2008－ARC－180 (10)
2008／10／21

シンプルで効率的なメニーコアアーキテクチャの開発

植原 昂[†1] 佐藤真平[†1] 森谷 章[†1]
藤枝直輝[†1] 高前田伸也[†2] 渡邉伸平[†1]
三好健文[†3] 小林良太郎[†4] 吉瀬謙二[†1]

近い将来，プロセッサのコア数を数十から数百搭載したメニーコアの時代に到達すると考えられる．そこで我々は，簡易な構成で実現可能なメニーコアアーキテクチャモデルを提案する．このモデルの1つの実現形態である M-Core Ver.1.0(Many-Core Version 1.0) アーキテクチャ及び，その評価のためのプロセッサシミュレータ，SimMc を開発している．本稿では，提案するアーキテクチャモデル及びその実現方式について述べ，SimMc 上で実行する並列プログラムの記述法について解説する．また，SimMc を用いて，コア数の増加に伴う速度向上率を評価する．

Development of Simple and Effective Many-Core Architecture

Koh Uehara,[†1] Shimpei Sato,[†1] Akira Moriya,[†1]
Naoki Fujieda,[†1] Shinya Takamaeda,[†2] Shimpei Watanabe,[†1]
Takefumi Miyoshi,[†3] Ryotaro Kobayashi[†4] and Kenji Kise[†1]

In the near future, high performance many-core processors will have dozens of hundreds of cores. In this paper, we introduce the concept of a simple and effective many-core architecture model. Then we propose the many-core architecture M-Core Ver.1.0 on this model and show a sample source code for this architecture. Our evaluation results with our software simulator SimMc show that M-Core architecture achieves good speedup for some benchmark programs.

1. はじめに

プロセッサの性能向上と消費電力の削減を目的に，広い分野にわたり，複数個のコアを搭載するチップマルチプロセッサが主流となりつつある．その背景には，プロセスの微細化に伴う配線遅延の相対的な増加などにより，単一コアの高速化が非効率的になっていることがある．今後は，半導体の集積度向上により，さらに多くのコアを集積するメニーコアプロセッサへと向かうと考えられる．

プロセッサアーキテクチャの今後の発展の方向性を明らかにするため，様々なメニーコアアーキテクチャモデルの検討が不可欠となる．そこで，我々は多数の簡素化したコアを効率的に利用できるメニーコアアーキテクチャモデルを提案する．本モデルの1つの実現方式として，数十から数千までのコア数をターゲットとするメニーコアアーキテクチャM-Core Ver.1.0(Many-Core Version 1.0) を提案する．また，メニーコアプロセッサのソフトウェアシミュレータ SimMc の開発も行っており，SimMc を用いた評価から M-Core アーキテクチャの有効性を明らかにする．

本稿の構成を述べる．2章では，提案するアーキテクチャモデルのコンセプトと概要について述べる．3章でこのモデルの1つの実現方式である M-Core アーキテクチャについて解説し，4章で M-Core のシミュレータとシミュレータ上で実行するプログラムに着いて触れる．5章で M-Core アーキテクチャを評価し，6章でまとめる．

†1 東京工業大学 大学院情報理工学研究科
Graduate School of Information Science and Engineering, Tokyo Institute of Technology
†2 東京工業大学 工学部情報工学科
Department of Computer Science, Tokyo Institute of Technology
†3 東京大学 大学院創造情報学専攻
Dept. of Creative Informatics, The Univ. of Tokyo
†4 豊橋科学技術大学 情報工学系
Department of Information and Computer Sciences, Toyohashi University of Technology

2. シンプルで効率的なアーキテクチャ

2.1 コンセプト

数個のコアを搭載するマルチコア時代から，さらに多くのコアを集積するメニーコア時代へと向かっている．これを踏まえ，我々の開発するメニーコアアーキテクチャモデルでは，1チップに搭載可能なコア数が

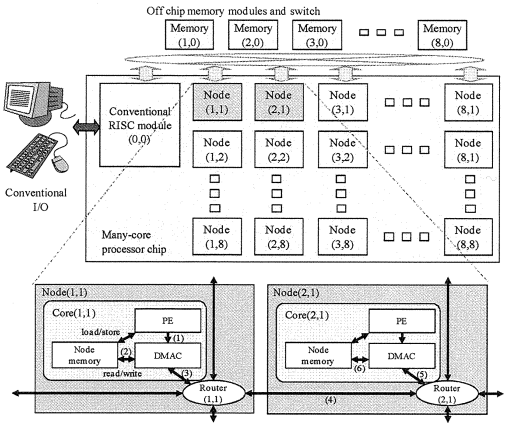

図 1 提案するメニーコアアーキテクチャモデル．チップには，汎用プロセッサコアである module(0,0) と，メッシュ状に配置された多数のノードを搭載し，オフチップメモリモジュールと接続する．下部は，ノードの詳細の拡大図である．各ノードはコアとルータから構成される．コアには，PE，DMAC，ノードメモリを格納する．

数十から数千というオーダまで増え続けることを想定している．我々は，これらの豊富なコアを効率的に利用するアーキテクチャの開発を行っている．重視するコンセプトを以下に列挙する．

- 各ノードのコアから従来の投機処理機構のハードウェア量を削減し，コアの高効率化を目指す．
- 複雑な割り込み制御機構を排除することで，各ノードのコアの更なる高効率化を目指す．
- 小規模な均一のコアを多数並べる構成を採用し，システムソフトウェアとの協調により，チップの高性能化を目指す．
- データ通信のオーバヘッドを削減することで，高い並列化効率を目指す．

2.2 メニーコアアーキテクチャモデル

我々の開発するアーキテクチャモデルを**図 1** に示す．8×8 個に配置されたユニットをノードと呼び，それ以外をモジュールと呼ぶ．本稿では 2 次元のメッシュ接続を前提とする．ノード及びモジュール間のデータ転送には，チップ内ネットワークを使用する．ノード及びモジュールにはチップ内で一意の ID（識別子）を割り当てる．ノードやモジュール (0,0) は，ID を用いて他のノード及びモジュールと通信することが可能である．ID は X 座標と Y 座標の組合せで表現する．本稿では，ID が (X,Y) となるノードを Node(X,Y) と表記する．左上に位置するモジュール (0,0) は，従来の汎用プロセッサと同等の機能を有するコアを持ち，外部 I/O 処理などの割り込み制御に加え，ノードに対するタスク割り当てを行わせることを想定している．図 1 の上部に配置されているモジュールはメインメモリであり，全てのノード及びモジュールがネットワークを介してアクセスする．ノードではアプリケーションプログラム等が実行される．

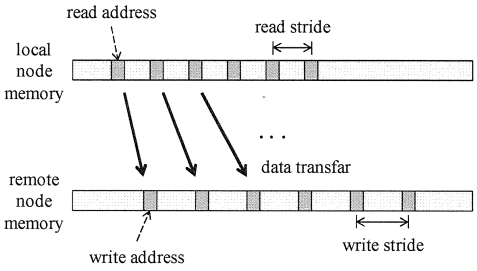

図 2 DMA_PUT の様子．DMA 転送により，ローカルノードからリモートノードに DMA 転送する．DMA 転送では，読み出しメモリアドレス，書き込みメモリアドレス，読み出し時のストライド値，書き込み時のストライド値，送信サイズを指定して，柔軟な転送が可能である．

図 1 の下部にノードの詳細の拡大図を示す．ノード内部にはコアとルータがある．コアは，演算処理ユニットである PE（プロセッシングエレメント），ノードメモリ（各ノードが持つ小規模メモリ），DMAC(Direct Memory Access Controller) で構成される．自ノードのノードメモリに対しては，PE のロード/ストア命令によりアクセスする．他のノード及びモジュールのメモリに対しては，DMAC に対して DMA 転送を発行することでアクセスする．DMA 転送は，DMA_PUT と DMA_GET に大別される．前者は自ノードのノードメモリにあるデータを，他ノードのノードメモリ又はメインメモリに転送する．後者は他ノードのノードメモリ又はメインメモリにあるデータを，自ノードのノードメモリに転送する．

DMA_PUT を例に，処理の流れ (1) から (6) を図 1 の下部に示す．これは，Node(1,1) が Node(2,1) に DMA 転送する様子である．簡単のため，以下では，Node(1,1) をローカルコア，Node(2,1) をリモートコアと呼ぶことにする．以下で示す処理を行う．(1) PE がリモートコアに対する DMA 転送を発行する．(2) ローカルコアの DMAC がノードメモリから転送するデータを読み出す．(3) ローカルコアの DMAC がデータをパケットに加工し，ローカルコアのルータに転送する．(4) ローカルコアのルータがリモートコアのルータにパケットを転送する．(5) リモートコアのルータから DMAC にパケットを転送する．(6) リモートコアの DMAC がノードメモリにデータを書き込む．

DMAC は，読み出しメモリアドレス，書き込みメモリアドレス，読み出し時のストライド値，書き込み時のストライド値，送信サイズの 5 つのパラメタを用いて DMA を実行する．ストライドを利用することにより，**図 2** のように一定間隔おきにメモリアクセスが可能である．これは 2 次元配列の列データをアクセスする場合などに有益である．ストライド値はリード/

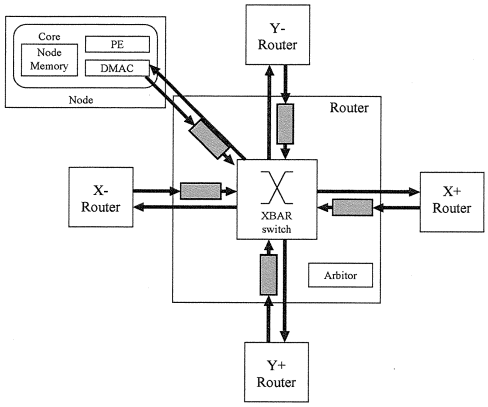

図 3 ルータの接続の様子．4 方向のルータと 1 つの DMAC とデータを通信する．

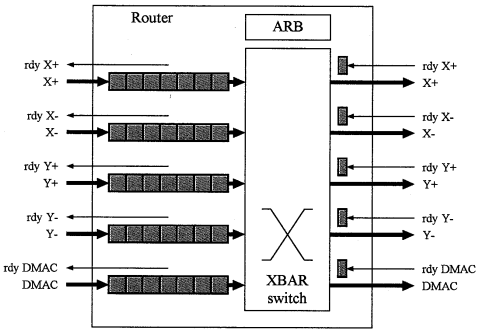

図 4 ルータ内部構造．

ライトについて別々に指定可能である．また，リモート ID として自身の ID を指定することも可能で，ノード内で指定したストライドを用いてリードとライトを処理する．これにより，2 次元配列の列データを 1 次元配列に整列するといった作業を効率的に処理できる．

2.3 ノードの概要

PE は 2-way スーパスカラ程度のシンプルな RISC プロセッサとする．

ノードメモリは PE が直接アクセス可能なメモリであり，数百 KB 程度の容量とする．ノードメモリに収まらないデータは，メインメモリ又は他ノードのノードメモリに格納する．

ルータは**図 3** に示す通り，隣接する 4 つのルータ及び 1 つの DMAC を接続する．データの転送単位であるパケットは，入力線を経由して入力バッファに格納され，クロスバーを通り，しかるべき方向へと出力される．ルータはアービタを持つ．複数方向から入力されたパケットが 1 つの方向に同時に出力されるのを防ぐため，アービタが 1 つのパケットにのみ，出力許可を与える．出力許可を与えられないパケットは，許可を得るまで入力バッファに留まる．これは出力線が 1 パケット分のビット幅しか備えていないことによる．

ノードメモリ，DMAC，ルータは，特定のコアの命令セットアーキテクチャには依存しない．このため，様々な命令セットの PE を実装することが可能である．

3. M-Core アーキテクチャの提案

メニーコアアーキテクチャモデルの 1 つの実装方式として M-Core アーキテクチャを定義する．

PE には，比較的単純かつ効率的であるという理由から MIPS32 プロセッサを採用する．広く普及しており，開発環境を容易に構築できるのも 1 つの利点である．PE は 1 サイクルに 1 命令を実行するものとする．ただし，PE のネットワークに対する処理速度は可変とする．

ノードメモリには，命令メモリとデータメモリの 2 つのメモリを用いる．メモリアクセスは 1 サイクルで完了し，1 度のアクセスで 1 ワード (32bit) のリード/ライトを行う．命令メモリは，PE による命令の読み出し，DMAC のリード/ライトを同時にサポートするため，3 ポートを必要とする．データメモリは，PE によるロード又はストアと，DMAC のリード/ライトを同時にサポートし，こちらも 3 ポートが必要である．32bit のアドレス空間を数百 KB のメモリで処理するには，メモリ管理機構が必要になる．ただし，本バージョンではノードメモリは十分豊富な容量を備えるものとし，メモリ管理機構の実装を省略している．

DMAC の内部にはメモリマップされたレジスタを複数配置する．PE がそのレジスタに対して必要な情報をストアすることにより，DMA 転送を実現する．また，DMAC とルータの間は全二重通信が可能であり，DMAC はフリット（パケットは複数のフリットで構成され，1 つのフリットは 32bit の容量をもつ）の送受信を 1 サイクルで同時に処理する．

ルータの内部構造を**図 4** に示す．各入力バッファにはいくらかのフリットを格納する領域を備える．仮想チャネルは持たない．ルータ間の通信には入出力の信号線を別に用意し，ルータは 1 サイクルでフリットの送受信を同時に行う．ルーティング手法には XY 次元順ルーティングを採用する．この手法は，例を挙げると，Node(1,1) から Node(3,2) に向かうパケットが流れる経路は，Node(1,1)→ Node(2,1)→ Node(3,1)→ Node(3,2) となるように，X の座標値が宛先の X 座標と一致するまで進み，一致したら Y 座標で同様に処理する，というものである．フロー制御には，Xon/Xoff 手法を採用する．これは，各ルータが入力バッファの空きの有無を 1bit で表現し，隣接するルータに通知する手法である．Xon/Xoff によるフロー制御は，DMAC との通信にも使用される．ルータ内ではパイプライン化はしておらず，経路選択，アービトレーション，伝

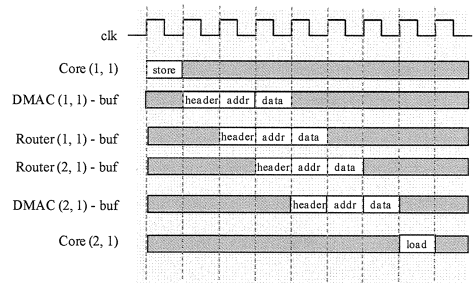

図 5 DMA 転送のタイミング.

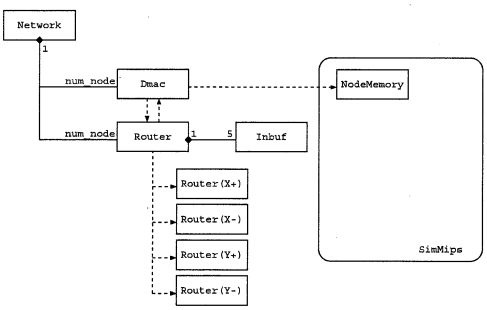

図 6 SimMc のクラス図. Network クラスが全ての Router と Dmac を所有する. Router は Dmac と上下左右の 4 つの Router と接続する. SimMips については, ネットワークから参照するのはメモリクラスのみであるため, それ以外のクラスの表記は省略している.

送を 1 サイクルで完了する[1]. すなわち, ルータ間及びルータ-DMAC 間は 1 サイクル/1 ホップとする.

図 5 を用いて, 隣接する Node(1,1) から Node(2,1) へ 1 ワードを DMA 転送する場合を例に, DMA 転送のタイミングを説明する. Node(1,1) の PE がストア命令により DMA 転送を要求すると, 次のサイクルにはヘッダフリットが送信される. ヘッダフリットには送信先の ID を格納する. 次に, 書き込むメモリアドレスを示すアドレスフリット, データを格納するデータフリットと続き, DMA 転送の開始から 7 サイクル後にロード可能である. DMAC のハードウェア構成をシンプルにすることで, 以上のようにデータ通信のオーバヘッドを削減する.

4. M-Core の評価環境

4.1 メニーコアシミュレータ SimMc Ver.1.0

本説では, M-Core アーキテクチャを模倣するシミュレータ SimMc について述べる.

図 6 に SimMc を C++で実装した際のクラス関係図を示す. 図中の実践はあるクラスの中で別のクラスのインスタンスが生成される関係を表している. 線の両端にある数値は多重度を表している. 例えば Net-

497	core/board.cc	773	core/mipsinst.cc
309	core/cp0.cc	227	core/simloader.cc
693	core/define.h	272	main.cc
308	core/device.cc	873	network.cc
980	core/fpu.cc	388	network.h
307	core/memory.cc	40	output.cc
912	core/mips.cc	6579	total

表 1 SimMc Ver.1.0 のファイル構成とソースコードの行数. core ディレクトリ内は, SimMips のソースコード, カレントディレクトリにあるのは, SimMc 用に追加したソースコードである. total は全ソースコードの行数を表す.

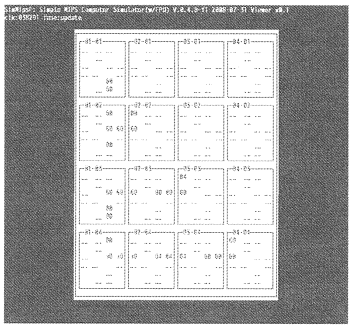

図 7 SimMc:デバッグ時のスクリーンショット. 図は 4×4 のノードで実行した場合であり, Node(2,2), Node(3,3), Node(4,4) が Node(1,1) に対してパケットを送信している状態を示している. 特に Node(2,2) から Node(1,1) への経路にパケットが集中し, この部分のネットワークがボトルネックとなっていることが確認できる.

work と Dmac の関係では, 1 つの Network クラスが num_node 個の Dmac インスタンスを生成することを表す. また, 図中の破線は矢印の元のクラスが矢印の示す先のクラスを参照することを表している. Network, Router, Dmac, Inbuf クラスは, SimMc 用に新規に用意したクラスである. PE とノードメモリには, MIPS32 プロセッサの機能レベルシミュレータである SimMips[2] を使用する. 機能レベルシミュレータとは, 1 サイクルで 1 命令を処理するよう簡素化されたシミュレータである.

シミュレーション結果の追跡を容易にするため, 2 種類のデバッグ出力を用意した. 1 つは DMAC におけるフリットの送受信履歴を全て出力するものである. これは, DMA 転送命令の発行や, パケットの到達確認に使用する. もう 1 つは, **図 7** に示す CUI である. これは, ネットワーク上を流れるパケットを視覚的に確認する際に利用する. CUI では, 簡単なキー操作で任意の時間軸におけるパケットの位置を確認することが可能で, コマ送りすることにより, パケットの流れやネットワークの混雑度などを調べる場合にも適している.

SimMc はいくつかのオプションを備えており, 様々な構成でのシミュレーションが可能である. 変更可能な

項目には，実行するコア数（列数と行数で指定），ネットワークに対するコアの相対的な処理速度，DMACによるメモリリード時のレイテンシを自由に設定可能である．これにより，コア数を増加させることによる速度向上率の評価や，ネットワークに対するコアの高速化，メインメモリにアクセスした際のシミュレーションなどが可能である．

4.2 API の導入

SimMc 上で動作するアプリケーションの開発では，メモリマップを利用することで，様々な機能を実現する．例えば，自身のコア ID の取得，DMA 転送要求といった，メニーコアならではの機能や，開始から現在までのクロックサイクル数の取得といった，シミュレーション時に使用する機能を提供する．しかし，メモリマップに対するアクセスは，比較的低レベルのプログラミングを余儀なくされる上に，マップされるアドレスが変更される可能性もあり，その都度プログラムを書き換えるのは非効率的である．

そこで，メモリマップに対する全てのアクセスを，API としてアプリケーションプログラマに対して，SimMc 用に作成した関数群をまとめたライブラリ MClib を提供する．次節で，MClib を用いたプログラムの記述例を紹介する．

4.3 プログラムの記述例

作成したベンチマークプログラムから Equation Solver Kernel を例に挙げ，SimMc 上で実行する並列プログラムの記述法について解説する．Equation Solver Kernel の処理内容は，2 次元配列の全要素に対して，四近傍とその中心の平均で値を更新する処理を複数ステップ実行するものである．全コアの処理領域が均等になるように 2 次元座標を格子状に区切り，各コアに処理を割り振ることで，並列処理を実現する．ただし，各コアの担当領域の境界に位置する配列要素については，隣接するコアが参照する必要がある．そのため，ステップ毎に全コアにおいて，隣接ノードに対する DMA 転送が必要である．また，DMA 転送の完了前に次のステップを開始すると整合性が取れなくなることから，DMA 転送が完了したことを保証するため，ノード間で同期をとる必要もある．

```
1  /***** global variables *****/
2  unsigned int value[Y_SIZE][X_SIZE];
3  unsigned int n_val[Y_SIZE][X_SIZE];
4
5  int main(int argc, char *args[]) {
6      MC_init(&id_x, &id_y, &rank_x, &rank_y);
7      init(value);
8      MC_setidxy(&id[0], id_x, id_y - 1);
9      ...
10
11     MC_barrier();
12     MC_clock(&timestamp_start);
13
14     /***** main loop *****/
15     for (step = 0; step < STEPS; step++) {
16         /**** computation of n_val *****/
17         for (y = begin_y; y <= end_y; y++)
18             for (x = begin_x; x <= end_x; x++)
19                 n_val[y][x] =
20                     (value[y][x] + ... )/5;
21
22         /***** update *****/
23         for (y = begin_y; y <= end_y; y++)
24             for (x = begin_x; x <= end_x; x++)
25                 value[y][x] = n_val[y][x];
26
27         /***** synchronize (1) *****/
28         MC_barrier();
29
30         // send to the upper core
31         if (id_y != 1)
32             MC_dma_put(id[0],
33                        value[begin_y] + begin_x,
34                        value[begin_y] + begin_x,
35                        (end_x-begin_x+1)*sizeof(int),
36                        sizeof(int), sizeof(int));
37         // send to the left core
38         ...
39
40         /***** synchronize (2) *****/
41         MC_barrier();
42     }
43     MC_barrier();
44     MC_clock(&timestamp_end);
45
46     ...
47
48     if (id_x == 1 && id_y == 1)
49         MC_printf("elapsed cycles = %10d\n\n",
50                 timestamp_end - timestamp_start);
51     ...
52
53     MC_finalize();
54     return 0;
55 }
```

図 8 Equation Solver Kernel のソースコード．要所のみを抜き出している．

図 8 は，Equation Solver Kernel のソースコードの main 関数である．関数名が MC_で始まるものは，MClib のライブラリ関数であることを表している．6 行目の MC_init は MClib の初期化処理を行い，引数にある，4 つの値を更新する．id_x と id_y は自身のノード ID が格納される．rank_x と rank_y には X 座標と Y 座標がともに最大となるノードの ID が格納される．rank_x と rank_y はチップ上のメッシュサイズを把握するために使用する．11 行目の MC_barrier は，チップ上の全てのコアで同期をとる．先にこの関数を実行したコアは，全てのコアがこの関数を実行するまで待ち合わせる．12 行目の MC_clock は，シミュレーション開始から現在までの実行サイクル数を取得するもので，ベンチマークの評価のために使用する．シミュレーション終了後には実行サイクル数を出力するようにしてあるが，初期化処理などの評価する部分とは本来関係無い処理に費したサイクル数を省いた正確な評価を行う場合にこの関数を利用する．15 行目からメインループを開始する．17〜25 行目で担当領域の計算及び，配列要素の更新を行う．28 行目で全コアで同期をとった後，31 行目以降で，自身の担当領域の境界に当たる配列要素を隣接コアに送り，次のステップで必要となるデータを転送する．隣接コアに対するデータの送信は，32 行目にある MC_dma_put で行う．引数は，送り先ノード ID，自ノードのノードメモリから読み出す先頭のメモリアドレス，宛先ノードのノードメモリに書き込む先頭のメモリアドレス，送信するバイト数，自ノードでの読み出し時のストライド値，宛先ノードでの書き込み時のストライド値，である．宛先ノード ID は 8 行目の MC_setidxy で指定している．これは，自身の ID よりも Y 座標が 1 小さ

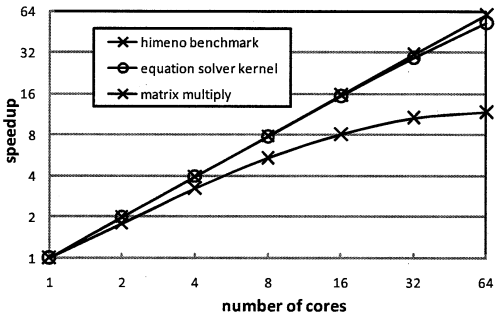

図 9 Equation Solver Kernel のシミュレーション結果より，SimMc を用いてコア数を変化させて測定した M-Core アーキテクチャの速度向上率.

い値を指定しているので，上側に隣接するノード ID を表すことになる．読み出しメモリアドレスと書き込みメモリアドレスに同じアドレスを使用しているのは，全てのコアにおいて，同じアドレスに配列 value が配置されることを利用している．41 行目で同期をとり，DMA 転送が完了したことを保証し，次のステップに移る．すべてのステップの終了後に，Node(1, 1) のコアが 49 行目でメインループの所要サイクル数を表示する．53 行目の MC_finalize で MClib の後処理を行い，54 行目の return でシミュレーションを終える．

5. SimMc による M-Core の評価

M-Core の評価のため，Equation Solver Kernel, Matrix Multiply，姫野ベンチマークの 3 つのベンチマークプログラムを作成あるいは並列化した．Matrix Multiply は 2 つの行列の積を求めるベンチマークプログラムである．1 つの行列のサイズは 128×128 である．

図 9 は，3 つのベンチマークプログラムの評価結果であり，コア数とシミュレータ上での実行サイクル数の関係を示す．グラフの横軸はコア数，縦軸はコア数 1 のシミュレーション結果で正規化した速度向上比を表す．コア数は，メッシュの形状がなるべく正方形に近くなるように調整した．グラフから，Equation Solver Kernel については，実行するコア数の増加に伴い線形に近い割合で性能が向上しているが，一方で Matrix Multiply については，多コア実行時での性能向上が飽和することが確認できる．これは，複数コアでのデータの配置方法の違いに起因している．Matrix Multiply は Equation Solver Kernel 同様，行列を均等に配分し各ノードに担当領域として割り当てることで処理を並列化する．しかし，両者のプログラムはデータの格納方法に大きな違いがある．Equation Solver Kernel では，処理するデータが各ノードに格納されている．一方，Matrix Multiply では，マスタノードを 1 つ定め，マスタノードが全てのデータを所有する．マスタノード以外のノードは，DMA を使用してマスタノードからデータを取得する．そして各自担当する処理を行い，処理の完了後にマスタノードにデータを転送する．その際，複数ノードから 1 つのノードに対して DMA が集中し，逐次処理が多発することで並列化効率が低下しているものと考えられる．このように，多数のベンチマークを用いた M-Core アーキテクチャの初期評価結果から，効率的なアーキテクチャであることを確認した．

6. おわりに

プロセッサの 1 チップに集積するコア数が増え続けており，マルチコア時代から，メニーコア時代へと向かいつつある．それを踏まえ，我々は，次世代プロセッサアーキテクチャとなるメニーコアアーキテクチャモデルを提案している．その実現に向けた第一段階にあたる M-Core アーキテクチャVer.1.0 を定義し，その評価を行うため，SimMc Ver.1.0 を開発した．さらに，MClib を導入，SimMc 上でシミュレーションするアプリケーション開発の促進を図った．

今後の課題について述べる．メモリ管理機構を導入し，ノードメモリの容量を数百 KB 程度に減らすことに加え，ポート数の削減についても検討する必要がある．シミュレータについては，現在の C++での実装では処理速度の問題から，大規模アプリケーションのシミュレーションとなると現実的な時間で処理できない．例えば，8×8 の 64 コア時のシミュレーションにおいて，シミュレーション速度は 1 秒間に約 40Kcycle と遅い．そのため，FPGA に実装することによる超高速化を検討している．以上の課題を筆頭に，シンプルで効率的なメニーコアアーキテクチャの実装に向け，さらなる検討・開発を進める予定である．

謝　辞

本研究の一部は，科学技術振興機構・戦略的創造研究推進事業 (CREST) 「アーキテクチャと形式的検証の協調による超ディペンダブル VLSI」の支援による．ルータに関して適切な御指導を賜わりました，電気通信大学の吉永努先生，慶応義塾大学の松谷宏紀博士に感謝いたします．

参考文献

1) R. Mullins, A. West and S. Moore: Low-Latency virtual-channel routers for on-chip networks, *In ASP-DAC '06*, pp. 164–169 (2006).
2) 藤枝直輝, 渡邉伸平, 吉瀬謙二: SimMips: 教育・研究に有用な Linux が動く 5000 行の MIPS システムシミュレータ, コンピュータシステム・シンポジウム (ComSys2008) 論文集 (2008).