Implementing a Scalable Parallel Reduction in Unified Parallel C

Introduction

A reduction is the process of combining elements of a vector (or array) to yield a single aggregate element. It is commonly used in scientific computations. For instance the inner product of two n-dimensional vectors $x$, $y$ is given by:

$$x^T y = \sum_{i=1}^{n} x_i y_i$$

This computation requires $n$ multiplications and $n-1$ additions. The $n$ multiplications are independent from each other, therefore could be executed concurrently. Once the additive terms have been computed they can be summed together to yield the final result.

Given the importance of reduction operations in scientific algorithms, many parallel languages provide support for user defined reductions. For example, OpenMP provides a user reduction clause to be used in the OMP parallel construct. In the following example the reduction clause indicates that “sum” is a reduction variable:

```c
int A[N];
int sum=0;
#pragma omp parallel for reduction ( +: sum )
for ( i=1;i<N;i++) {
    sum = sum + A[i];
}
```

In the code snippet above each thread performs a portion of the additions that make up the final sum. At the end of the parallel loop, the partial sums are combined into the final result.

Reduction Implementation in Unified Parallel C

In this article we will explore different implementations of a global sum reduction in Unified Parallel C, in an attempt to find an efficient and scalable implementation.

Unified Parallel C (UPC) is an extension to the C programming language that allows users to express parallelism in their code. The UPC language subscribes to the Partition Global Address Space programming model. Partitioned Global Address Space (PGAS) languages such as UPC are increasingly seen as a convenient way to enhance programmer productivity for High-Performance Computing (HPC) applications on large-scale machines.

A UPC program is executed by a fixed number of threads (THREADS) which are created at program startup. In general a program statement might be executed concurrently by all threads. To facilitate the distribution and coordination of work among threads UPC provides a rich set of language features. A tutorial on Unified Parallel C is available at http://domino.research.ibm.com/comm/research_projects.nsf/pages/xlupc.confs.html/$FILE/PACT08Tutorial.pdf.
Let's consider how a naive sum reduction could be written in Unified Parallel C

```c
#include <upc.h>
#define N 100000000
shared int A[N];
shared int sum=0;
int main() {
    int i;
    upc_forall (i=0;i<N;i++;&A[i]) { A[i] = 1; }
    upc_barrier;
    upc_forall (i=0;i<N;i++;&A[i] ) { sum = sum + A[i]; }
    upc_barrier;
    if( MYTHREAD == 0 && sum != N )
    {
        printf ("Thread:%d,result=%d,expect=%d\n",MYTHREAD,sum,N);
        return 1;
    }
    return 0;
}
```

At lines 5 - 6 we have declared a shared array "A" and a shared variable "sum". At line 10 we initialize all elements of A to 1. At line 13 we have attempted to perform the reduction by accumulating the sum of A's elements values in shared variable "sum". Is this program correct?

A possible program output is:

```
Thread:0,result=27176662,expect=100000000
```

The result is obvious wrong, but what is the problem? The keen reader might point out that the program as written contains a race condition. Multiple threads can write into shared variable "sum" concurrently, possibly overwriting a partial value previously stored.

In order to eliminate the race condition we could protect writes into variable "sum" using a critical section. In UPC this is accomplished by using a "lock" variable as follow:

```c
upc_forall ( i=0;i<N;i++;&A[i] ) {
    upc_lock ( lock );
    sum = sum + A[i];
    upc_unlock ( lock );
}
```

The modified version of the program will output the correct result. However what are the implications of this "solution"? The use of the lock effectively serializes the upc_forall loop iterations, preventing any performance gain from parallel execution. To confirm this theory we have measured how long it takes for the upc_forall loop above to compute the sum of the array elements. Our experiments were conducted on a POWER 5 system running AIX5.3 using up to 32 threads (Figure 1).
From the results illustrated in Figure 1 we can infer that the time it takes to execute the upc_forall loop does not improve considerably when the number of threads used to execute the program increases. This is what we expected because the use of the lock in the loop prevents concurrent execution of loop iterations.

To get better scalability (increased program performance as the number of threads increases), it is critical to remove the lock in the upc_forall loop. This can be done by accumulating the partial sum computed by each thread into a thread-local variable. A thread-local variable is allocated in the private memory space of each thread, thus there are THREADS “instances” of the variable. Each instance of the thread-local variable can be used to accumulate the sum of the array elements having affinity to each thread:

```upc
upc_forall ( i=0;i<N;i++;&A[i] ) {
   partialsum += A[i];
}
upc_barrier;
```

```upc
upc_lock (lock);
sum = sum + partialsum;
upc_unlock(lock);
```

In the code fragment shown above the thread-local variable “partialsum” is used to store the sum of the array elements having affinity to the executing thread (MYTHREAD). For example THREAD 0 will add array elements A[0], A[THREADS], A[2*THREADS], etc… in its instance of “partialsum”. In order to compute the final result it is necessary to add the “partialsum” contributions from each thread. To avoid a race condition (write-after-write hazard on variable “sum”), we use the UPC lock functions to serialize the accesses on “sum”.

Figure 1: Effect of using a lock inside a upc_forall loop.
The performance result illustrated by Figure 2 demonstrates that the program is now “scalable”. That is the time taken to compute the reduction diminishes as the number of threads used to execute the program increases. The reason for this improvement is simple: the lock is now acquired THREADS times in total instead of being acquired by each thread in every loop iteration.

**Conclusion**

In this article, we illustrate the concept of a reduction operation and explained how to implement a parallel reduction in Unified Parallel C. We have shown how the UPC lock primitives can be used to guarantee program correctness. We have then compared the performance of two distinct correct reduction implementations, one using a lock inside a upc forall loop; the other using thread-local variables to accumulate partial results on each thread. The performance measurements obtained clearly indicates that locks should be used judiciously (if at all) inside loops.
May 2010

References in this document to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM program product in this publication is not intended to state or imply that only IBM’s program product may be used. Any functionally equivalent program may be used instead.

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. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. 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 "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml

© Copyright International Business Machines Corporation 2010. US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.