ハイブリッド・システム向け並列プログラミング・フレームワーク

土居 意弘　村瀬 正名　前田 久美子　小松 秀昭

A Parallel Programming Framework for Hybrid Systems

Munehiro Doi, Masana Murase, Kumiko Maeda and Hideaki Komatsu

ハイブリッド・システムが広がり始めているが，アルゴリズムの並列化，アーキテクチャーごとの最適化やデータ通信には高度な知識とノウハウが必要であり，複雑なプログラミングが課題となっている．本論文では，ハイブリッド・システム向けの新しいプログラミング・フレームワークを提案する．本フレームワークでは，熟練開発者がアーキテクチャーに最適化されたプログラム部品を開発し，一般開発者はドラッグ・アンド・ドロップ操作を用いて複雑なアプリケーションを記述する．演算リソースの割り当てと通信コードの生成は，実際のシステムに合わせてコンパイラーが自動的に実現する．姫野ベンチマークを例として，並列度やシステム・リソース，部品の制約に応じた構成の実行形式が自動生成されることを示す．

Hybrid Computing is being widely used in a variety of areas, from high performance computing to embedded computing, with the aim of achieving higher performance and power efficiency. However, one important problem when using hybrid systems is how to program hybrid applications. In this paper, we propose a novel programming framework, with a focus on both higher performance and productivity by exploiting a two-layered programming model, comprising component development by expert programmers and data-flow programming using a visual editor by application developers. The compiler for this programming framework automatically carries out resource scheduling and the generation of networking code. We demonstrate by using HimenoBMT that our compiler generates optimized executables based on a given hardware configuration and application properties.

Key Words & Phrases：ハイブリッド・システム，ストリーム・アプリケーション，ビジュアル・プログラミング，スループット・コンピューティング
hybrid systems, stream application, visual programming, throughput computing

1. はじめに

近年，現在主流となっているマルチコア・システムのさらなる性能向上のため，特定の演算で高い性能を発揮するアクセラレーターを組み合わせたハイブリッド（またはヘテロジニアス）・システムと呼ばれる形態が，注目を集めている．例えば，世界中のスーパーコンピューターをランキングしたTop500では，2010年6月時点のベスト10のうち3システムがハイブリッド・システムである［1］［2］．特に，Graphics Processing Unit（GPU）を汎用演算に転用したもの（General Purpose GPU）をアクセラレーターとして利用するものが増えつつある．日本国内でも，NVIDIA Tesla［3］をアクセラレーターに利用するPCクラスターとして，東京工業大学のTSUBAME 1.2［4］や理化学研究所（理研）のRIKEN Integrated Cluster of Clusters（RICC）［5］などが稼働している．またIBM zEnterprise™［6］のようなビジネス向けコンピューターにもハイブリッド・システムの応用例が現れ始めており，研究から実用の段階に移りつつある．

本論文では，異なるマイクロアーキテクチャーを有する複数の演算リソースがネットワークで接続されたシステムを，ハイブリッド・システムと定義する（図1）．ネットワークにはInfiniBand，Ethernet，PCI Express（PCIe），メモリー，内部バスなどが含まれる．

ハイ・パフォーマンス・コンピューティング（HPC: High Performance Computing）・アプリケーションにおけるマルチコア・システム向けのアプリケーション開発は，Message Passing Interface（MPI）を用いるのが現在主流となっているが，熟練者であってもプログラミングの

提出日:2010年5月10日　再提出日:2010年9月7日

難易度は高い［7］といわれているが，ハイブリッド・システム向けプログラミングは，アーキテクチャーごとに最適な手法が異なっており，さらに難しい．こうした背景から，筆者らは高生産性，高性能を実現するハイブリッド・システム向けプログラミング・フレームワークを開発している．本論文では，ハイブリッド・システム向け並列アプリケーション開発における課題を以下の3つに整理した．

課題 1）ハイブリッド化による生産性の低下

一般にシステムのアーキテクチャーが異なると，最適なプログラムの書き方は異なる．場合によってはプログラミング言語やツールも異なる．ハイブリッド・システムではプログラマーはそれぞれの演算リソースに応じてプログラムを書き分けなければならないため，膨大な手間と高度なスキルが必要になる．この生産性の低下を抑制することが第 1 の課題である．

課題 2）演算リソースの最適配置

アプリケーションを構成する各処理に，それぞれ最も適した演算リソースを割り当てることが第 2 の課題である．与えられた演算リソースの範囲で性能を最大化するように，スケジューリング問題を解かなくてはならない．

課題 3）大規模並列通信

通信プログラムのコーディング作業が第 3 の課題である．並列通信プログラムは，開発に手間がかかるだけではなく，再現や改善が困難なタイミングに依存したバグを生みやすい．加えて，プログラマーは，プロセス間通信，Ethernet，InfiniBand など，さまざまな通信手段を使い分ける必要がある．

本論文では，上記の課題を解決するため，2- レベル・プログラミング・モデルを提案し，演算リソースの最適割り当て，および通信プログラムの自動生成をコンパイラーで実現する．本論文が提案する 2- レベル・プログラミング・モデルでは，熟練開発者が再利用可能なアルゴリズムを単一のマイクロアーキテクチャーに最適化された部品としてプログラミングする．一般開発者はドラッグ・アンド・ドロップ操作で部品をつなぎ合わせることで，より複雑なアプリケーションを記述する（課題 1）．筆者らが開発中のコンパイラーは，対象となるハイブリッド・システムに合わせて，アプリケーションを構成する各部品の演算リソースへの最適割り当て，割り当て先に応じた部品間通信の効率的なコードの自動生成を実現する（課題 2，3）．本フレームワークにより，従来は熟練開発者に限られていたハイブリッド・アプリケーションの構築を，一般開発者がドラッグ・アンド・ドロップ操作と数個のパラメーターの操作だけで実現できる．また,部品ライブラリー開発者もアプリケーション開発者も，実際のハイブリッド・システムのハードウェア構成に特化したプログラミングをする必要がなくなる．

事例として，本フレームワークを理化学研究所様の RICC に適用したプロジェクトを取り上げ，取り組みを紹介する．

本論文の構成は以下の通りである．第 2 章で従来技術について述べる．第 3 章で本フレームワークの概要とプログラミング手順について解説し，第 4 章で本フレームワークを流体シミュレーションに適用した実験結果を報告する．第 5 章で本論文の総括と今後の課題を述べる．

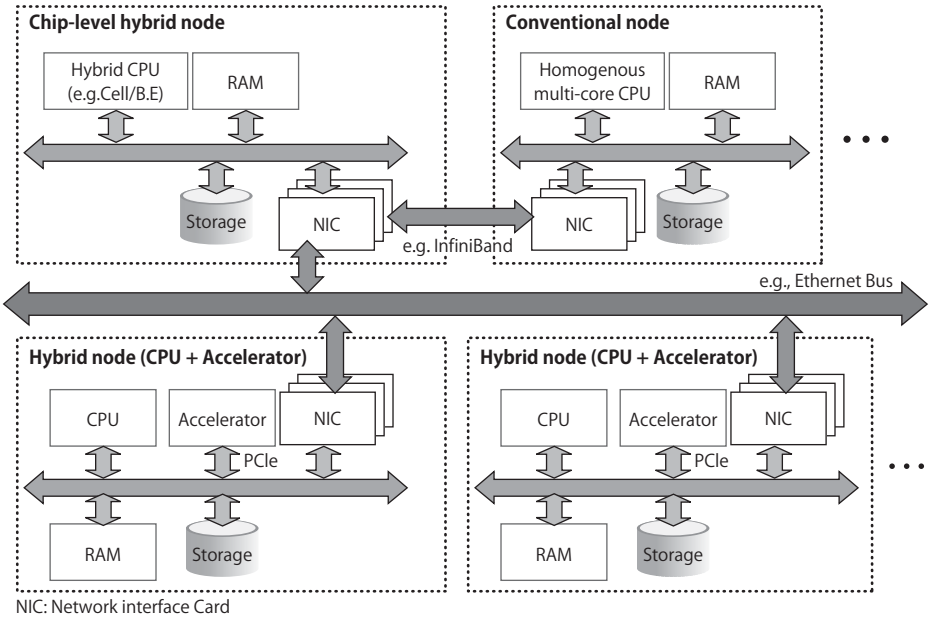

図 1. ハイブリッド・システム

2. 並列処理の従来技術

Message Passing Interface（MPI）［8］は，HPC の分野で広く使われている，Single Program Multiple Data（SPMD）型のプログラミング・モデルの並列化ライブラリーである．ハイブリッド・システムを考慮していないこと，明示的なノード間通信が必要な点などに課題がある．

OpenMP［9］ と Cell Super Scalar（CellSs）［10］では，ソースコードにディレク

ティブを挿入することで，コンパイラーによる自動並列化と通信コードの生成を行う．MPI と比べてアプリケーションの並列化は容易であるが，対称マルチプロセッシング（Symmetric Multiprocessing; SMP）システムのみを対象としており，ハイブリッド・システムに広く使えるものではない．

X10［11］，AutoPipe［12］は並列コンピューティング向けのプログラミング言語である．これらは並列化アプリケーションの開発における高性能と高い生産性の両立を狙いとしているが，ハイブリッド・システムの演算リソースの最適配置については解決していない．

ストリーム・アプリケーション向けのプログラミング言語として，StreamIt［13］，IBM Stream Processing Application Declarative Engine（SPADE）［14］，Sequoia［15］，Brook［16］，Compute Unified Device Architecture（CUDA）［17］などがあるが，特定のハードウェア・アーキテクチャー専用のものが多く，演算リソースの最適配置，ハイブリッド・システムでの生産性低下の課題がある．

本論文の貢献は，以下の 3 点である．

第一の貢献は，2- レベル・プログラミング・モデルの提案である．本モデルにより，部品開発とアプリケーション開発が分離され，熟練開発者は単一アーキテクチャー向けの最適化に，一般開発者はアーキテクチャーから独立したアプリケーションの構築に集中できる．

第二の貢献は，演算リソースを自動割り当てするコンパイラーの提供である．アプリケーションの各部分の並列度の判断や，演算リソースへの部品割り当ては，利用するシステムに合わせて適切に自動処理されるため，開発者の並列化のノウハウや配置の試行錯誤が必要ない．

第三の貢献は，通信コードの自動生成である．最適な通信方法の判断，部品間の同期，共有リソースの排他制御，メモリー管理などを行うコードを，部品の演算リソースへの配置に応じて自動的に生成する．開発者の手間を省き，見つけにくいバグを抑制できる．

これらの貢献により，研究者などコンピューターの専門家ではない開発者であっても，本フレームワークを通じて少数の熟練開発者の成果を共有でき，多くの一般開発者にもハイブリッド・システムが利用しやすくなる．

3. プログラミング・フレームワーク

3.1 フレームワークの概要

本フレームワークは，2- レベル・プログラミング・モデル，およびコンパイラーによる演算リソースの自動割り当てと通信コードの自動生成からなる（図 2）．

2- レベル・プログラミング・モデルでは，特定ハードウェア向けの最適化ができる少数の熟練開発者によってアルゴリズムを部品化し，多くのアプリケーション開発者はその部品をドラッグ・アンド・ドロップ操作でつなぎ合わせてストリーム・グラフを作り，より複雑なアプリケーションを構築する．

コンパイラーは，リソース・スケジューリングと，通信コードの自動生成および実行形式の生成の 2 段階で処理を行う．第 1 段階で，利用できる演算リソースの中で，アプリケーションのスループットを最大化するように，部品を配置する方法を決定する．第 2 段階で,部品間のデータ通信コードを同期やメモリー管理などと併せて自動的に生成する．

以下，本フレームワークを用いたプログラムの開発手順について順に解説する．

3.2 プログラム部品の開発

プログラム開発の最初のステップは，プログラム部品

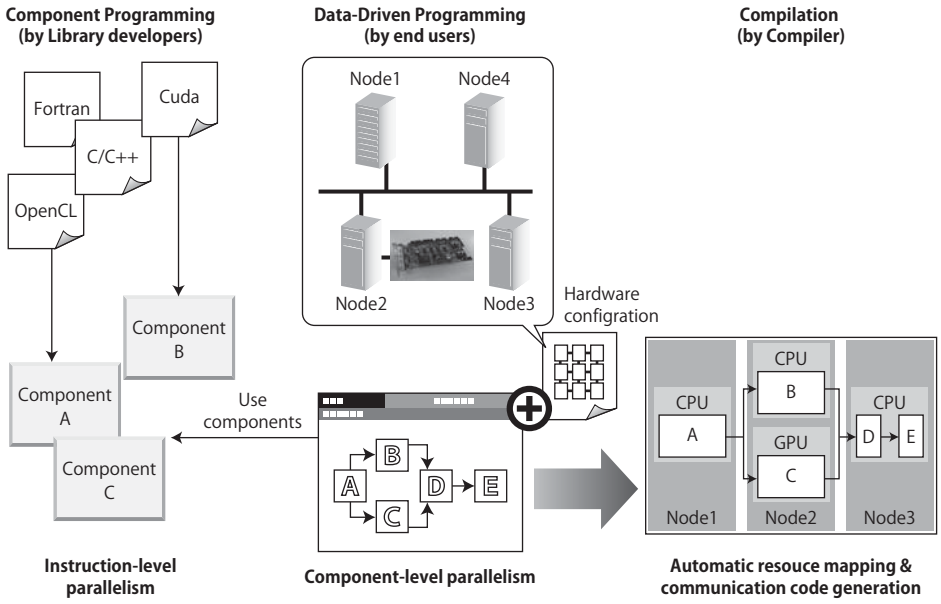

図 2. 本フレームワークの概要

```
/// UDOP UDOP_HELLO_SOURCE
/// -char[14] out

/// UDOP UDOP_HELLO_STREAM
/// +char[14] in
/// -char[14] out

/// UDOP UDOP_HELLO_SINK
/// +char[14] in
```

(a) UDOP 定義

```
#include <string.h>
#include "hello.h"

/// KERNEL hello_source
/// - char[14] out
/// > UDOP_HELLO_SOURCE
/// @x86
void hello_source(char *out) {
   char tmp[14] = "Hello stream!¥0";
   static counter = 0;
   *out = tmp[counter] ^ key;
   counter++;
   if(counter > strlen(tmp))
      counter = 0;
}
```

```
#include "hello.h"
/// KERNEL hello_source
/// + char[14] in
/// - char[14] out
/// > UDOP_HELLO_STREAM
/// @x86
void hello_stream(char *in, char *out) {
   *out = *in ^ key;
}
```

```
#include <stdio.h>

/// KERNEL hello_sink
/// + char[14] in
/// > UDOP_HELLO_SINK
/// @x86
void hello_sink(char *in) {
   printf("%s¥n", *in);
}
```

(b) カーネル定義

図 3. Hello World の UDOP とカーネルの定義

となる UDOP（User Define OPerator）の定義とカーネルの実装である．UDOP は，特定のマイクロ・アーキテクチャーには依存せず，機能とデータ入出力を定義しただけの抽象コンポーネントで，ストリーム・グラフの記述に用いる．UDOP 定義は，/// で始まる指示文数行からなる短いテキスト・ファイルである．UDOP 名と，データの入出力についてデータ型，データサイズ，ポート名を定義する必要がある（図 3（a））．

UDOP を特定のマイクロ・アーキテクチャー向けに実装したものをカーネルと呼ぶ．カーネルの定義文も /// で始まる指示文で，UDOP からデータ入出力の定義を継承するほかに，アーキテクチャーや使用するアクセラレーターの指定などを行う必要がある．カーネルの実装は C 言語，C++ または FORTRAN などで作成可能で，定義されたデータ入出力ポート名と同じ名前の引数を持つ（図 3（b））．1 つの UDOP に対して複数のカーネルを実装することもできる．

プログラム部品の粒度に制約はないが，機能的に完結していて再利用しやすいことが望ましい．例えば，信号処理で周波数解析に多用される高速フーリエ変換（FFT: Fast Fourier Transformation）や連立一次方程式の反復的計算法である SOR（Successive Over-Relaxation，逐次過緩和）法などの単位ならば，扱うデータ量と演算量の観点において，カーネルの最適化，アプリケーションの構築のいずれにも効率がよい．

プログラム部品はシングル・コア用に開発すればよいが，十分に性能が高いことが望ましい．部品開発者は，既存のコンパイラーの多くが提供している自動的な最適化機能を利用できる．

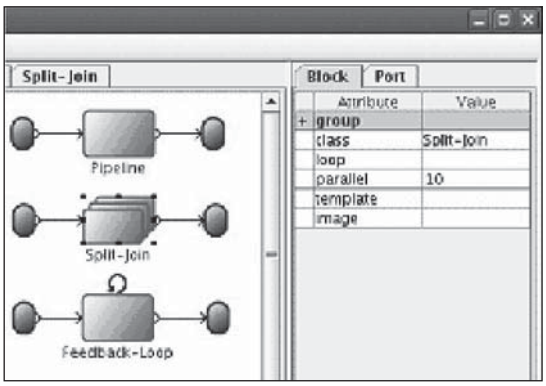

図 4. ビジュアル・プログラミング環境

3.3 ビジュアル・プログラミング

一般開発者は，ドラッグ・アンド・ドロップ操作を主体としたビジュアル・プログラミング環境を利用してアプリケーションの開発を行う．具体的には，UDOP をパレットから選んで画面に置き，UDOP のペアを，データ・フローを表す線で接続したストリーム・グラフを作成する．StreamIt［13］や SPADE［14］などと同様のパイプライン，スプリット・ジョイン，フィードバック・ループの 3 種類のグラフ構造をサポートしている（図 4）．

3.4 リソース・スケジューリング

本フレームワークで書かれたアプリケーション（ストリーム・グラフ）では，各プログラム部品がソフトウェア・パイプラインのステージとして動作する．パイプラインを構成する各ステージの実行時間が短いほど，全体のスループットは向上する．このため，コンパイラーは，ハイブリッド・システムのハードウェア・コンフィギュレーションで与えられたリソース制約の中で，ストリーム・グラフ上の各 UDOP の実行時間の最大値を最小化するように，リソース・スケジューリングを行う．具体的には，並列化や複数の候補からのカーネルの選択などをリスト・スケジューリングに基づいた独自のアルゴリズムによって適切な組み合わせを決定する．

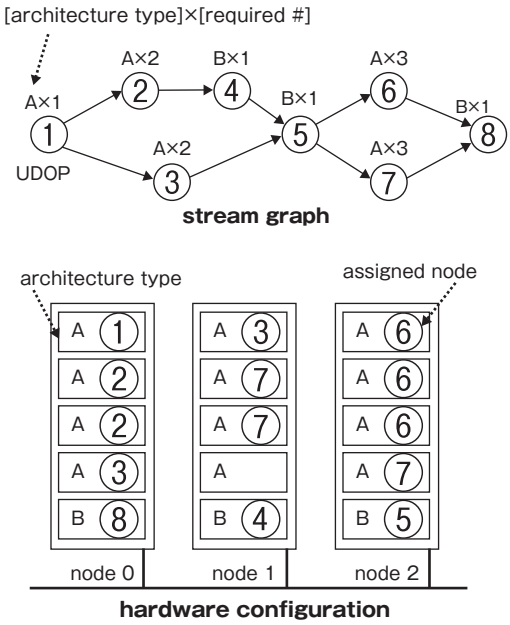

図 5. ネットワーク・エンベッディング

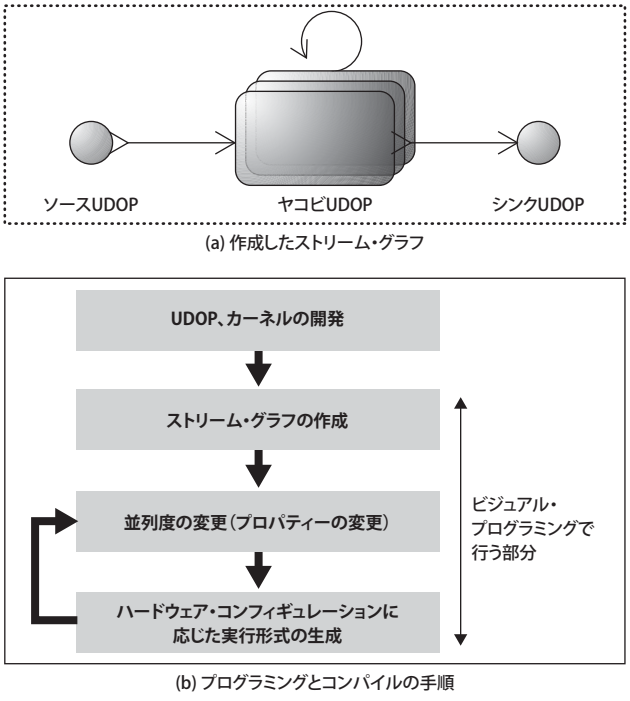

図 6. 実験の概要

表1. リソース配置の結果

#of parallel	SMALL		MIDDLE		LARGE	
	Kernel Type	#of Nodes	Kernel Type	#of Nodes	Kernel Type	#of Nodes
1	GPU	1	GPU	1	GPU	1
2	GPU	2	GPU	2	GPU	2
4	CPU	1	GPU	4	GPU	4
8	CPU	2	CPU	2	GPU	8

UDOP は必ずしも 1 つのカーネルに割り当てられるわけではなく，並列化されたりハイブリッド化されたりする場合もある．従ってコンパイラーは，リソース・スケジューリングに先立って，さまざまなカーネルの組み合わせ（実行パターン）を生成し，それらの実行パターンについて実行時間を計測し，結果をデータベースに記憶する．

筆者らが考案した演算リソース割り当ては以下のアルゴリズムで行う．まず，リソース制約がないものとして，各 UDOP に実行時間が最短の実行パターンを仮割り当てする．次に，リソース制約が満たされるまで，必要リソースのより少ない実行パターンに入れ替える．入れ替えは，現在最も長い実行時間を持つ UDOP よりも実行時間が長くならないものを選ぶ．そのような入れ替えができない場合は現在の実行時間が最短のものを次の実行パターンに入れ替える．なお，このリソース割り当ては多項式時間では解けない組み合わせ問題であるが，実行パターンの入れ替えを常にリソースを減らす方向に制限することで，収束を保証している．

UDOP のカーネルへの置き換えと使用するリソースの選択が完了したら，図 5 に示すように各カーネルの物理的な演算リソースに割り当てと，通信方式の決定を行う（ネットワーク・エンベッディング）．原則としてストリーム・グラフ上の UDOP を幅優先探索で始点に近いタスクから同じノードに配置していく．配置できなくなったら隣接ノードで同じことを繰り返す．

割り当てはコンパイル時に静的に行われ，実行時に変化することはないので，部品ごとに最適な演算リソースを使うことが保証される．

マルチプロセッサー・スケジューリングの従来手法の多くがアプリケーション全体の実行時間を最小化するように静的なスケジューリングを行う［18］［19］［20］．しかし，ハイブリッド・システムでは，アプリケーションのある部分を用途固定のアクセラレーターで実行する場合，その間はほかのプロセッサーはアイドルになるため実効効率が低下するという課題がある．本フレームワークではアプリケーションをパイプライン化してスループットを最大化するように演算リソースを配置するため，問題が多数ある場合はシステム全体の稼働率を高く保つことができる．

3.5 通信コードの自動生成

リソース・スケジューリングに基づき，ネットワークやメモリー・コピー，あるいはポインターのやり取りなど，部品間通信コードを生成する．生成されたソースコードは，gcc や nvcc などのコンパイラーによって実行形式に

変換される．本フレームワークではMPMD（Multiple Program Multiple Data）モデルで実行形式を生成する．さらに，アプリケーションの実行開始と終了待機を行うコントロール・プログラムと，実行形式の各ノードへの配置と起動を行うためのスクリプトも自動生成する．

4. アプリケーションへの適用例

4.1 姫野ベンチマーク

姫野ベンチマーク［21］は，ポアソン方程式をヤコビの反復法で解く際の演算時間を計測する流体シミュレーション・プログラムである．本演算は，非圧縮系の流体シミュレーションにおいて主要な演算であり，実アプリケーションに近い評価ができると考えられる．

本論文では，本フレームワーク上に実装した姫野ベンチマークを対象に，ハイブリッド・システムへの適用および効果を確認する．姫野ベンチマークは，データ並列性が高く，格子ごとの演算は同時に実行できるが，隣接する格子の値を使うため反復ごとに分割領域の境界でノード間通信が必要になる．筆者らが開発中のコンパイラーが，データの依存性を解析して上記ノード間通信を自動生成するため，熟練開発者，一般開発者ともに本ノード間通信を明示的にプログラミングする必要はない．

4.2 ベンチマーク結果と考察

本フレームワークを用いて適切なリソース配置と通信コードの生成が行われることを確かめるため，図６（a）のような単一のストリーム・グラフを作成し，並列度を変更して実行形式を生成し（図６（b）），それぞれベンチマークの実行時間を計測した．実行環境としてRIKEN Integrated Cluster of Clusters（RICC）［5］の多目的PCクラスターを利用した．多目的PCクラスターは，2基のIntel® Xeon® 5570と1基のNVIDIA Tesla C1060を搭載したハイブリッド・ノードが，InfiniBandと10 Gbit Ethernetで接続されている．

演算はすべて単精度浮動小数点を用い，データ・サイズはSMALL（64 × 64 × 128），MIDDLE（128 × 128 × 256），LARGE（256 × 256 × 512）の3種類を用意した．ストリーム・グラフにはソースUDOP，シンクUDOP，ヤコビ法UDOPがあり，ヤコビ法UDOPについてループと並列化を設定した．ループ回数は800に固定し，並列度を1から8まで変更した．並列度の変更は，ストリーム・グラフ構造およびカーネルの書き換えを行うことなくビジュアル・プログラミング環境上のパラメーター操作のみで実現した．ハードウェア・コンフィギュレーションは，CPUとGPUのハイブリッド・システムを用い，ヤコビ法UDOPにはCPU用カーネルとGPU用カーネルの2種類を用意した．なお，GPU用カーネルは，入力サイズが32 × 32 × 64以上でなければならない制約がある．また，ソースUDOP，シンクUDOPもそれぞれCPUを1コアずつ利用する．

生成された実行形式の内容を表１に示す．一般にGPUの方がCPUよりも高速であるため，GPU用カーネルが優先的に選ばれる．ただし，一部のケースでは

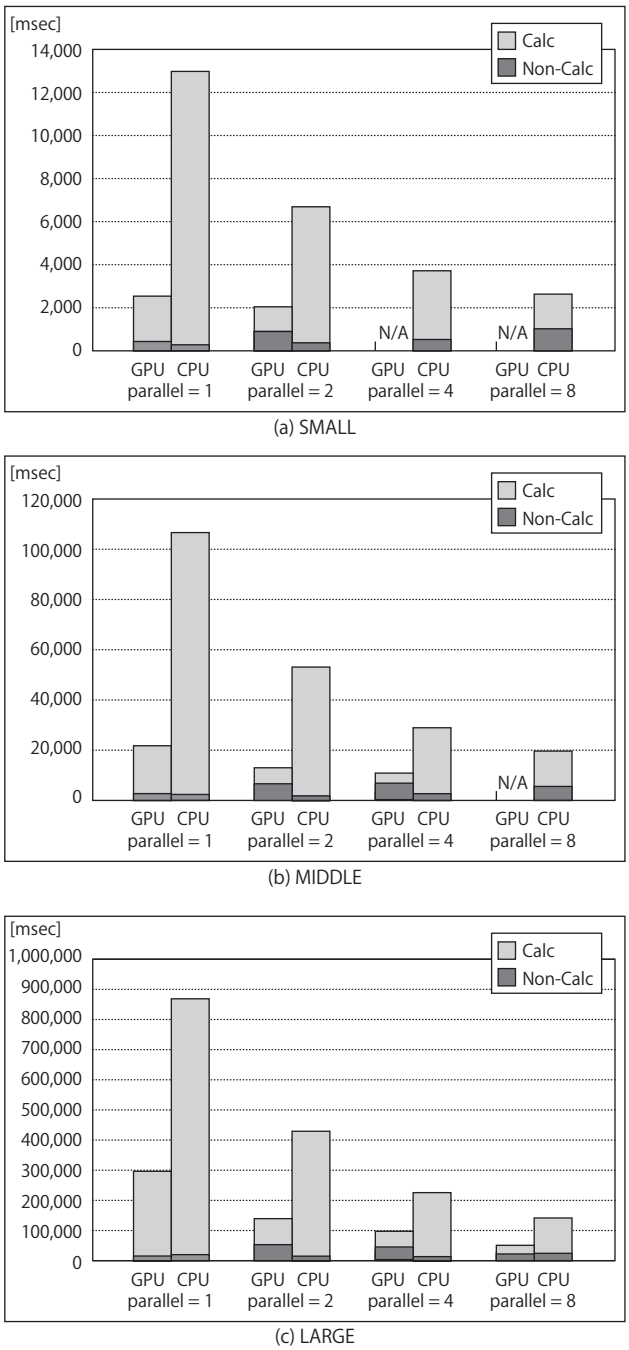

図 7. 姫野ベンチマークの実行時間

GPU 用カーネルのサイズ制約のため CPU 用カーネルが選ばれている（網掛け部分）．GPU 用カーネルが選ばれた場合，GPU は 1 ノードに 1 台であるため，並列度に等しいノード数が使われる．一方，CPU 用カーネルが選ばれた場合は，1 ノードあたり 8 コアまで利用できるため，8 並列時のみ 2 ノードになる．

並列化に伴う領域分割もコンパイラーが自動的に生成するが，現在のプロトタイプでは 1 次元分割のみをサポートしている．異なるノードに配置された部品間の通信には 10 Gbit Ethernet を，同じノード内の部品間通信にはメモリーのコピーを使った．

参考として，GPU 用カーネルのみ，CPU 用カーネルのみを用いた場合の，ヤコビ法 UDOP の実行時間を図 7 に示す．Calc はカーネル関数の累積実行時間，Non-Calc は通信を主としたカーネル関数以外のランタイム処理時間で，単位はミリ秒である．parallel=N はヤコビ法 UDOP の並列度が N であることを表している．

Calc 部分は並列度に反比例して減少しており，よくスケールしている．Non-Calc 部分は,大きな変化はないが，GPU 用の場合は 2 並列以上でノード間のネットワーク通信が発生するため大きな値になっている．CPU 用カーネルの場合は，8 並列時以外は 1 台のノードの共有メモリー上で実行されるため通信の影響は小さい．

本実験で用いたカーネル関数は性能に関して特別な最適化は施していないが，姫野ベンチマークについては青木や成瀬らによる高速な実装がある［22］［23］．例えば，これらの実装をカーネル関数として用いることで，一般ユーザーは専門的な最適化が施された成果を利用することが可能である．

5. まとめと今後の課題

本論文ではハイブリッド・システム向けのプログラミング・フレームワークについて提案した．本フレームワークでは，熟練開発者による最適化済みプログラムの部品化と，一般開発者によるデータ・フロー開発の 2- レベルのプログラミング・モデルにより，性能と生産性の両立に関する課題を解決した．また，プログラム部品の演算リソースへの割り当ては，筆者らが新規に考案したスケジューリング・アルゴリズムを用いた自動化ツールにより解決した．通信部分のプログラミングも，部品同士の配置に応じて適切な通信手段を利用するコードを自動生成することで解決した．

応用例として姫野ベンチマークを本フレームワークに移植し,複数の CPU と GPU で並列演算する実験を行った．実際のアプリケーションにおいても，並列度に応じて演算リソースの種類や台数を自動的に選択し，性能が良好にスケールしていることを確認できた．

今後，このようなハイブリッド・システム向けのプログラミング・フレームワークは，多種の演算リソース，通信リソースを活用するクラウド・コンピューティングにおいて，戦略的に重要な要素になり得る．CloudBurst™［24］や，Blue Gene®，BladeCenter® など［25］の IBM のマルチコア，あるいはハイブリッド・システムへの適用や，より実用的なアプリケーションへの適用を評価していきたい．また，通信時間も評価したスケジューリングや，演算と通信の並列動作などを導入して，さらに性能を向上させたい．

謝辞

RICC への適用は，理化学研究所様調達案件「RICC 多目的クラスタ開発環境の最適化にかかわるプログラム開発」により行われました．ここに感謝いたします．また，東京基礎研究所 吉澤 武朗さんにはスケジューリング・アルゴリズムに多くのアドバイスを，GBS 窪 陽祐さんにはビジュアル・プログラミング環境の開発にご協力をいただきました．ここに深く感謝いたします．

参考文献

［1］ “TOP500 Super Computer Sites,” http://www.top500.org/
［2］ K. J. Barker, et al: “Entering the petaflop era: the architecture and performance of roadrunner,” in SC ’08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing. Piscataway, NJ, USA: IEEE Press, pp. 1–11 (2008).
［3］ NVIDIA Tesla: http://www.nvidia.co.jp/object/tesla_computing_solutions_jp.html
［4］ TSUBAME 計算サービス, http://www.gsic.titech.ac.jp/~ccwww/
［5］ RICC, http://accc.riken.jp/ricc.html
［6］ IBM zEnterprise, http://www.ibm.com/jp/press/2010/07/2301.html
［7］ H. Sutter: “The free lunch is over: A fundamental turn toward concurrency in software,” in Dr. Dobb’s Journal, vol. 30, no. 3, (2005).
［8］ MPI, http://www.mcs.anl.gov/research/projects/mpi/
［9］ OpenMP, http://www.openmp.org.
［10］ P. Bellens, et al: “Cellss: a programming model for the cell be architecture,” in SC ’06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing. New York, NY, USA: ACM, p. 86 (2006).
［11］ P. Charles, et al: “X10: an object-oriented approach to non-uniform cluster computing,” in OOPSLA ’05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. New York, NY, USA: ACM, pp. 519–538 (2005).
［12］ Mark A. Franklin, et al: “Auto-Pipe and the X Language: A Pipeline Design Tool and Description Language,” in Proc. of Int’l

Parallel and Distributed Processing Symp., (2006.4).
［13］W. Thies, et al: “Streamit: A language for streaming applications,” in CC ’02: Proceedings of the 11th International Conference on Compiler Construction. London, UK: Springer- Verlag, pp. 179–196 (2002).
［14］B. Gedik, H. et al: “Spade: the system s declarative stream processing engine,” in SIGMOD ’08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data. New York, NY, USA: ACM, pp. 1123–1134 (2008).
［15］K. Fatahalian, et al: “Sequoia: programming the memory hierarchy,” in SC ’06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing. New York, NY, USA: ACM, p. 83 (2006).
［16］I. Buck, et al: “Brook for gpus: stream computing on graphics hardware,” in SIGGRAPH ’04: ACM SIGGRAPH 2004 Papers. New York, NY, USA: ACM, pp. 777–786 (2004).
［17］NVIDIA Corporation, “CUDA Zone,” http://www.nvidia.com/object/cuda_home_new.html
［18］T. L. Adam, et al: “A comparison of list schedules for parallel processing systems,” in Communications of the ACM, vol. 17, no. 12, pp. 685–690 (1974).
［19］C. McCreary, et al: “A comparison of heuristics for scheduling dags on multiprocessors,” in Proceedings of the 8th International Symposium on Parallel Processing. Washington, DC, USA: IEEE Computer Society, pp. 446–451 (1994).
［20］H. Topcuoglu, et al: “Task scheduling algorithms for heterogeneous processors,” in HCW ’99: Proceedings of the Eighth Heterogeneous Computing Workshop. Washington, DC, USA: IEEE Computer Society, p. 3 (1999).
［21］High Performance Computing, 理化学研究所, http://accc.riken.jp/HPC/HimenoBMT.html
［22］青木 尊之 : “2007 年度 理研ベンチマーク報告 － 4GPU による驚異の高速化－,” http://www.nr.titech.ac.jp/~taoki/Whatsnew/Doc/RikenBMT2007.pdf (2008, 2010-08-20 閲覧).
［23］成瀬 彰 , et al: “GPGPU 上での流体アプリケーションの高速化手法 : 1GPU で姫野ベンチマーク 60GFLOPS 超,” IPSJ SIG Notes 2008(99), pp. 49-54, (2008.10.08).
［24］IBM CloudBurst, http://www.ibm.com/software/jp/tivoli/products/cloudburst/
［25］IBM ディープ・コンピューティング I HPC 製品, http://www.ibm.com/software/jp/tivoli/products/cloudburst/

日本アイ・ビー・エム株式会社
マイクロエレクトロニクス事業部
主任ソフトウェア・エンジニア

土居 意弘 Mumehiro Doi

［プロフィール］
2003 年 IBM 入社．画像処理システム, Cell/B.E. アプリケーション開発などの組み込み技術経験を経て，現在はハイブリッド・システムの研究開発に従事．IEEE，電子情報通信学会，情報処理学会各会員．

日本アイ・ビー・エム株式会社
東京基礎研究所
主任研究員

村瀬 正名 Masana Murase

［プロフィール］
2003 年 IBM 入社．以来東京基礎研究所にてシステムセキュリティ，システムソフトウェア，HPC 向けアプリケーションフレームワークの研究開発に従事．IEEE, ACM, 電子情報通信学会，情報処理学会各会員．

日本アイ・ビー・エム株式会社
東京基礎研究所
副主任研究員

前田 久美子 Kumiko Maeda

［プロフィール］
2009 年，日本 IBM 入社．東京基礎研究所にて，HPC 向けアプリケーション・フレームワークの研究開発に従事．博士（情報科学）．

日本アイ・ビー・エム株式会社
東京基礎研究所
シニア・テクニカル・スタッフ・メンバー

小松 秀昭 Hideaki Komatsu

［プロフィール］
1985 年 IBM 入社．以来東京基礎研究所にて Prolog コンパイラー，HPF コンパイラー，Java™ JIT コンパイラーなどのシステムソフトウェア，シミュレーション用の言語，シミュレーションシステムの研究開発に従事．ACM, 情報処理学会各会員．博士（情報科学）．