マルチコア時代の並列プログラミング

著者	若谷 彰良
雑誌名	甲南大学紀要.知能情報学編
巻	1
号	2
ページ	223-247
発行年	2008-12-20
URL	http://doi.org/10.14990/00001273

解説論文

マルチコア時代の並列プログラミング

若谷彰良
甲南大学　知能情報学部
神戸市東灘区岡本 8-9-1, 658-8501

(受理日 2008 年 11 月 3 日)

概 要

マルチコアアーキテクチャを持つシステムに対する 3 つの並列プログラミングの手法について述べる．まず，POSIX Thread は共有メモリ環境をもつ並列コンピュータ上で広く使われるマルチスレッドの仕様であり，様々な実装がある汎用のプログラミング環境である．CUDA(Compute Unified Device Architecture) 環境は GPU(Graphic Processing Unit) をマルチコアのコプロセッサとして並列処理を行うためのプログラミング環境である。また，Cell B.E. (Cell Broadband Engine) をヘテロジニアスマルチコアの代表例としてとりあげ，そのプログラミングスタイルについて述べる．

キーワード： マルチスレッド，並列処理，GPU，マルチコア

1 はじめに

コンピュータは 1940 年代の ENIAC や EDSAC などの登場に始まり，スィッチング素子としてのトランジスタの利用，IC/LSI の活用・高機能化により目覚しく発展した．すなわち，1971 年の Intel 4004 以来，半導体の微細化が進み，動作周波数の向上などコンピュータの性能改善は猛烈な勢いで進んできた [1]．しかし，消費電力の増大や微細化の困難さにより，近年は動作周波数の改善よりも，空間的な改善，すなわち，複数のエレメントを同時並行的に活用する並列処理の需要が高まって来ている．

並列処理を用いたコンピュータは 1970 年代の Illyac IV のような科学技術計算を中心としたスーパーコンピューティングでの利用に始まり，1990 年代の CM-5 や CRAY T3D 等の最先端コンピュータに続き [2]，近年は IBM の BlueGene [3] などの並列コンピュータが TOP500 の最上位についている．さらに，Google は大規模データベースを運用するために数十万台のパソコンからなるデータセンターで MapReduce という手法で大規模並列分散計算を行っている [4]．また，複数の汎用パソコンを LAN で結合した PC クラスタも MPI (Message Passing Interface) [5] の普及とともに身近なものになっている．

一方，単一プロセッサ内のマイクロアーキテクチャにも広く深く並列処理は利用されている [6]．例えば，命令実行行うフローを複数の段に分け，それらを命令毎に同時並行して処理する実行パイプラインや，複数の演算器を用意し，依存関係の無い複数の命令発行を同時に行うスーパースケーラなどは，多くのマイクロプロセッサに実装され，それらを効率的に行うための分岐予測や投機的実行などは多くの研究成果がある．近年は，Intel Core 2 や Sparc Niagara など，命令デコードを含む命令実行エレメント（プロセシングコアもしくはコアと呼ぶ）を複数備えたマルチコアプロセッサが普及し，

実用的に利用されている．マルチコアプロセッサには，全てのコアが同一アーキテクチャであるホモジニアスマルチコア (Homogeneous) と，異なるアーキテクチャのコアを含むヘテロジニアスマルチコア (Heterogeneous) があり，Intel Core 2 や Sparc Niagara など多くのマルチコアプロセッサは前者で，Cell B.E. (Cell Broadband Engine) [7] は後者のアプローチである．GPU (Graphic Processing Unit) を汎用処理に用いる，いわゆる GPGPU (General-Purpose Computing on GPU) も後者のアプローチに含まれる [8].

並列処理におけるプログラミングスタイルには大きく分けて，1) 並列処理用言語，2) 並列化コンパイラ，3) 並列処理ライブラリの３つのアプローチがある．第 1 のアプローチの例としては，Transputer という並列コンピュータ向けに開発された Occam がある [9]．通信と演算をチャネルでつないでプログラムしていく手続き型言語であり，Transputer との親和性は高い．また，既存の言語に並列化に必要な構文を付加した並列処理言語を用いる例として，Fortran を元にした Co-Array Fortran がある．これは，SPMD (Single Program Multiple Data) を実現するために，通常の他のプロセッサが保持するデータを参照するための構文があり，プロセッサ間の通信を文法上，隠蔽することができる．しかし，いずれも普及している言語との文法の違いがあり，修得するまでの敷居が高いという難点がある．第 2 のアプローチはコンパイル以外にユーザすべきことが特にないので，導入しやすい．その代表としては，ベクトルコンピュータのためのベクトル化コンパイラがあげられる [11]．ベクトルコンピュータは依存関係の無い配列演算を高速に行うものであるが，ベクトル化コンパイラが既存のプログラム (主に FORTRAN) のプログラムの中からベクトル計算に向いた配列演算（DO ループ）を見つけだし，場合によれば，ベクトル化可能なようにプログラムを再構成するものである．ベクトル化コンパイラはベクトルコンピュータの普及に多いに貢献したが，ベクトル化可能なのは科学技術計算プログラムが中心であり，全てのタイプのプログラムで効果がある分けではない．また，ベクトルコンピュータ以外で，並列化コンパイラが有効に働くケースは必ずしも多くない．

現在では第 3 のアプローチで並列処理に取り組むケースが一番多いと考えられる．第 3 のアプローチの代表的なものに，MPI のライブラリを用いた分散プログラミング，POSIX Thread [12] を用いたマルチスレッドプログラミング，OpenMP [13] を用いたマルチスレッドプログラミングなどがある．いずれも，既存の言語 (C や FORTRAN) で書いたプログラムにライブラリコールやディレクティブやプラグマを追加することにより並列処理を記述するので，ユーザの負担が全く無いわけではないが，Occam のように言語をゼロから修得するほどのコストは不要である．

MPI は，分散メモリ並列コンピューティング環境における標準的なライブラリで，プロセス間の通信を中心としたものである．広く普及している実装として MPICH [14] があり，PC クラスタなどで利用されている．また，OpenMPI [15] は FT-MPI と LA-MPI と LAM/MPI などの実装をを統合したオープンソフトウェアとして開発されている．MPI は基本的に分散メモリコンピューティング環境向けであるので，共有メモリ形式をとることが多いマルチコア環境では使われることは少ない．また，OpenMP では既存の言語にディレクティブやプラグマを追加し，並列化すべき部分をコンパイラに伝え，マルチスレッドプログラムを生成するものである．OpenMP と並列化コンパイラとの違いは，並列化する部分を検出し指示するのはユーザであり，コンパイラが自動的に並列部分を検出しない点である．また，プラグマもしくはディレクティブで記述するので慎重にコーディングすれば，逐次用と並列用の両方が単一のプログラムで構成できるメリットがあるが，性能向上などためにスレッドを細かくしすぎない工夫も必要である．一方，POSIX Thread は POSIX (IEEE 1003.1) で規定された標準に基づいたマルチスレッドに関する API であり，共有メモリ環境での利用を前提にしている．OpenMP

に比べ，追加記述すべきことやプログラムの書き換えが多くなるが，細かくスレッドを制御でき，高い性能を期待できるので，本論文では汎用のプログラミング環境として POSIX Thread をとりあげる．

以下，本論文の構成を示す．第 2 章では汎用プログラミング環境の代表として，POSIX Thread によるマルチスレッドプログラミングを述べる．第 3 章では大規模マルチコアシステムとして GPU を利用する CUDA (Compute Unified Device Architecture) 環境 [16] についてその概要と，ヘテロジニアスマルチコアアーキテクチャの専用プログラミング環境の代表例として，Cell B.E. でのマルチスレッドプログラミングと実行例を示し，第 4 章でまとめを述べる．

2 汎用プログラミング環境

マルチコアアーキテクチャでは共有メモリモデルを仮定する場合が多いので，そのような環境ではマルチプロセスを用いたプログラミングもしくはマルチスレッドを用いたプログラミングが利用される．昨今は低スケジューリングコスト及びリソースの有効活用の観点からマルチスレッドプログラミングが用いられることが多い．汎用プログラミング環境の代表として，特定のハードウェアに依存しないマルチスレッドプログラミングとしては，POSIX Thread，OpenMP，Win32API, JAVA 言語の Thread クラスなどがあるが，本章では，その代表として，POSIX Thread によるマルチスレッドプログラミングを述べる．

2.1 マルチスレッドとは

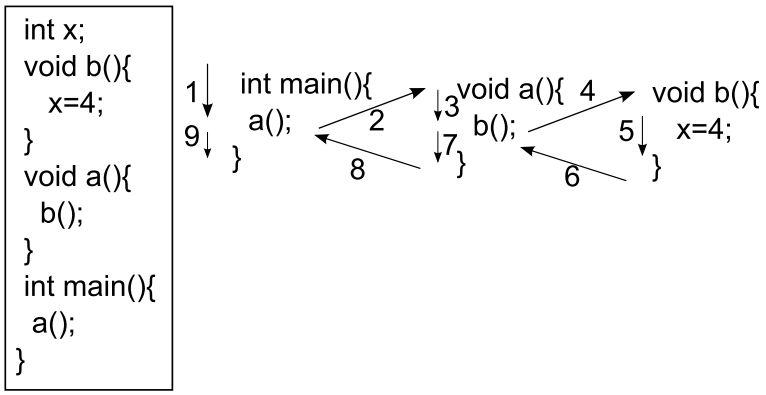


図 1: シングルスレッドでのプログラム呼出し

コンピュータにおける処理は一連の動作を機械語命令で構成し，それをメモリに格納し，順次メモリから読みだして実行するというノイマン方式がとられることが一般的である．この機械語命令はプログラミング言語で書かれたプログラムをコンパイルすることによって生成され，必要なライブラリなどとリンクされたものを実行プログラムとし，ハードディスクなどにファイルとして保持されている．プログラムの実行は，格納されている実行プログラムをメモリ上に配置し，その機械語命令を実行することにより行われるが，この実行単位をプロセスと呼ぶ．

プログラム中の main 関数から関数 a が呼ばれ，さらに，その中から関数 b が呼ばれている様子を図 1 に示す．

プロセスの実行においては，まずメモリ上にプログラムとデータ (この場合は変数 x の領域) を配置し，その後は，プログラムに示した順 (1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9) に機械語命令を逐次に実行する．したがって，関数呼び出しもその順で行われることになる．

しかし，アプリケーションの挙動の中には，逐次に実行する機械語命令を複数用意して，それらを協調させて実行したい場合がある．例えば，動画像の表示をしながら，キーボードやマウスからの入力を待ち，入力があればその処理を行う場合などである．そこで，複数の機械語命令の列の同時実行を許す方式，すなわち，マルチスレッド方式が用いられる．スレッドとは「糸」をあらわし，個別の機械語命令の実行列の流れをスレッドと呼ぶ．なお，図 1 の実行方式は実行の流れが単一であるで、これをシングルスレッドと呼び，区別している．

マルチスレッドでは，同時並行的に実行されるスレッドを動的にもしくは静的に生成し，必要に応じて，スレッド間で同期させて実行を行う．同時並行的とは，複数のハードウェアがあれば同時に実行されるが，単一ハードウェアでは時分割で交互に実行されることを表す．一般に，同じプロセス内のスレッドはメモリ空間を共有しているので，ヒープや大域変数は共有化される．すなわち，あるスレッドで書き込まれた大域変数の値は，異なるスレッドで観測可能である．しかし，スレッド毎にスタックメモリを持つので，スタック上の確保される auto 変数（関数内部で定義された変数) は局所変数となり，スレッド間で共有されない．

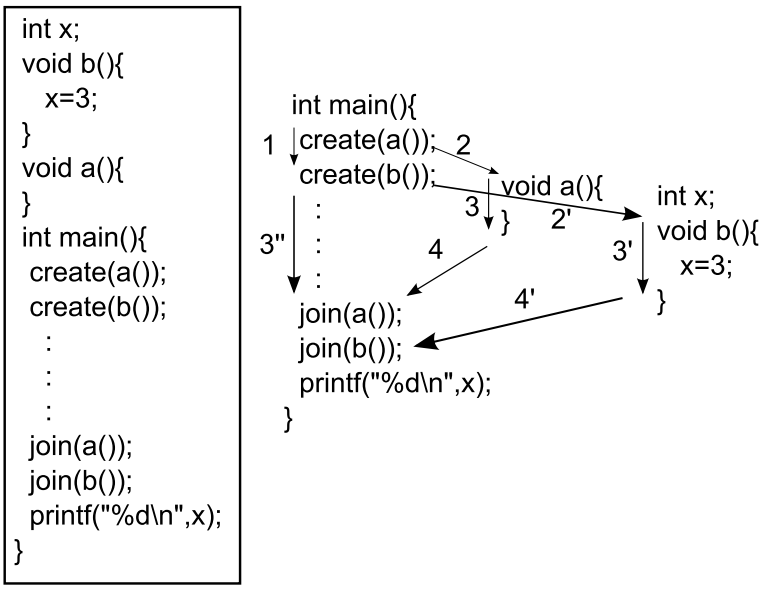


図 2: マルチスレッドでのプログラム呼出し

図 2 にマルチスレッドでの実行例を示す．図 2 では，スレッドを生成する関数 (“create”) とスレッドの実行終了を待つ関数 (“join”) が使用されている．create 関数の引数に，生成されたスレッドが最初に実行すべき関数の名前が指定される．なお，プログラム起動時の最初のスレッドは main 関数が最初に実行すべき関数となる．

この例では，main 関数から新たに 2 個のスレッドが生成され (2 と 2')，それぞれ関数 a と関数 b が実行開始関数に割り当てられる．生成されたスレッドを子スレッドと呼び，生成した側のスレッドを親スレッドと呼ぶ．スレッド実行が開始されると，親スレッドの実行部分 (3") とともに，新たに開始された子スレッドの実行部分 (3 と 3') が同時並行的に実行される．なお，前述の通り，プロセッサが複数ある場合は，同時並行的に実行されるスレッドは別のプロセッサに割り当てられ，実際にも並列実行されることがあるが，単一プロセッサの場合は，複数のスレッドは時分割で実行され，見かけ上，同時並行動作実現されていることになる．また，複数プロセッサの場合でも，プロセッサの数以上にスレッドが生成されている場合は，一部のスレッドは時分割で実行されることになる．さらに，特定のスレッドを特定のプロセッサに対応付けることも可能である．

また，関数 b で変数 x に値が書き込まれているが，その関数を実行しているスレッドおよび関数 a を実行している子スレッドは，親スレッド内の join 関数で，終了を待機している．すなわち，それぞれの子スレッドの実行が完了するまで，親スレッド join 関数はリターンしないので，親スレッドと子スレッドが join 関数を用いて同期していることになる．よって，子スレッドが終了した後に親スレッドが printf 関数を実行するので，変数 x の値として 3 を出力することができる．

2.2 POSIX Thread 概要

2.2.1 スレッド生成

本セクション以下では，POSIX Thread 環境の下での具体的なプログラミング手法について述べる．POSIX Thread においてスレッドの生成と終了待機を行う関数は *pthread_create* 関数と *pthread_join* 関数である．

```
#include <pthread.h>
int pthread_create(pthread_t * thread, pthread_attr_t * attr,
                   void * (*start_routine)(void *), void * arg);
```

新しいスレッドは，arg を第 1 引数とする start_routine という関数になる．新しいスレッドは，pthread_exit 関数を呼び出すことによって明示的に終了するか，関数 start_routine から返ることで暗黙的に終了する．引数 attr には，その新しいスレッドに適用するスレッド属性 (スケジューリングポリシーなど) を指定するが，デフォルト値として NULL を利用することができる．この関数が成功すると，新しく作成したスレッドの識別子が引数 thread の指す領域へ格納される．

```
#include <pthread.h>
int pthread_join(pthread_t th, void **thread_return);
```

pthread_join 関数は，呼び出しスレッドの実行を停止し，th で指定したスレッドが pthread_exit 関数を呼び出して終了するか，取り消しされて終了するのを待つ．thread_return が NULL でないときには，th の返り値が thread_return で指し示される領域に格納される．

簡単なプログラム例は次のようになる．なお，gcc などでコンパイルするときは，“pthread” をオプションにつける必要がある．

```
#include <pthread.h>
void sub(int *x){
  printf("hello! [%d\n]",*x);
}
int main(){
  int i,id[4];
  pthread_t th[4];
  for(i=0;i<4;i++){
    id[i]=i;
  }
  for(i=0;i<4;i++){
      pthread_create(&th[i], NULL, (void *)sub, (void *)&id[i]);
  }
  for(i=0;i<4;i++){
      pthread_join(th[i], NULL);
  }
}
```

上記のプログラムでは，main 関数から 4 個の子スレッドを生成し，関数 sub を起動させる．その際，引数に，変数 i の値を書き込んだ配列 id の要素 id[i] を渡し，子スレッドの sub 関数で，その値を printf するようにしている．

なお，上記のプログラムの pthread_create 関数を用いる部分を

```
pthread_create(&th[i], NULL, (void *)sub, (void *)&i);
```

としても同じ動作をしそうであるが，子スレッドに渡るのは変数 i のポインタであるので，タイミングによってはうまくいかない場合がある．

2.2.2 スレッドへ渡す引数

前章で述べたように，スレッドの生成には pthread_create 関数を用いるが，起動する関数の引数を pthread_create 関数の 4 番目の引数に指定できる．しかし，指定できる引数は 1 個だけであり，通常の関数起動として数が不足する．そこで，子スレッドの起動に用いる関数の引数は複数の変数をまとめた構造体を利用する．また，引数で必要なデータを渡すだけでなく，大域変数も有効に利用することになる．

例えば，配列要素の総和を 4 個のスレッドに分割し，部分的な総和を子スレッドで計算した後に，親スレッドが部分和の総和を計算する場合を下記に示す．

```
#include <pthread.h>
int x[100]; // 総和計算の対象となる配列
int ps[4];  // 子スレッドで計算した部分和を格納する配列
typedef struct _parg_t{
```

```
  int id;       // スレッドを区別する番号
  int bgn,end;  // 部分和を計算する区間の下限と上限
} parg_t;

void ssum(parg_t *arg){
   int i,ibgn,iend,id,s;
   id =arg->id;
   ibgn=arg->bgn;
   iend=arg->end;
   s=0;
   for(i=ibgn;i< iend;i++)
     s+=x[i]; //部分和の計算
   ps[id]=s;
}

int main(){
  int i,total=0;
  pthread_t th[4];
  parg_t arg[4];
  for(i=0;i< 100;i++){// 配列 x の初期化
    x[i]=i;
  }
  for(i=0;i< 4;i++){// 生成するスレッドへの引数の設定
    arg[i].id =i;
    arg[i].bgn=(100/4)*i;
    arg[i].end=(100/4)*(i+1);
  }
  for(i=0;i< 4;i++){
      pthread_create(&th[i], NULL, (void *)ssum, (void *)&arg[i]);
  }
  for(i=0;i< 4;i++){
      pthread_join(th[i], NULL);
  }
  for(i=0;i< 4;i++)// 部分和から総和の計算
    total+=ps[i];
  printf("total: %d [%d->%d]\n",total,0,99);
}
```

上記の例では，子スレッドの開始関数 (sub) に渡す引数として arg_t という型を定義し，子スレッドが実行する際に必要となる，1) 自分のスレッド番号，2) 部分和計算の範囲の下限，3) 部分和計算の範囲の上限，を構造体で定義している．このような構造体は生成するスレッドによって，適切に記述

される必要がある．また，部分和を計算するための配列や，計算結果の部分和自体は，大域変数 (x, ps) で参照されている．

2.2.3 排他制御

前章のプログラム例で変数 total の計算は親スレッドが行っていたが，total 変数を大域変数とし，その更新を子スレッドに行わせることもできると考えられる．

```
#include <pthread.h>
int x[100]; // 総和計算の対象となる配列
int ps[4];  // 子スレッドで計算した部分和を格納する配列
int total=0;
typedef struct _parg_t{
  int id;       // スレッド番号
  int bgn,end;  // 部分和を計算する下限と上限
} parg_t;

void ssum(parg_t *arg){
   int i,ibgn,iend,id,s;
   id =arg->id;
   ibgn=arg->bgn;
   iend=arg->end;
   s=0;
   for(i=ibgn;i< iend;i++)
     s+=x[i];
   ps[id]=s;
   total+=ps[id];
}

int main(){
  int i;
  pthread_t th[4];
  parg_t arg[4];
  for(i=0;i< 100;i++){// 配列 x の初期化
    x[i]=i;
  }
  for(i=0;i< 4;i++){// 生成するスレッドへの引数の設定
    arg[i].id =i;
    arg[i].bgn=(100/4)*i;
    arg[i].end=(100/4)*(i+1);
  }
```

```
  for(i=0;i< 4;i++){
      pthread_create(&th[i], NULL, (void *)ssum, (void *)&arg[i]);
  }
  for(i=0;i< 4;i++){
      pthread_join(th[i], NULL);
  }
  printf("total: %d [%d->%d]\n",total,0,99);
}
```

しかし，このプログラムは正しく動作しない可能性がある[1]．それは各スレッドで実行する，

```
total+=ps[id];
```

の部分に原因がある．この部分の動作は， 1) 大域変数 total を読み出し，2) その値に ps[id] を加算し，3) total に格納する，となる．しかし，この処理はスレッド間で非同期に行われるので正しく加算できない場合がある．例えば，

1. total=0 とする
2. スレッド 1 が total を読み出す (total=0)
3. スレッド 1 が ps[1] (=100) を加算する (total=100)
4. スレッド 1 が total を格納する (total=100)
5. スレッド 2 が total を読み出す (total=100)
6. スレッド 2 が ps[2] (=200) を加算する (total=300)
7. スレッド 2 が total を格納する (total=300)

の順で実行された場合は正しい結果になるが，

1. total=0 とする
2. スレッド 1 が total を読み出す (total=0)
3. スレッド 1 が ps[1] (=100) を加算する (total=100)
4. スレッド 2 が total を読み出す (total=0)
5. スレッド 1 が total を格納する (total=100)
6. スレッド 2 が ps[2] (=200) を加算する (total=200)
7. スレッド 2 が total を格納する (total=200)

[1]時分割している環境では正しく動く場合もある．

の順に実行されると，total の値は不正なものとなる．

このような不正な動作を避けるために共有変数の更新が複数スレッドで同時に行われないようにスレッド間で同期をとって実行するメカニズムを排他制御と呼び，その一つの手段が排他制御変数 (mutex) を用いた， lock 関数，unlock 関数によるものである．すなわち，他のスレッドに割り込まれることなく排他的に実行したい部分の前に lock 関数で他のスレッドの侵入を禁止状態にし，その部分が終了したときに unlock 関数で解放を行う．このときにロックしている領域を識別するための変数が排他制御変数である．このメカニズムによるプログラム例を下記に示す.

```
#include <pthread.h>
int x[100]; // 総和計算の対象となる配列
int ps[4];  // 子スレッドで計算した部分和を格納する配列
int total=0;
typedef struct _parg_t{
  int id;       // スレッド番号
  int bgn,end;  // 部分和を計算する下限と上限
} parg_t;
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
  // 排他制御変数の宣言 (初期化付き)

void sub(parg_t *arg){
   int i,ibgn,iend,id,s;
   id =arg->id;
   ibgn=arg->bgn;
   iend=arg->end;
   s=0;
   for(i=ibgn;i< iend;i++)
     s+=x[i];
   ps[id]=s;
   pthread_mutex_lock(&mutex); // 排他制御区間の始まり
   total+=ps[id];
   pthread_mutex_unlock(&mutex); // 排他制御区間の終わり
   printf("id=%d: %d [%d->%d]\n", id, ps[id],bgn,end-1);
}

int main(){
 /*
 ここの部分は先のプログラムと同じ
 */
}
```

上に示すように，total+=ps[id]; の文は，mutex の lock 関数と unlock 関数に囲まれているので，その区間に入れるスレッドは 1 つのみとなる．例えばスレッド 1 が lock 関数により，その区間に入り，

スレッド 1 が unlock 関数から戻るまでは，他のスレッドはその区間に入れない．したがって，前述したような不正な動作を避けることができる．pthread における排他制御に関係する機構としては，他にバリア同期や条件変数などがあるが，詳細は割愛する．

2.3 実行例

前節で紹介した総和計算プログラムを POSIX Thread で並列化した場合の実行例を示す．実験では Intel Xeon を 2 個備えたシステムを用いる．仕様を表 1 にまとめる．

表 1: 実験環境

CPU	プロセッサ数	プロセッサ当たりコア数	メモリ
Xeon E5335 (2.0GHz)	2	4	4GB

システムとしての総コア数は 8 であるので，最大のスピードアップは 8 となる．総和をとる配列のサイズを変えて，スレッド数を 1 から 8 に変えた場合のスピードアップを図 3 に示す.

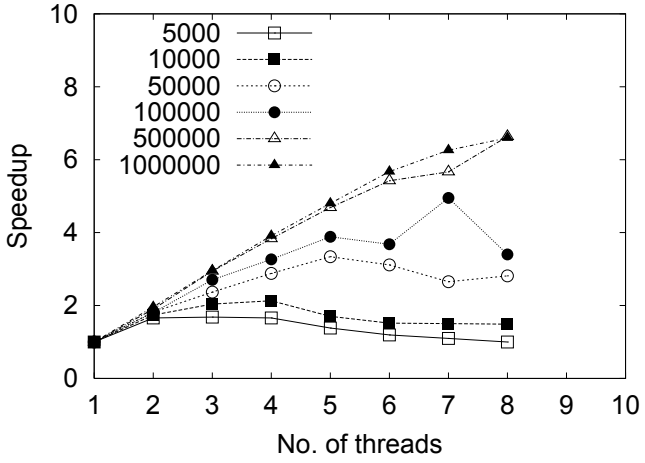


図 3: Xeon 上での実験結果

サイズが小さい場合，スレッド数を増やしてもスピードアップはあまり増加しないが，サイズを増やすにつれて並列効果は増加し，サイズが 1000000 の場合スレッド数を 8 とすると 6 を超えるスピードアップを得られた．サイズが小さい場合はスレッドを生成するコストや排他制御のコストが並列効果を低下させるが，大きい場合は生成コストなどが相対的に小さくなるので並列効果が高まる．

3 専用プログラミング環境

3.1 CUDA

3.1.1 背景

近年の GPU は高性能化が進み，汎用プロセッサの演算性能を大きく超えている．例えば，3.2GHz の Intel Harpertown (Xeon) の Linpack Peak が 102GLOPS であるのに対し，NVIDIA GeForce GTX 200 GPU のピーク性能はほぼ 1TFLOPS (=1000 GFLOPS) に達している．従来の GPU は専用ハードウエアで構成されていたので，グラフィック処理・画像処理に特化され，数値計算やデータベース処理などの汎用アプリケーションには用いられなかったが，NVIDIA の GeForce8 シリーズなどの最近の GPU は unified shader で構成されているので，CUDA [16] や Brook [17] などにより，汎用プログラミング言語で GPU をプログラムできるようになっている．また，CUDA の上で構築された GPULib [18] は，高級言語に近いライブラリであり，GPU を利用することのメリットを事前に判断し，エミュレーションを併用したシステムである．

しかし，CUDA を用いたとしても GPU はグラフィックに特化したメモリ構成をもっているので，メモリモデルに考慮したアプリケーション作成を要求され，高性能なアプリケーションを実現することは簡単ではない．例えば，CUDA では，register，local memory，shared memory，global memory，constant memory，texture memory などのメモリがあり，サイズおよびレイテンシなどが異なり，適切なメモリを選択するための考慮が必要である．

3.1.2 概要

CUDA は GPU 上で汎用プログラムを作成・実行するためのフレームワークであり，C 言語に対する API で構成される．対象とするハードウェアは GeForce 8 以降の NVIDIA の GPU である．

GPU が読み書きするメモリは，global memory，constant memory，texture memory，shared memory，local memory の 5 種類ある．また，8 個の SP (Streaming Processor) を含む MP (Multi Processor) が演算実行を行う．演算はスレッド単位で行われ，それぞれのスレッドは local memory を読み書きし，複数のスレッドからなるスレッドグループは shared memory を共有する．それ以外のメモリは全てのスレッドから読み書きでき，CPU と GPU は global memory を共有する．アクセスの速度は local memory と shared memory は高速であるが，それら以外は低速であり，global memory はキャッシュされない．したがって，並列実行の際は，shared memory を有効に使用することがキーとなるが，容量が小さい (16 KByte) ので，慎重にプログラムすることが必要となる．

GPU 実行の大まかな流れは以下のようになる．

1. CPU の主記憶から GPU の global memory にデータの転送
2. GPU 実行の起動
3. GPU 実行の終了確認
4. GPU の global memory から CPU の主記憶にデータの転送

下に簡単なプログラム例を示す.

```
#include <stdio.h>
#include <cutil.h>

__global__ void cuda002Kernel( float* g_idata, float* g_odata)
{
    const unsigned int tid = threadIdx.x; // スレッドIDを取得
    g_odata[tid] = g_idata[tid]; //グローバルメモリ上でコピー
}

int main( int argc, char** argv)
{
    CUT_DEVICE_INIT();  //デバイスの初期化

    //メインメモリ上にfloat型のデータを100個を2組生成する
    float* h_idata = (float*) malloc(sizeof( float) * 100);
    float* h_odata = (float*) malloc(sizeof( float) * 100);
    for( int i = 0; i < 100; i++) {
        h_idata[i] = i;
    }

    //GPUカードにも同じくfloat型100個分のメモリを確保する
    float* d_idata, d_odata;
    cudaMalloc( (void**) &d_idata, sizeof( float) * 100 );
    cudaMalloc( (void**) &d_odata, sizeof( float) * 100);

    //メインメモリからGPUカードのメモリにデータを転送する
    cudaMemcpy( d_idata, h_idata, sizeof( float) * 100 ,
                cudaMemcpyHostToDevice);

    //ここでGPUを使った計算が行われる
    dim3  grid( 1, 1, 1);      //(1,1,1)個のグリッド
    dim3  threads(100, 1, 1);  //100スレッド
    cuda002Kernel<<< grid, threads>>>( d_idata, d_odata);

    //GPUカードからメインメモリ上に実行結果をコピー
    cudaMemcpy( h_odata, d_odata, sizeof( float) * 100,
                cudaMemcpyDeviceToHost);
```

```
    CUT_EXIT(argc, argv);  //終了処理
}
```

ホストCPU上に，h_idataとh_odataの配列をmallocし，h_idataの配列を初期化する．また，GPUカード上のメモリにd_idataとd_odataの配列を確保する．実行は，1個のグリッド上で100個のスレッドを生成して実行する．実行するプログラムは，cuda002Kernel関数で，自分のスレッド番号をインデックス(tid)としたデータのコピー(g_odata[tid] = g_idata[tid];)を行うものであり，この部分が並列実行となる．cuda002Kernelをコールするところで“<A,B>”と記述されるのは実行するスレッド形態を表すものである．すなわち，1個のマルチプロセッサ上((1,1,1)のグリッド形態)で100個のスレッドを起動して実行することが記述されている．ホストCPUとGPUカードのメモリ間の転送はcudaMemcpy関数で行い，最後の引数で転送の方向を示す.

3.1.3 実行例

表2: NVIDIA 8600 GTの仕様

no. of MP	Shader clk.	Core clk.
4	1180MHz	540MHz

本章ではベクトル量子化圧縮(VQ)のコードブック生成法の一つであるPNN (Pairwise Nearest Neighbor)法[19], [20]に注目し，その主要ループ部分をCUDAで実装し，NVIDIA GeForce 8600 GT上で評価する．インターネットの普及とともに，ネットワーク上を流れるデータは膨大になり，その圧縮は重要な技術の一つである[21]-[23]．VQは様々なアプリケーションに利用されている手法であるが，その実行時間の短縮は重要である．VQは，コードブック生成，圧縮，伸長からなるが，その中でもコードブック生成は実行コストが大きい．したがって，GPUを用いた高速化の可能性を検証することは必要である．

今回の実験で用いるGPUはNVIDIA 8600 GTである．仕様を表2にまとめる．なお，8600 GTよりもハイエンドなGPUは8600 GTよりも高い性能を達成することが予想される．

8600 GTには，MP (Multi Processor)が4個であり，MP内に8個のSPがあるので，最大32並列の並列度が得られる．最大で3演算が1クロックで実行できるので，理論上の最大性能は51.8 GFlops($=32\times3\times540$)となる．また，shader clockはcore clockよりも遅いので，一つの命令に対して複数のスレッドをパイプライ的に実行することが効率的な実行のためのキーとなる．今回のハードウェアに対しては，4サイクルのパイプライン実行が必要であり，したがって，MPあたり32スレッドの実行が最小限必要である（これをWarpと呼ぶ）．よって，MPの個数倍の128スレッド以上で実行することが望ましい.

ホストPCでの実行時間をベースとしたGPUでの実行時間のスピードアップを図8に示す．ホストPCのCPUはIntel Core 2 Duo (E4500, 2.2GHz)であり，メモリは512Mbyte，OSはWindows XP SP3である．また，使用したコンパイラは，nvcc (release 1.1)である．PNNアルゴリズムのアウトラインを図4に示す.

今回の実行例では，2 つの方法で実装した．第 1 の方法では，図 4 の j のループをスレッドに分散した．よって，i のループはスレッド外で，k のループはスレッド内で逐次に実行する．distance 関数は j 番目のベクトルと k 番目のベクトルの重み付きユークリッド距離を計算する関数であり，本プログラムの中心となる計算である．しかし，ベクトル全体のサイズはかなり大きい．j 番目のベクトルはスレッドで固定であるので，local memory にコピーして利用するが， k 番目のベクトルは distance 関数を起動する毎に global memory からロードするので，そのためのコストは高いと考えられる．

```
for(i=T;i>K;i- -){
    jmin=kmin=-1; xmin=10000000;
    for(j=0;j<i;j++)
        for(k=j+1;k<i;k++){
            x=ditance(j,k);
            if(x<xmin){
                xmin=x;
                jmin=j;kmin=k;
            }
        }
    merge(jmin,kmin);
}
```

図 4: PNN 法

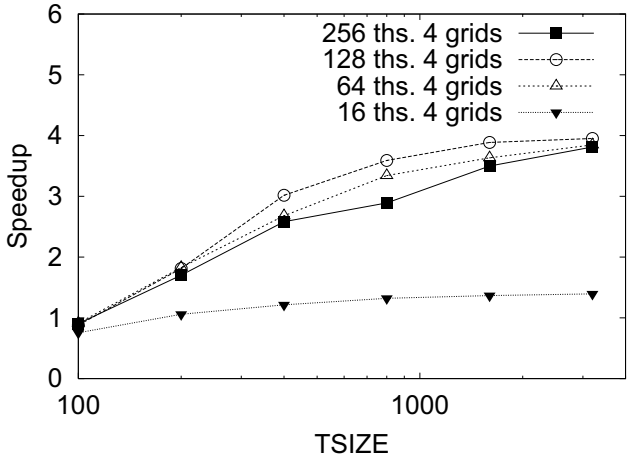


図 5: 8600 GT 上での実験結果

図 5 に第 1 の方法での実行結果を示す．grid サイズは CUDA でのスレッドグループの個数を表し，8600 GT には 4 個の MP があるので，grid を 4 以上にとることが望ましい (図 5 には grid 数 4 の場合を示す)．横軸は初期トレーニングベクトルのサイズ (T) であり，縦軸は CPU での実行時間をベースとしたスピードアップを示し，各 grid のスレッド数を 16, 64, 128, 256 の場合の結果を表す．初期ベ

クトルのサイズを大きくするにつれて，スピードアップが増加していることが分かる．前述の通り，GPU の MP を効率よく実行するには，MP あたり 128 スレッド以上が同時並行して実行される必要があるので，スレッド数が 16 の場合は，高いスピードアップが実現できない．しかし，スレッド数が 64〜256 であっても，4 程度のスピードアップしか達成できなくて，必ずしも高効率とはいえない．その理由は distance 関数で必要とするベクトルを global memory から取り込んでいるので，そのレイテンシにより性能が低下しているからである．よって，global memory を用いたプログラミングは容易であるが，高性能な実行を達成することは難しい．

そこで，distance 関数に読み込むベクトル shared memory に置くため，そのサイズ (16 Kbyte) に fit するサイズに k のループを strip-mining し，k のループを繰り返して実行するように変更した場合の結果を，図 6 に示す．

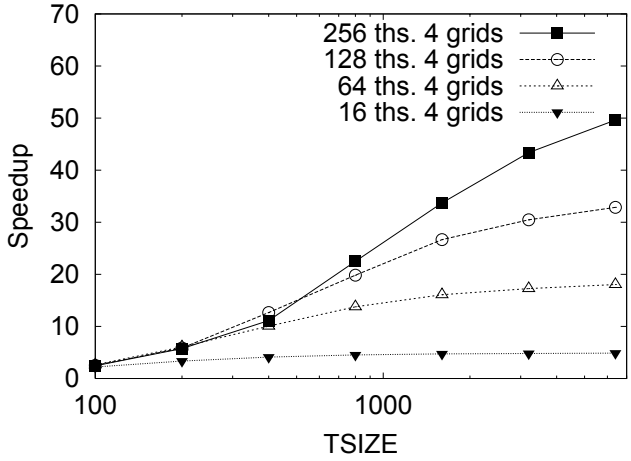


図 6: shared memory を用いた場合の実験結果

図から分かるように，スレッド数を増加することにより，MP の実行効率が改善され，スピードアップが増加しており，スレッド数が 256 の場合 (全体では 1024 スレッド) に最大性能が達成されている．以上のように，shared memory を用いるように改良することで，メモリレイテンシの改善がなされている．

3.2 Cell B.E.

3.2.1 背景

マイクロプロセッサの発展と普及はめざましく，コンピュータやモバイル機器の性能向上の多くは，マイクロプロセッサの性能向上によって達成されていると言っても過言ではない．しかし，近年，マイクロプロセッサの周波数の高速化が以前ほどの伸びておらず，デュアルコアやクアドコアなど，その性能はマルチコア化によってなされている傾向があり，さらにメニーコアへの移行も進んでいる [24]．ゲーム機器のマイクロプロセッサも同様の傾向である．Cell B.E. (Cell Broadband Engine) は，PLAYSTATION 3 (PS3) のために SONY，IBM，東芝が共同で開発したマイクロプロセッサで，ヘテ

ロジニアス型マルチコア構成のアーキテクチャを持つものである [7]．Cell B.E. には Power6 アーキテクチャの PPE (Power Processor Element) が 1 個と，8 個の SPE (Synergistic Processor Element) が載っており，PPE と SPE は異なるアーキテクチャである [25].

マルチコアプロセッサのプログラミング環境も近年整備されつつある．OpenMP は，元々は共有メモリ型並列コンピュータに対する標準プログラミングツールとして策定された仕様であり，C や FORTRAN などの既存の言語にディレクティブやプラグマなどの形式で情報を付加する形で言語を拡張し，並列実行する部分やその方式および変数の共有・複製などを指示するものである．また，Linux OS などで利用可能な POSIX thread は，より細かくスレッドの制御が可能であり，高い性能を必要とする場合に用いられる．しかし，今までのツールは SMP のようなホモジニアスな並列環境で使用されてきたものであり，Cell B.E. のようなヘテロジニアスな環境ではそれらのツールではカバーしきれない部分が存在する．たとえば，PPE と SPE でのプログラム分割・データ分割や，SPE のもつプライベートなメモリ空間の扱いなどである．

本章では，ヘテロジニアスなマルチコア環境として Cell B.E. に注目し，その上での実行例として BLAS ライブラリの SAXPY ルーチンの実装についての評価を行う．SAXPY については，DMA による SPE のもつプライベートなメモリ空間へのデータ転送が大きなオーバーヘッドになると考え，DMA 処理と計算とのオーバーラッピング手法の効果について検討を行う．

3.2.2 PS3 アーキテクチャ

PS3 は 3.02GHz の Cell B.E. をメインプロセッサとし，NVIDIA の RSX をグラフィックプロセッサとして搭載し，256 Mbyte の XDR DRAM をメインメモリとする．前述のとおり，Cell B.E. は，Power6 準拠の PPE と 8 個の SPE からなるヘテロジニアス・マルチコアアーキテクチャである．SPE は直接メインメモリにアクセスすることはできず，256 Kbyte の Local Storage (LS) がアクセスできるメモリとなる．したがって，SPE がメインメモリのデータにアクセスするためには，Memory Flow Controller (MFC) の DMA を用いて，LS にデータを転送し，その上で LS 内のデータにアクセスする必要がある．PS3 には，最大 60 GByte のハードディスクと，ギガビットイーサネット，Blu-ray ディスクドライブ，USB インタフェースを備え，Play Station 2 との互換性もある．

本章では，PS3 上で動作する Linux 環境での実装と評価について議論するが，Linux には RSX のドライバが実装されていないので，RSX へ直接アクセスすることはできない．

3.2.3 基本プログラミング

Cell B.E. でのプログラム実行は，Power 用のバイナリによる PPE での実行が基本となる．PPE には SIMD 演算機構 VMX が実装されているので，それらを陽につかうためには VMX 用 API を用いる必要がある．SPE で実行するためには，PPE から libspe2 に基づく API を利用して，1) 実行モジュールのオープン，2) コンテクストの生成，3) コンテクストのロード，4) コンテクストの実行，5) コンテクストの廃棄，などをコールし SPE を制御する．また，PPE からの SPE のコンテクスト実行はブロッキング操作であるので，PPE の実行と SPE の実行をオーバーラッピングもしくは，複数 SPE の実行を行うには，PPE からの SPE のコンテクスト実行をマルチスレッドで実行する必要があり，POSIX

thread を使用したマルチスレッド実行となる．

SPE がアクセスできるのは LS 内のデータだけなので，PPE が読み書きするメインメモリと LS の間は DMA を用いて，データの移動を陽に記述する必要がある．図 7 に基本的な DMA ルーチンを示す．

```
spu_mfcdma64(&x[0], mfc_ea2h(ea),
  mfc_ea2l(ea),size, tag, MFC_GET_CMD); // (1)
spu_writech(MFC_WrTagMask, 1 << tag);   // (2)
spu_mfcstat(MFC_TAG_UPDATE_ALL);        // (3)
```

図 7: DMA ルーチン

まず，(1) の関数で，“tag” 番のチャネルの DMA に対する実行リクエストを起動する．このとき，MFC_GET_CMD コマンドは LS へ取り込み，MFC_PUT_CMD コマンドは LS からメインメモリへ書き込む動作を行う．(2) のマクロは，“tag” 番のチャネルのマスクをセットし，そのマスクに対して，(3) の関数で終了を確認する．すなわち，(3) の関数は，DMA 動作が終了するまでリターンしない．なお，(1), (2), (3) を連続して書く必要はなく，(1) と (2) の間に別の SPE プログラムを書けば，DMA 動作と SPE 実行が並行処理できる．また，tag の値を変えることにより，複数の DMA 動作を並行して行うことも可能である．

なお，DMA は 1 つのリクエストで 16 Kbyte のデータまで転送できる．よって，16 Kbyte を超えるデータを転送するには，複数のリクエスト発行が必要である．また，DMA は 32 チャネルあるので，同時並行して 32 個の DMA 動作を行うことができる．

3.2.4 プログラム例

Cell B.E. のプログラミングはマスターとなる PPE のプログラムとスレーブとなる SPE のプログラムの２種類を書く必要がある．下記に PPE のプログラム例の概略を示す．

```
void *run_saxpy_spe(void *arg)
{
  /* 必要な変数宣言や初期化は省略 */
  entry = SPE_DEFAULT_ENTRY;
  ret = spe_context_run(arg->spe, &entry, 0,
                        arg->saxpy_params, NULL, &stop_info); //(e)
}

void calc_saxpy()
{
  /* 必要な変数宣言や初期化は省略 */
  prog = spe_image_open(spefile);                              //(a)
  spe  = spe_context_create(0, NULL);                          //(b)
```

```
  spe_program_load(spe, prog);                                  //(c)
  pthread_create(&thread, NULL, run_saxpy_spe, &arg);           //(d)
  pthread_join(thread[i], NULL);                                //(f)
}

int main(){
  /* 必要な変数宣言や初期化は省略 */
  calc_saxpy();
  return 0;
}
```

PPE のプログラムでは SPE で実行する部分以外のプログラムと，SPE での実行を起動・停止確認する部分の両方を書く．上記では主に後者のみを書いてある．SPE の起動方法は前節で述べた通りであるが，まず (a) の spe_image_open 関数で SPE で実行するプログラムのファイル名を指示し，オープンする．次に，(b) の spe_contect_create 関数で SPE 上にコンテクスト生成し，(c) の spe_program_load 関数で SPE のコンテクストにプログラムをロードする．この時点では実行はまだ開始されていない．実行の開始は (e) の spe_context_run 関数で行われるが，この関数は実行が終了するまでリターンされない．したがって，前節で述べたように，複数の SPE を起動する場合には，(d) のように pthread_create 関数でマルチスレッドで起動する必要がある．最後に，(f) で SPE の終了を確認する．

一方，SPE で実行するプログラム例の概略を下に示す．

```
int main(unsigned long long spe, unsigned long long argp)
{
  /* 必要な変数宣言や初期化などは省略 */

  /* DMA Transfer 1 : GET input parameters */
  spu_mfcdma64(&saxpy_params, mfc_ea2h(argp), mfc_ea2l(argp),
               sizeof(saxpy_params_t), tag, MFC_GET_CMD);     // (a)
  spu_writech(MFC_WrTagMask, 1 << tag);                       // (b)
  spu_mfcstat(MFC_TAG_UPDATE_ALL);                            // (c)

  start = saxpy_params.start;
  end = saxpy_params.end;

  /* Calculation */
  for(i=start;i<end;i+=BSIZE){
     .....
  }

  /* DMA Transfer 2 : PUT results */
  spu_mfcdma64(&ly[0], mfc_ea2h(ea), mfc_ea2l(ea),
               sizeof(float)*BSIZE, tag, MFC_PUT_CMD);        // (d)
```

```
    spu_writech(MFC_WrTagMask, 1 << tag);                    // (e)
    spu_mfcstat(MFC_TAG_UPDATE_ALL);                         // (f)

    return 0;
}
```

これも前節で述べたように，SPE では DMA を使って LS にデータを読み書きする必要があるので，(a), (b), (c) の手順でデータを LS に読み込んでいる．まず，(a) で MFC_GET_CMD コマンドでデータ読み込みの DMA で起動をかけ，(b) および (c) の手順でその終了を確認する．その後，SPE での計算を行い，終了後，結果を (d), (e), (f) の手順で LS のデータ PPE のメモリを書き出す．読み込みとの違いは MFC_PUT_CMD コマンドを用いている部分である．

3.2.5 実行例

本章では BLAS level 1 の SAXPY ルーチンに注目し，Cell B.E. 浮動小数の性能を計測し，DMA オーバーラッピング手法の効果を評価する [26]．本章で用いる実験環境を表 3 にまとめる．

表 3: 実験環境

platform	Play Station 3
CPU	Cell B.E. (3.02GHz)
memory	256 Mbyte
HDD	60 Gbyte
OS	Yellow Dog Linux 5.0
libspe	libspe 2.0.1
compiler	ppu/spu-gcc-3.3.72

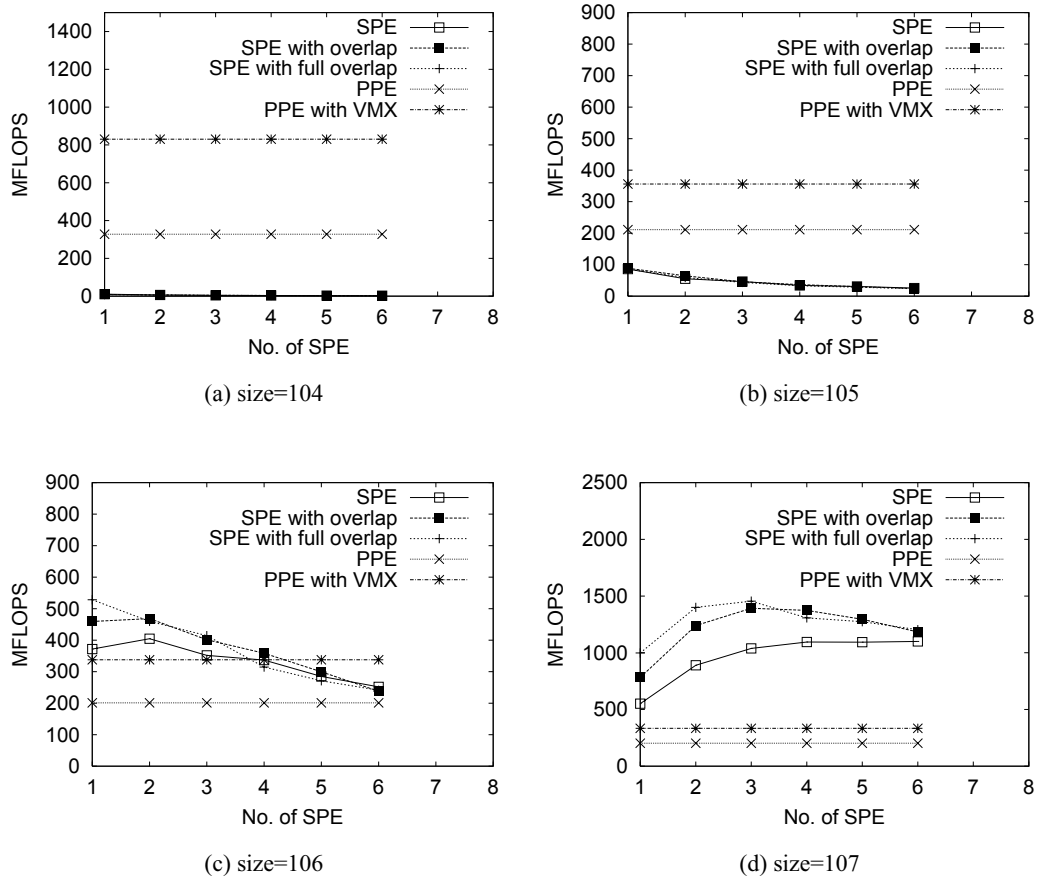
BLAS は線形代数関連のライブラリで，ベクトル計算，ベクトル・行列計算，行列・行列計算が，level 1, 2, 3 に含まれており，level 1 の代表的な SAXPY ルーチンは

$$y_i \quad = \quad \alpha \cdot x_i + y_i \ (0 \leq i < n) \tag{1}$$

を実行する単精度浮動小数点数に対する演算ルーチンである．なお，単精度浮動小数点数だけでなく，倍精度および複素数のルーチンも存在する．単精度浮動小数点数の場合，Cell B.E. では PPE でも SPE でも SIMD 演算で 4 個のデータの同時実行が可能で，それぞれ vec_madd 関数および spu_madd 関数を用いて，SIMD 演算で実行できる．また，各反復において，計算前に x_i と y_i をメインメモリから LS に転送し，計算後に y_i を LS からメインメモリに転送しなければならない．したがって，DMA の処理オーバーヘッドを相対的に軽減するために，DMA 処理の最大の 16 Kbyte 程度のデータ単位で転送することを考え，単精度浮動小数点数が 4 byte であることを考慮し，計算前に 4000 個の x と y を一括した転送し，計算後に計算後に 4000 個の y を一括した転送する．

実行結果を図 8 に示す．グラフの縦軸は実行性能を MFLOPS で表し，横軸は使用する SPE の個数を示す．なお，PPE の実行結果の場合，SPE の個数は関係ない．配列サイズが 10^4，10^5 の場合，配

列サイズに比例した計算時間に比べて， SPE プログラムのオープン，コンテクストの生成，プログラムのロードなどの固定時間が相対的に大きいので，SPE を用いるメリットはなく，PPE だけの計算のほうが性能が高い．配列サイズが 10^4 の場合，PPE は VMX を用いることにより，327 MFLOPS から 830 MFLOPS に性能が向上する．配列サイズが 10^5 の場合，VMX を用いても 355 MFLOPS の性能である．これは 10^4 の場合，必要なデータが PPE の L2 キャッシュ(512 Kbyte) に格納可能であるからである．



(a) size=104 (b) size=105

(c) size=106 (d) size=107

図 8: 結果 (DMA size = 16000 byte)

配列サイズが 10^6 では，SPE1 個および 2 個の場合，PPE よりも高い性能を上げられることが分かる．すなわち，intrinsic を用いた SPE では 370 MFLOPS から 400 MFLOPS を達成し，PPE を上回る．さらに，アンローリングを用いた方法および 完全 オーバーラッピング により DMA オーバーヘッドを軽減した結果は，さらに，高い性能をあげることができる．しかし，SPE の数をさらに増やすと，性能は下がる．これは，SPE の数が増えることにより，メインメモリと LS の間の DMA 転送の数が増え，よってメモリバスが混雑することにより，性能が下がるからである．

配列サイズが 10^7 では，10^6 のときに比べて，より高い台数効果が得られる．すなわち，SPE が 3 個の時に最大の性能が得られる (1454 MFLOPS)．10^6 と同様に，メモリバスの混雑により，性能が飽

和している．

今回の実験の SAXPY ルーチンは DMA によるメモリ I/O 時間に比較して各反復での計算量が相対的に小さいプログラムであるので SPE 数を増やしても性能向上が飽和し，高い並列効果は得られなかった．そこで，計算量を増やした次の演算ルーチンを考え，並列化による潜在的な実行性能を測定することを試みる．

$$\begin{aligned} y_i &= \alpha_1 \cdot x_i + \alpha_1 \cdot y_i + \alpha_2 \cdot x_i + \alpha_2 \cdot y_i \\ &+ \alpha_3 \cdot x_i + \alpha_3 \cdot y_i + \alpha_4 \cdot x_i + \alpha_4 \cdot y_i \\ &+ x_i \cdot y_i \ (0 \le i < n) \end{aligned} \tag{2}$$

通常の SAXPY に比べ，データ転送量は同じであるが，各 iterarion あたりの浮動小数演算数が 2 から 17 に増えている．この演算ルーチンに対するサイズ 10^7 の実行性能を図 9 に示す.

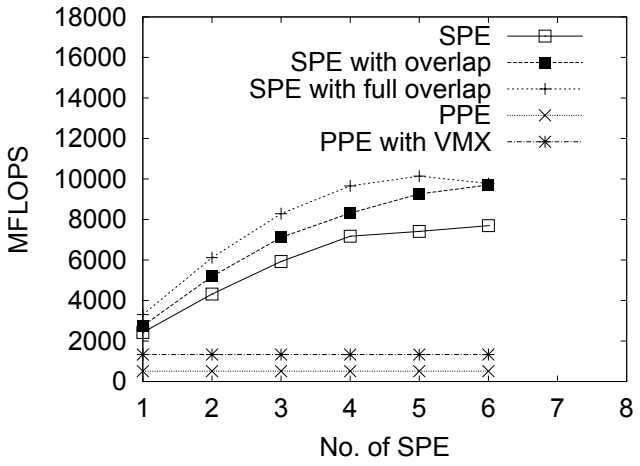


図 9: size=10^7,DMA size = 16000 byte

通常の SAXPY ルーチンに比べ，SPE 数増加による並列効果は向上し，アンローリングを用いる SPE 演算の場合，SPE が 6 台で 9.7 GLOPS を達成し，また，完全 DMA オーバーラッピング をする SPE 演算の場合，SPE が 5 台で 10.1 GLOPS を達成している．よって，BLAS レベル 3 の行列積の SSYMM ルーチンや，回帰計算の P-scheme [27] は計算量が多く，台数効果が高いと考えられ，SPE 使用数を増やして性能を上げることができる演算ルーチンであると考えられる．これを次章で確認する．なお，完全 DMA オーバーラッピング するプログラムは複雑であり，なんらかのプログラミングツールを用いてプログラムを作成することが必要である．

DMA オーバラッピングの効果であるが，SPE1 個の 10^7 サイズの SAXPY の場合，元の性能は 551 MFLOPS である．アンローリングし半分の DMA 処理をオーバラップすると 782 MFLOPS となり，完全 オーバーラッピング すると 998 MFLOPS まで向上し，約 2 倍の性能向上となる．一方，計算量を多くした図 9 の演算ルーチンの場合，元の性能は 2.4 GFLOPS であり，アンローリングし半分の DMA 処理をオーバラップすると 2.8 GFLOPS となり，完全 オーバーラッピング すると 3.3 GFLOPS まで向上し，約 1.4 倍の性能向上となる．以上より，バッファ領域の増加はあるものの，オーバラッピン

グの効果は顕著であり，ヘテロジニアス型マルチコアプロセッサにおいては必須の手法であることが分かる．

4 おわりに

本論文では，近年のマルチコアアーキテクチャの普及と発展を振り返り，その中で活用されている並列プログラミング手法として POSIX Thread，CUDA，Cell B.E. でのプログラミングを取り上げ，その概要について述べた．ヘテロジニアスアーキテクチャに対しては POSIX Thread 以外の手法を用いてプログラミングすることになり，単純な書換えは容易である．しかし，メモリ構成を意識した DMA の効率的な利用や高速な shared memory の有効利用が高性能を達成するキーとなる．

実用的並列システムが登場して以来，約 40 年間に，さまざまな並列プログラミング手法が登場しているが，アプリケーションやプラットフォームを考慮した最適な選択が必要である．しかし，マルチコアアーキテクチャに対する手法の標準化・統一化がなされていないので，プログラミング手法の教育だけでなく並列処理概念の教育が重要であるとともに，汎用的・標準的な手法の確立及びその普及が今後の課題である．

謝辞

本研究の一部は，文部科学省オープン・リサーチ・センター整備事業「知的情報ネットワークによる地域密着型教育の高度情報化に関する研究」(2004-2008) の支援を受けて，実施されました．

参考文献

[1] David A. Patterson, John L. Hennessy, Peter J. Ashenden and James R. Larus, *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Pub., 2004.

[2] David E. Culler, Jaswinder Pal Singh and Anoop Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Pub., 1998.

[3] Blue Gene,
http://www.research.ibm.com/bluegene/

[4] MapReduce: Simplified Data Processing on Large Clusters,
http://labs.google.com/papers/mapreduce.html

[5] Message Passing Interface Forum,
http://www.mpi-forum.org/

[6] David A. Patterson and John L. Hennessy, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Pub., 2006.

[7] Cell Broadband Engine,
http://cell.scei.co.jp/index_e.html

[8] Hubert Nguyen, *GPU Gems 3*, Addison-Wesley Pub., 2007.

[9] Geraint Jones and Michael Goldsmith, *Programming in Occam 2*. Prentice Hall, 1988.

[10] Yuri Dotsenko, Cristian Coarfa and John Mellor-Crummey, "A Multi-platform Co-Array Fortran Compiler," in *Proc. of the 13th International Conference of Parallel Architectures and Compilation Techniques*, pp. 29-40, 2004.

[11] Michael Joseph Wolfe, *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1995.

[12] B. Nichols, D. Buttlar and J. Farrell, *Pthreads Programming: A POSIX Standard for Better Multiprocessing*. O'Reilly Media, 1996.

[13] Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, Ramesh Menom and Rohit Chandra, *Parallel Programming in OpenMP*. Morgan Kaufmann Pub., 2000.

[14] MPICH-A Portable Implementation of MPI,
http://www-unix.mcs.anl.gov/mpi/mpich1/

[15] Open MPI: Open Source High Performance Computing,
http://www.open-mpi.org/

[16] CUDA Zone,
http://www.nvidia.com/object/cuda_home.html

[17] BrookGPU,
http://graphics.stanford.edu/projects/brookgpu/

[18] Peter Messmer, Paul J. Mullowney and Brian E. Granger, "GPULib: GPU Computing in High-Level Languages," *IEEE Computing in Science & Engineering*, vol. 10, no. 5, pp. 70-73, 2008.

[19] W. Equitz, "A New Vector Quantization Clustering Algorithm," *IEEE Trans. on Acoustics, Speech and Signal Processing*, vol. 37, no. 10, pp. 1568-1575, 1980.

[20] A. Gersho and R. Gray, *Vector Quantization and Signal Compression*. Kluwer Academic Pub., 1992.

[21] Jeng-Shyang Pan, Zhe-Ming Lu, and Sheng-He Sun, "An Efficient Encoding Algorithm for Vector Quantization Based on Subvector Technique," *IEEE Trans. on image processing*, vol. 12, no. 3, pp. 265-270, 2003.

[22] R.M. Gray, "Vector Quantization," *IEEE ASSP Magazine*, vol. 1, pp. 4-29, 1984.

[23] Y. Linde, A. Buzo, and R. M. Gray, "An Algorithm for Vector Quantizer Design, " *IEEE Trans. Commun.*, vol. 28, no. 1, pp. 84-95, 1980.

[24] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote and N. Borkar, “An 80-tile 1.28TFLOPS Network-on-chip in 65nm CMOS,” in *Proc. ISSCC2007*, p. 98, 2007.

[25] Jakub Kurzak, Alfredo Buttari, Piotr Luszczek and Jack Dongarra, “The PlayStation 3 for High-Performance Scientific Computing,” *IEEE Computing in Science & Engineering*, vol. 10, no. 3, pp. 84-87, 2008.

[26] BLAS (Basic Linear Algebra Subprograms), http://www.netlib.org/blas/

[27] A. Wakatani, “A Parallel and Scalable Algorithm for ADI Method with Pre-propagation and Message Vectorization,” *Parallel Computing*, vol. 30, no. 12, pp. 1345-1359, 2004.