,0&gt;</td>
<td>3</td>
<td>R00-M0-N04-J04</td>
</tr>
<tr>
<td>8</td>
<td>&lt;1,0,0&gt;</td>
<td>0</td>
<td>R00-M0-N04-J04</td>
</tr>
</tbody>
</table>

Example C-4 illustrates running `personality` with XYZT mapping for a comparison. Notice that the output has been ordered by MPI rank for readability.

**Example: C-4  Output generated with XYZT mapping**

```
/bgsys/drivers/ppcfloor/bin/mpirun -partition N04_32_1 -label -env "BG_MAPPING=XYZT" -mode VN -np 8 -cwd `pwd` -exe personality | tee personality_VN_8_XYZT.out
```

Memory size = 2048 MBytes
Virtual-node mode
number of MPI tasks = 128
number of processors in the pset = 32
torus dimensions = <4,4,2>

<table>
<thead>
<tr>
<th>MPI rank</th>
<th>has torus coords</th>
<th>cpu</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&lt;0,0,0&gt;</td>
<td>0</td>
<td>R00-M0-N04-J23</td>
</tr>
<tr>
<td>1</td>
<td>&lt;1,0,0&gt;</td>
<td>0</td>
<td>R00-M0-N04-J04</td>
</tr>
<tr>
<td>2</td>
<td>&lt;2,0,0&gt;</td>
<td>0</td>
<td>R00-M0-N04-J12</td>
</tr>
<tr>
<td>3</td>
<td>&lt;3,0,0&gt;</td>
<td>0</td>
<td>R00-M0-N04-J31</td>
</tr>
<tr>
<td>4</td>
<td>&lt;0,1,0&gt;</td>
<td>0</td>
<td>R00-M0-N04-J22</td>
</tr>
<tr>
<td>5</td>
<td>&lt;1,1,0&gt;</td>
<td>0</td>
<td>R00-M0-N04-J05</td>
</tr>
<tr>
<td>6</td>
<td>&lt;2,1,0&gt;</td>
<td>0</td>
<td>R00-M0-N04-J13</td>
</tr>
<tr>
<td>7</td>
<td>&lt;3,1,0&gt;</td>
<td>0</td>
<td>R00-M0-N04-J30</td>
</tr>
</tbody>
</table>
Porting applications

In this appendix, we summarize Appendix A, “BG/L prior to porting code,” in *Unfolding the IBM eServer Blue Gene Solution*, SG24-6686. Porting applications to massively parallel systems requires special considerations to take full advantage of this specialized architecture. Never underestimate the effort required to port a code to any new hardware. The amount of effort depends on the nature of the way in which the code has been implemented.

Answer the following questions to help you in the decision-making process of porting applications and the level of effort required (answering “yes” to most of the questions is an indication that your code is already enabled for distributed-memory systems and a good candidate for Blue Gene/P):

1. Is the code already running in parallel?
2. Is the application addressing 32-bit?
3. Does the application rely on system calls, for example, `system`?
4. Does the code use the Message Passing Interface (MPI), specifically MPICH? Of the several parallel programming APIs, the only one supported on the Blue Gene/P system that is portable is MPICH. OpenMP is supported only on individual nodes.
5. Is the memory requirement per MPI task less than 1 GB?
6. Is the code computational intensive? That is, is there a small amount of I/O compared to computation?
7. Is the code floating-point intensive? This allows the double floating-point capability of the Blue Gene/P system to be exploited.
8. Does the algorithm allow for distributing the work to a large number of nodes?
9. Have you ensured that the code does not use `flex_lm` licensing? At present, `flex_lm` library support for Linux on System p™ is not available.

If you have answered “yes” to all of these questions, then answer the following questions:

- Has the code been ported to Linux on System p?
- Is the code Open Source Software (OSS)? These type of applications require the use of the GNU standard `configure` and special considerations are required.
- Can the problem size be increased with increased numbers of processors?
- Do you use standard input? If yes, can this be changed to single file input?
Mapping

In this appendix, we summarize and discuss mapping of tasks with respect to the Blue Gene/P system. We define mapping as an assignment of MPI rank onto Blue Gene processors. As with Blue Gene/L, the network topology for Blue Gene/P is a three-dimensional (3D) torus or mesh, with direct links between the nearest neighbors in the +/-x, +/-y, and +/-z directions. When communication involves the nearest neighbors on the torus network, you can obtain a large fraction of the theoretical peak bandwidth. However, when MPI ranks communicate with many hops between the neighbors, the effective bandwidth is reduced by a factor that is equal to the average number of hops that messages take on the torus network. In a number of cases, it is possible to control the placement of MPI ranks so that communication remains local. This can significantly improve scaling for a number of applications, particularly at large processor counts.

The default mapping is to place MPI ranks on the system in XYZT order, where <X,Y,Z> are torus coordinates and T is the processor number within each node (T=0,1,2,3). If the job uses symmetrical multiprocessing (SMP) node mode on the Blue Gene/P system, only one MPI rank is assigned to each node using processor 0. For SMP node mode and the default mapping, we get the following results:

- MPI rank 0 is assigned to <X,Y,Z,T> coordinates <0,0,0,0>.
- MPI rank 1 is assigned to <X,Y,Z,T> coordinates <1,0,0,0>.
- MPI rank 2 is assigned to <X,Y,Z,T> coordinates <2,0,0,0>.

The results continue like this, first incrementing the X coordinate, then the Y coordinate, and then the Z coordinate. In virtual node mode and in dual node mode, the same XYZT order remains the default.

For example, in virtual node mode, the system first places one MPI rank using processor 0 on each of the nodes in XYZ order. The next MPI ranks are assigned to processor 1, again in XYZ order, and so forth. In many cases, it might be better to change this assignment so that the first four MPI ranks use processors 0,1,2,3 on the first node, then the next four ranks use processors 0,1,2,3 on the second node, where the nodes are populated in XYZ order. This ordering is called TXYZ order (first increment T, then X, then Y, and then Z).

The predefined mappings available on Blue Gene/P are the same as those available on Blue Gene/L: XYZT, XZYT, YZXT, ZYXT, TXYZ, TXYZ, TXZY, TYZX, TYXZ, TZYX, TZXY, TZX.
Table E-1 illustrates this type of mapping using the output from the personality program presented in Appendix C, "Files on architectural features" on page 297.

Table E-1  Topology mapping 4x4x2 with TXYZ and XYZT

<table>
<thead>
<tr>
<th>Mapping option</th>
<th>Topology</th>
<th>Coordinates</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>TXYZ</td>
<td>4x4x2</td>
<td>0,0,0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0,0,0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0,0,0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0,0,0</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1,0,0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1,0,0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1,0,0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1,0,0</td>
<td>3</td>
</tr>
<tr>
<td>XYZT</td>
<td>4x4x2</td>
<td>0,0,0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1,0,0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2,0,0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3,0,0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0,1,0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1,1,0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2,1,0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3,1,0</td>
<td>0</td>
</tr>
</tbody>
</table>

The way to specify a mapping depends on the method that is used for job submission. The `mpirun` command for the Blue Gene/P system includes two methods to specify the mapping. You can add `-mapfile TXYZ` to request TXYZ order. Other permutations of XYZT are also permitted. You can also create a map file, and use `-mapfile my.map`, where `my.map` is the name of your map file. Alternatively, you can specify the environment variable `-env BG_MAPPING=TXYZ` to obtain one of the predefined non-default mappings.

The use of a customized map file provides the most flexibility. The syntax for the map file is simple. It must contain one line for each MPI rank in the Blue Gene partition, with four integers on each line separated by spaces, where the four integers specify the <X,Y,Z,T> coordinates for each MPI rank. The first line in the map file assigns MPI rank 0, the second line assigns MPI rank 1, and so forth. It is important to ensure that your map file is consistent, with a unique relationship between MPI rank and <X,Y,Z,T> location.

**General guidance**

For applications that use a 1D, 2D, 3D, or 4D (D for dimensional) logical decomposition scheme, it is often possible to map MPI ranks onto the Blue Gene torus network in a way that preserves locality for nearest-neighbor communication. For example, in a one-dimensional processor topology, where each MPI rank communicates with its rank +/- 1, the default XYZT mapping is sufficient at least for partitions large enough to use torus wrap-around.
Torus wrap-around is enabled for partitions that are one midplane = 8x8x8 512 nodes, or multiples of one midplane. With torus wrap-around, the XYZT order keeps communication local, except for one extra hop at the torus edges. For smaller partitions, such as a 64-node partition with a 4x4x4 mesh topology, it is better to create a map file that assigns ranks that go down the X-axis in the +x direction, and then for the next Y-value, fold the line to return in the -x direction, making a snake-like pattern that winds back and forth, filling out the 4x4x4 mesh. It is worthwhile to note that for a random placement of MPI ranks onto a 3D torus network, the average number of hops is one-quarter of the torus length, in each of the three dimensions. Thus mapping is generally more important for large or elongated torus configurations.

Two-dimensional logical processes topologies are more challenging. In some cases, it is possible to choose the dimensions of the logical 2D process mesh so that one can fold the logical 2D mesh to fit perfectly in the 3D Blue Gene torus network. For example, if you want to use one midplane (8x8x8 nodes) in virtual node mode, a total of 2048 CPUs are available. A 2D process mesh is 32x64 for this problem. The 32 dimension can be lined up along one edge of the torus, say the X-axis, using TX order to fill up processors (0,1,2,3) on each of the eight nodes going down the X-axis, resulting in 32 MPI ranks going down the X-axis.

The simplest good mapping, in this case, is to specify -mapfile TXYZ. This keeps nearest-neighbor communication local on the torus, except for one extra hop at the torus edges. You can do slightly better by taking the 32x64 logical 2D process mesh, aligning one edge along the X-axis with TX order and then folding the 64 dimension back and forth to fill the 3D torus in a seamless manner. It is straightforward to construct small scripts or programs to generate the appropriate map file. Not all 2D process topologies can be neatly folded onto the 3D torus.

For 3D logical process topologies, it is best to choose a decomposition or mapping that fits perfectly onto the 3D torus if possible. For example, if your application uses SMP node mode on one Blue Gene rack (8x8x16 torus); then it is best to choose a 3D decomposition with 8 ranks in the X-direction, 8 ranks in the Y-direction, and 16 ranks in the Z-direction. If the application requires a different decomposition - for example, 16x8x8 - you might be able to use mapping to maintain locality for nearest-neighbor communication. In this case, ZXY order works.

Quantum chromodynamics (QCD) applications often use a 4D process topology. This can fit perfectly onto Blue Gene/P using virtual node mode. For example, with one full rack, there are 4096 CPUs in virtual node mode, with a natural layout of 8x8x16x4 (X,Y,Z,T order). By choosing a decomposition of 8x8x16x4, communication remains entirely local for nearest neighbors in the logical 4D process mesh. In contrast, a more balanced decomposition of 8x8x8x8 results in a significant amount of link sharing, and thus degraded bandwidth in one of the dimensions.

In summary, it is often possible to choose a mapping that keeps communication local on the Blue Gene torus network. This is recommended for cases where a natural mapping can be identified based on the parallel decomposition strategy used by the application. The mapping can be specified using the -mapfile argument for the mpirun command.
Appendix F. htcpartition

The htcpartition utility boots or frees a HTC partition from a front end node or service node. It is similar to mpirun because it communicates with the mpirun daemon on the service node; however, htcpartition cannot run a job. Its return status describes whether or not the request succeeded, zero indicates success and non-zero means failure.

The mpirun scheduler plug-in interface is always called when htcpartition is executed. The plug-in interface provides a method for specifying the partition name if not passed as an argument on the command. If a resource scheduler does not allow mpirun outside its framework, then that policy is also enforced with htcpartition.

When htcpartition begins execution, it looks for an environment variable called MMCS_SERVER_IP. This environment variable if specified must contain the IP address of the service node that the mpirun server daemon is listening on. The --host command-line parameter can be used to override the value of the MMCS_SERVER_IP environment variable. Table F-1 gives a complete list of options for the htcpartition command.

<table>
<thead>
<tr>
<th>Parameter (and syntax)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--help</td>
<td>Provides extended help information.</td>
</tr>
<tr>
<td>--version</td>
<td>Version information.</td>
</tr>
<tr>
<td>--boot</td>
<td>Boot a partition in HTC mode.</td>
</tr>
<tr>
<td>--free</td>
<td>Free a HTC partition.</td>
</tr>
<tr>
<td>--partition</td>
<td>Partition to boot or free. The partition name can be omitted if supplied by the scheduler interface plug-in.</td>
</tr>
<tr>
<td>--mode &lt;SMP/VN/DUAL&gt;</td>
<td>Boot the HTC partition in the requested mode. The default is SMP.</td>
</tr>
<tr>
<td>--userlist &lt;user list</td>
<td>*ALL&gt;</td>
</tr>
</tbody>
</table>
Example F-1 shows how to boot a partition in SMP mode.

Example: F-1 Booting in SMP mode

dd2sys1fen3:~> /bgsys/drivers/ppcfloor/bin/htcpartition --boot --partition R00-M0-N00

By default, *htcpartition* boots a partition so only the owner can run jobs on that partition. You can use the *--userlist* argument to add additional users so they can run jobs on the partition. This is illustrated in Example F-2.

Example: F-2 Adding users

dd2sys1fen3:~> ./htcpartition --boot --mode DUAL --partition R00-M0-N01 --userlist sam,tom,mark,brant

Freeing a partition is shown in Example F-3.

Example: F-3 Free partition

dd2sys1fen3:~> ./htcpartition --free --partition R00-M0-N00

Example F-4 shows the help option.

Example: F-4 htcpartition -h

```bash
~> htcpartition -h
htcpartition --boot|--free [options]
options:
--partition <name> The partition to boot or free
--host <host> mpirun server hostname (default 127.0.0.1, overrides MMCS_SERVER_IP environmental)
--port <port> mpirun server port (default 9874)
--config <file> mpirun config file path
--trace <0-7> tracing level (default 0)
--version display version info
--help this help text

boot options:
--mode <SMP|DUAL|VN> Boot partition in requested mode (default SMP)
--userlist <list> A comma separated list of users allowed to run, *ALL to allow any user jobs on this partition
```
Use of GNU profiling tool on Blue Gene/P

In this appendix we describe the GNU profiling toolchain for Blue Gene/P.

For additional information about the usage of the GNU toolchain profiling tools, visit GNU gprof:
http://sourceware.org/binutils/docs-2.16/gprof/index.html

Speed your code with the GNU profiler:
Profiling with the GNU toolchain

Profiling tools provide information about potential bottlenecks in your program, they help identify functions or sections of the code that could become good candidates to optimize. When using gmon profiling, there are 3 levels of profiling information that can be generated, machine instruction level, procedure level or full level. The choice of options depends on the amount of detail desired and the amount of overhead that is acceptable. Profiling via the GNU compiler set is usually enabled by adding -pg to the gcc compile flags.

Timer tick (machine instruction level) profiling

This level of profiling will provide timer tick profiling information at the machine instruction level. To enable this type of profiling, add the -p option on the link command, but no additional options on the compile commands.

- This level of profiling will add the least amount of performance collection overhead.
- This does not provide call graph information.

Procedure-level profiling with timer tick information

This level of profiling will provide call graph information. To enable this level of profiling, include the -p option on all compile commands and on the link command. In addition to call level profiling, you will get profiling information at the machine instruction level.

- This level of profiling will add some additional overhead during performance data collection.
- When using higher levels of optimization, the entire call flow might not be available due to inlining, code movement, scheduling, and other optimizations performed by the compiler.

Full level of profiling

To enable all available profiling for a program, add the -pg options to all compiles and links. This will provide profiling information that can be used to create call graph information, statement level profiling, basic block profiling, and machine instruction profiling. This level of profiling will introduce the most overhead while collecting performance data. When higher levels of compiler optimization are used, the statement mappings and procedure calls might not appear as expected due to inlining, code movement, scheduling, and other optimizations performed by the compiler.

Additional function in the Blue Gene/P gmon support

The basic gmon support is described in the man pages for the GNU toolchain:
http://gcc.gnu.org/

On Blue Gene/P, in addition to the functionality provided in the standard GNU toolchain, profiling information can be collected on each node. An application may run on multiple nodes, in which case profiling data is collected on each node of execution. To provide data for each node, gmon on Blue Gene/P generates a gmon.out file for each node where the application runs. The files us named gmon.out.x, where x is the rank of the node where profiling information was collected.
Enabling/disabling profiling within your application

To turn profiling on and off within your application, the application must still be compiled with the `-p` and/or `-pg` options as described previously. By inserting the following procedures at various points in the application, the user can enable and disable profile data collection and only collect data for the significant sections of the application:

- `__moncontrol(1)` turns profiling on
- `__moncontrol(0)` turns profiling off

Collecting the gmon data as a set of program counter values

Performance data can be collected in an alternate format, as a set of instruction addresses that were executing at the time of each sampling interval, instead of a summarized histogram. To enable this type of collection, set the environment variable `GMON_SAMPLE_DATA="yes"` before running your program. When data is collected this way, the output files will be named `gmon.sample.x` instead of `gmon.out.x`. In most cases, this file is much smaller than the `gmon.out.x` file, and also allows the user to see the sequence of execution samples instead of the summarized profile. The `gprof` tool in the BG toolchain has been updated to read this type of file.

Enabling profile data for threads in Blue Gene/P

Because enabling profiling on threads impacts the performance of non-profiled runs, the thread profiling function is not included in the base gmon support. To do this type of profiling, an alternate toolchain must be built.

Enhancements to gprof in the Blue Gene/P toolchain

Because Blue Gene/P is a massively parallel system, the GNU toolchain requires additional functionality to collect profiling information about multiple nodes.

Using gprof to read gmon.sample.x files

The version of gprof in the Blue Gene/P toolchain has been modified to recognize and process `gmon.sample.x` files as described previously. When using gprof on a sample file, gprof generates the same type of report as it does for `gmon.out.x` files. If the `-sum` option is added, gprof generates a `gmon.sum` file that is in normal `gmon.out` format from the data in the `gmon.sample.x` file(s). The `-d` option display the program counter values in the order in which they were collected.

Using gprof to merge a large number of gmon.out.x files

The base version of gprof has a limit on the number of `gmon.out.x` files that can be merged in one command invocation. This is due to the Linux limit on input arguments to a command.

The following new option has been added to gprof to allow merging of an unlimited number of `gmon.out.x` files:

```bash
> /bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc-bgp-linux-gprof -sumbg some.pgm
```
This command searches the current directory for all gmon.out files of the form gmon.out.x where x is an integer value, starting with 0 until a file in the sequence cannot be found. The data in these files is summed in the same way as gprof normally does.

As in the previous case, this command searches the current directory for all gmon.sample files of the form gmon.sample.x where x is an integer value, starting with 0 until a file in the sequence cannot be in. A gmon histogram is generated by summing the data found in each individual file, and the output goes to gmon.sum.

> /bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc-bgp-linux-gprof -sumbg=gmon.sample pgm
Statement of completion

IBM considers the Blue Gene/P installation to be complete when the following activities have taken place:

- The Blue Gene/P rack or racks have been physically placed in position.
- The cabling is complete, including power, Ethernet, and torus cables.
- The Blue Gene/P racks can be powered on.
- All hardware is displayed in the Navigator and is available.
References

1. TOP500 Supercomputer sites:
   
   http://www.top500.org/

2. The MPI Forum. The MPI message-passing interface standard. May 1995:
   

3. OpenMP application programming interface (API):
   
   http://www.openmp.org

4. IBM XL family of compilers:
   
   – XL C/C++
     
   
   – XL Fortran
     

5. GCC, the GNU Compiler Collection:
   
   http://gcc.gnu.org/

6. IBM System Blue Gene Solution: Configuring and Maintaining Your Environment, SG24-7352.

7. GPFS Multicluster with the IBM System Blue Gene Solution and eHPS Clusters, REDP-4168.

8. Engineering and Scientific Subroutine Library (ESSL):
   
   http://www-03.ibm.com/systems/p/software/essl.html


10. See note 3.

11. See note 4.

12. See note 5.


15. See note 8.

   

17. See note 2.

18. See note 3.

19. See note 5.

20. See note 8.

21. Ganier, C J. “What is Direct Memory Access (DMA)?”
   
   http://cnx.org/content/ml1867/latest/

22. See note 2.
23. See note 3.
27. See note 3.
29. Ibid.
30. Ibid.
31. See note 3.
32. Flynn’s taxonomy in Wikipedia:
   http://en.wikipedia.org/wiki/Flynn%27s_Taxonomy
33. Rennie, Gabriele. “Keeping an Eye on the Prize.” *Science and Technology Review*, July/August 2006:
   http://www.llnl.gov/str/JulAug06/pdfs/07_06.3.pdf
   http://www.llnl.gov/str/JanFeb06/Schwegler.html
35. SC06 Supercomputing Web site, press release from 16 November 2006:
36. *Unfolding the IBM eServer Blue Gene Solution*, SG24-6686
   http://prola.aps.org/abstract/PRL/v55/i22/p2471_1
39. See note 33.
41. Ibid.
   http://portal.acm.org/citation.cfm?id=952903.952912&dl=GUIDE&dl=ACM
46. See note 42.
47. See note 43.
48. Ibid.
49. Ibid.
50. Ibid.
51. Ibid.
52. See note 44.
53. LAMMPS Molecular Dynamics Simulator:
   http://lammps.sandia.gov/
54. See note 45.
56. Brünger, A. I. “X-PLOR, Version 3.1, A System for X-ray Crystallography and NMR.” 1992: The Howard Hughes Medical Institute and Department of Molecular Biophysics and Biochemistry, Yale University. 405.
63. Ibid.
64. Ibid.
65. Ibid.
66. Ibid.
69.Ibid.


71. See note 38.

72. (a) CPMD V3.9, Copyright IBM Corp. 1990-2003, Copyright MPI fur Festkörperforschung, Stuttgart, 1997-2001. (b) See also:
   
   http://www.cpmd.org

73. Marx, D. and Hutter, J. *Ab-initio molecular dynamics: Theory and implementation* in *Modern Methods and Algorithms of Quantum Chemistry*. J. Grotendorst (ed.), NIC Series, 1, FZ Julich, Germany, 2000. See also the following URL and references therein:
   

   
   http://prola.aps.org/abstract/PRB/v41/i11/p7892_1

75. See note 72.


77. Ibid.

78. Ibid.


81. Ibid.


84. Heyman, J. “Recommendations for Porting Open Source Software (OSS) to Blue Gene/P,” white paper WP101152:
   
   http://www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP101152
Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks

For information about ordering these publications, see “How to get IBM Redbooks” on page 322. Note that some of the documents referenced here might be available in softcopy only.

- IBM System Blue Gene Solution: Blue Gene/P Safety Considerations, REDP-4257
- Blue Gene/L: Hardware Overview and Planning, SG24-6796
- Blue Gene/L: Performance Analysis Tools, SG24-7278
- Evolution of the IBM System Blue Gene Solution, REDP-4247
- GPFS Multicluster with the IBM System Blue Gene Solution and eHPS Clusters, REDP-4168
- IBM System Blue Gene Solution: Application Development, SG24-7179
- IBM System Blue Gene Solution: Configuring and Maintaining Your Environment, SG24-7352
- IBM System Blue Gene Solution: Hardware Installation and Serviceability, SG24-6743
- IBM System Blue Gene Solution Problem Determination Guide, SG24-7211
- IBM System Blue Gene Solution: System Administration, SG24-7178
- Unfolding the IBM eServer Blue Gene Solution, SG24-6686

Other publications

These publications are also relevant as further information sources:

- Brünger, A. I. “X-PLOR, Version 3.1, A System for X-ray Crystallography and NMR.” 1992: The Howard Hughes Medical Institute and Department of Molecular Biophysics and Biochemistry, Yale University. 405.


http://prola.aps.org/abstract/PRB/v41/i11/p7892_1


### Online resources

These Web sites are also relevant as further information sources:

- Compiler-related topics:
  - XL C/C++
    
  - XL C/C++ library
    
  - XL Fortran Advanced Edition for Blue Gene
    
  - XL Fortran library
    
Debugger-related topics:
- GDB: The GNU Project Debugger
  http://www.gnu.org/software/gdb/gdb.html
- GDB documentation:
  http://www.gnu.org/software/gdb/documentation/

- Engineering and Scientific Subroutine Library (ESSL) and Parallel ESSL
  http://www-03.ibm.com/systems/p/software/essl.html

- GCC, the GNU Compiler Collection
  http://gcc.gnu.org/

- Intel MPI Benchmarks is formerly known as “Pallas MPI Benchmarks.”

- Mathematical Acceleration Subsystem

- Message Passing Interface Forum
  http://www.mpi-forum.org/

- MPI Performance Topics
  http://www.llnl.gov/computing/tutorials/mpi_performance/

- The OpenMP API Specification:
  http://www.openmp.org

- Danier, CJ, “What is Direct Memory Access (DMA)?”
  http://cnx.org/content/m11867/latest/

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads
ibm.com/support

IBM Global Services
ibm.com/services
Index

Numerics
10 Gb Ethernet network 11
32-bit static link files 295
3D torus network 10

A
Ab Initio method 269
adaptive routing 70
address space 22
allocate block 138
AMBER 271
ANSI-C 61
APIs
Control System 195–235
Dynamic Partition Allocator 257–263
Real-time Notification 237–256
applications
checkpoint and restart support for 159–166
developing 95–136
porting 301
running and debugging 137–157
Argonne National Labs 18
arithmetic functions 123
asynchronous API 199
asynchronous file I/O 22

B
base partition 231
definitions 232
BG_CHKPT_ENABLED 166
BG_SHARED_MEMPOOLSIZE 42
bgpmaster daemon 26
binary functions 125
binutils 97
block 138
blrts_xlc 99
blrts_xlc++ 99
blrts_xlf 99
Blue Gene specifications 12
Blue Gene XL compilers 95
Blue Gene/L PowerPC 440d processor 95
Blue Gene/P
environment 6
hardware 3
memory considerations 18
programs 11
software 16
system architecture 4
threading support 18
Blue Matter 272
Bridge API 196
asynchronous API 199
examples 232
first and next calls 197
invalid pointers 197
library files 238, 258
memory allocation and deallocation 197
requirements 196, 238
return codes 199, 247
sample makefile 238
bss 19
built-in floating-point functions 115

C
cache
L1 46–47, 74
L2 46, 48
L3 46, 48
Car-Parrinello Molecular Dynamics (CPMD) 270
Cartesian communicator functions 75
checkpoint and restart
API 163
BG_CHKPT_ENABLED 166
BGLatCheckpoint 164
BGLatContinue 164
BGLatRestart 164
BGLCheckpoint 163
BGLCheckpointExcludeRegion 164
BGLCheckpointInit 163
BGLCheckpointRestart 164
directory and file naming conventions 165
I/O considerations 161
restarting application 165
signal considerations 161
technical overview 160
checkpoint library 160
checkpoint write complete flag 165
chip 4
CIOD (control and I/O daemon) 33, 35
threading 37
Classical Molecular Mechanics/Molecular Dynamics
(MM/MD) 269
collective 11
collective MPI 82, 282
collective network 69
common node services 34
Communication Coprocessor mode 17, 39–40, 49
communications performance 80
compilers
GNU 23
IBM XL 24
complex type manipulation functions 119
calculate card 4
calculate node 5–6, 9
calculate node kernel 5, 17, 22, 32, 54
socket services 25
control and I/O daemon (CIOD) 33, 35, 37
control network  6, 11
control system   25
bgpmaster   26
Bridge API   25
ciodb   25
MMCS   25
mpirun   25
Control System APIs   195–235
jm_attach_job   208
jm_begin_job   208
jm_cancel_job   208
jm_debug_job   209
jm_load_job   210
jm_signal_job   211
jm_start_job   211
job state flags   209
message types   230
messaging API   230
partition state flags   204
pm_create_partition   202
pm_destroy_partition   203
requirements   196, 258
rm_add_job   207
rm_add_part_user   202, 240, 248
rm_add_partition   201, 240, 248
rm_assign_job   202
rm_free_BGL   229
rm_free_BP   229
rm_free_job   229
rm_free_job_list   229
rm_free_nodecard   229
rm_free_nodecard_list   229
rm_free_partition   229
rm_free_partition_list   229
rm_free_switch   229
rm_get_BGL   200
rm_get_data   198, 200
rm_get_job   209
rm_get_jobs   209
rm_get_partitions   203, 240, 248–249
rm_get_partitions_info   204, 241, 249
rm_get_serial   201
rm_new_BP   228
rm_new_job   228
rm_new_nodecard   228–229
rm_new_partition   229
rm_new_switch   229
rm_query_job   210
rm_release_partition   205
rm_remove_job   210
rm_remove_part_user   206, 240, 248
rm_remove_partition   205
rm_set_data   198, 201
rm_set_part_owner   206
rm_set_serial   201
copy-primary operations   116
copy-secondary operations   117
core files   154
Core Processor tool   146
cores, computation of   5

CPMD (Car-Parrinello Molecular Dynamics)   270
critical pragma   90
cross operations   116
cross-copy operations   118

D
data   19
DB_PROPERTY   238
DCMF_EAGER   70
DDR (double data RAM)   48
debug client, debug server   141
debugging applications   137, 140–157
deterministic routing   69
DLL (dynamically loaded libraries)   101
DOCK6   275
double data RAM (DDR)   48
Double Hummer FPU   99
double-precision square matrix multiply example   134
DUAL mode (2X2)   42
dual node mode   42, 51
dynamic linking   23
Dynamic Partition Allocator APIs   257–263
dynamically loaded libraries   100

E
eager protocol   70, 73
electronic correlation   276
electronic structure method   269
Engineering and Scientific Subroutine Library (ESSL)   103
entities   25
ESSL (Engineering and Scientific Subroutine Library)   103
Ewald sums   270
execution process modes   39–43
extended basic blocks   110

F
fault recovery   160
  see also checkpoint and restart
file I/O   22
flood of messages   72
freepartition   168
front end node   6, 13
functional Ethernet (10 Gigabit)   11
functional network   6

G
GDB (GNU Project debugger)   141
gdbserver   141
General Parallel File System (GPFS)   13
gid   54
global collective network   11
global interrupt network   11
GNU
  3.2 C   23
  C++   23
  Fortran77   23
Index

GDB 141
GNU Project debugger (GDB) 141
GPFS (General Parallel File System) 13

H
hardware 3–14
  information related to, obtaining 297–300
  naming conventions 287–292
header files 293–295
heap 19
High Throughput Computing (HTC) 187
High-Throughput Computing (HTC) paradigm 187–192
HMMER 277
host system 13
host system software 14
HTC (High Throughput Computing) 187
htcpartition 188, 307–308

I
I/O node 5–6, 10, 24, 34
daemons 25
  file system services 24
  kernel boot 24
  software 24
I/O, See input/output (I/O)
IBM XL compiler 24, 95
inline functions 111
inlining 111
input/output (I/O) 22
  file 22
  sockets calls 23
installation 313
Intel MPI Benchmarks 80

K
kernel functionality 31–37

L
LAMMPS 273
libbgreatime.so 238
libraries 293–295
ligand atoms 275
load and store functions 120
LoadLeveler 139

M
mapping 303–305
MASS (Mathematical Acceleration Subsystem) 103
Mathematical Acceleration Subsystem (MASS) 103
mcServer daemon 26
memory 18, 45–51
  address space 22
  addressing 19
  considerations 9
  distributed 46, 68
  leaks 22
  management 22
  model 22
  virtual 46
memory management 47
message layer 41
Message Passing Interface (MPI) 68
  bandwidth 80
  collective 82, 282
  eager protocol 70
  latency 80
  point-to-point 81
  rendezvous protocol 70
  short protocol 70
  too much memory 71
microprocessor 8
midplane 7
Midplane Management Control System (MMCS) 25, 27
Midplane Management Control System APIs 257
MM/MD (Classical Molecular Mechanics/Molecular Dynamics) 269
mmap 42
MMCS (Midplane Management Control System) 25, 27
MMCS console 138
MMCS daemon 26
MPI (Message Passing Interface) 68
  bandwidth 80
  collective 82, 282
  communications 75
  eager protocol 70
  latency 80
  point-to-point 81
  rendezvous protocol 70
  short protocol 70
  too much memory 71
MPI_Barrier 75
MPI_COMM_WORLD 76
MPI_irecv 70
MPI_isend 70
MPI_SUCCESS 76
MPI_Test 72
MPI_Wait 73
MPI-2 18
mpiBLAST-PIO 278
MPICH2 18, 68
mpiexec 169
mpirun 25, 139, 167–186
  challenge protocol 171
    -env 181
    freepartition 168
    get_parameters() 185
    mpirun_done() 186
    multiple program, multiple data (MPMD) 169
    -psets_per_bp 180
    SIGINT 184
mpirun.cfg 171
mpix.h file 75
multiple program, multiple data (MPMD) 169
multiply-add functions 127

N
NAMD 274
natural alignment  105
network  10
   10 Gb Ethernet  11
   3D torus  10
   collective  11, 69
   control  11
   functional Ethernet  11
   global collective  11
   global interrupt  11
   torus  10, 69
node
   front end  13
   I/O  24
   service  13
   storage  13

O
OpenMP  86, 99, 142

P
parallel operations  116
particle mesh Ewald (PME) method  270
performance  47–48, 82
   engineering and scientific applications  267–283
   personality  33
PingPong  280
PME (particle mesh Ewald) method  270
pmemd  271
PMI_Cart_comm_create()  75
PMI_Pset_diff_comm_create()  76
PMI_Pset_same_comm_create()  76
pointer aliasing  112
pointers, uninitialized  22
point-to-point MPI  81
pool (HTC)  190
PowerPC 440d Double Hummer dual FPU  115
PowerPC 440d processor  95
PowerPC 450 microprocessor  8
processor set (pset)  75
pset (processor set)  75
Python  103

Q
q64  99
qaltivec  99
qarch  98
qcachem  98
qffttrap  99
qinline  111
qipa  111
QM/MM (Quantum Mechanical/Molecular Mechanical)  270
qmshrobj  99
qnoautoconfig  98
qpic  99
qtune  98
Quantum Mechanical/Molecular Mechanical (QM/MM)  270

R
rack  4
raw state  241
real-time application code  249
Real-time Notification APIs  237–256
   blocking or non-blocking  239
   libbgrealtime.so  238
   rt_api.h  238
   RT_CALLBACK_CONTINUE  241
   RT_CALLBACK_QUIT  241
   RT_CALLBACK_VERSION_0  241
   rt_callbacks_t  241
   RT_CONNECTION_ERROR  239, 249
   RT_DB_PROPERTY_ERROR  248
   rt_get_msgs  239
   RT_HANDLE_CLOSE  249
   rt_handle_t  239
   rt_init  238
   RT_INVALID_INPUT_ERROR  248–249
   rt_set_blocking  239
   rt_set_filter  240
   rt_set_nonblocking  239
   RT_STATUS_OK  241
   RT_WOULD_BLOCK  249
real-time server  239
Redbooks Web site  322
   Contact us  xiii
reduction clause  90
rendezvous protocol  70, 73
rm_modify_partition  204
running applications  138

S
security  139, 168
segfaults  49
service actions  26
service node  6, 13
shared libraries  100
shared memory  42
shm_open  42
SIMD (single-instruction, multiple-data)  104
SIMD computation  116
single-instruction, multiple-data (SIMD)  48, 104
single-instruction, multiple-data (SIMD), See SIMD (single-instruction, multiple-data)
size command  19
small partition
   allocation  231, 234, 249
   query  234
SMP mode (1X4)  40
SMP node mode  49
socket support  59
sockets calls  23
software  15–27
SPI (System Programming Interface)  59
stack  19
standard input  22
static libraries  100
stdin  22
storage node 13
structure alignment 110
submit APIs 192
submit command 188
SUBMIT_CWD 189
SUBMIT_POOL 189
SUBMIT_PORT 189
Symmetrical Multi-Processing (SMP) node mode 40, 49
system architecture 4
system calls 53–63
  other 59
  return codes 54
  unsupported 62
System Programming Interface (SPI) 59

T
  threading support 18
  TLB (translation look-aside buffer) 49
torus communications 75
torus network 69
torus wrap-around 305
translation look-aside buffer (TLB) 49
TXYZ order 303

U
  uid 54
  unary functions 123
  uninitialized pointers 22

V
  vectorizable basic blocks 110
  virtual FIFO 41
  virtual memory 46
  virtual node mode 17, 39–40, 50
  virtual paging 22
  VN mode (4X1) 40

X
XL
  __alignx function 114
  __attribute__((always_inline)) extension 111
  __cimag 120
  __cimagf 120
  __cimagl 120
  __cmplx 119
  __cmplx_f 119
  __cmplxl 119
  __creal 120
  __crealf 119
  __creall 120
  __fpabs 125
  __fpadd 125
  __fpctiw 123
  __fpctiwz 124
  __fpmsub 128
  __fpmsub 128
  __fpnabs 125
  __fpneg 125
  __fpnmadd 127
  __fpnmsub 128
  __fpre 124
  __fprsp 124
  __fprsqpte 124
  __fpesel 133
  __fpesub 123
  __fxpmadd 129
  __fxpmsub 130
  __fxpnmadd 130
  __fxpnmmsub 130
  __fxcpnmpma 131
  __fxcpnmsma 131
  __fxcpnmsma 131
  __fxcsnmpma 132
  __fxcsnmsma 132
  __fxcsmadd 129
  __fxcsmsub 130
  __fxcsnsub 130
  __fcsnmsub 130
  __fxcmnma 131
  __fxcmnms 132
  __fxcmnmpma 132
  __fxcmnmsma 132
  __fxcmadd 128
  __fxmr 123
  __fxmsub 129
  __fxmul 126
  __fxmadd 128
  __fxnmsub 129
  __fxmul 127
  __fxsmul 127
  __lfpd 121
  __lfts 120
  __lfxd 121
  __lfxs 121
  __stfpmadd 122
  __stfpmiw 123
  __stfms 122
  __stfxd 122
  __stfs 122
#pragma disjoint directive 112
ALIGNNX 114
arithmetic functions 123
  basic blocks 110
  batching computations 113
  binary functions 125
  built-in floating-point functions 115
  built-in functions usage 133
CIMAG 120
CIMAGF 120
CIMAGL 120
CMPLX 119
CMPLXF 119
compiling and linking 97
complex type manipulation functions 119
  copy-primary operations 117
  copy-secondary operations 117
CREAL 120
CREALF 119
Blue Gene/P Application Development

Blue Gene/P Application Development
IBM System Blue Gene Solution: Blue Gene/P Application Development

Understand the Blue Gene/P programming environment

Learn how to run and debug MPI programs

Learn about Bridge and Real-time APIs

This IBM Redbooks publication is one in a series of IBM books written specifically for the IBM System Blue Gene/P Solution. The Blue Gene/P system is the second generation of a massively parallel supercomputer from IBM in the IBM System Blue Gene Solution series. This book provides an overview of the application development environment for the Blue Gene/P system. It is intended to help programmers understand the requirements to develop applications on this high-performance massively parallel supercomputer.

In this book, we explain instances where the Blue Gene/P system is unique in its programming environment. We also attempt to look at the differences between the IBM System Blue Gene/L Solution and the Blue Gene/P Solution. This book does not delve into great depth about the technologies that are commonly used in the supercomputing industry, such as Message Passing Interface (MPI) and Open Multi-Processing (OpenMP), nor do we try to teach parallel programming. References are provided in those instances for you to find more information if desired.

Prior to reading this book, you must have a strong background in high-performance computing (HPC) programming. The high-level programming languages that are used throughout this book are C/C++ and Fortran95. Previous experience using the Blue Gene/L system can help you understand better some concepts in this book that we do not extensively discuss. However, several IBM Redbooks publications about the Blue Gene/L system are available for you to obtain general information about the Blue Gene/L system.

For more information: ibm.com/redbooks