PCによる並列コンピューティング

大 塚　　登

HPC（High Performance Computing）と呼ばれる高性能コンピュータによる大規模計算の分野では並列コンピューティングが主流となっている。最先端ではスーパーコンピュータが用いられるが、中小規模ではPCが利用されることもある。PCを利用する場合でも用途に応じてさまざまなハードウェア・ソフトウェアが開発されていて、それほどコストをかけずともある程度の規模の計算が可能となっている。本稿では最近のPCによる並列コンピューティングの状況をまとめ、中でも代表的な手法であるMPI、OpenMP、TBB、Cilk、CUDA、OpenCL、OpenACCおよびC++AMPについて具体例を挙げて比較検討する。

キーワード：HPC、並列コンピューティング、MPI、OpenMP、TBB、Cilk
GPGPU、CUDA、OpenCL、OepnACC、OpenHMPP、C++AMP

目　次

1 はじめに

科学技術分野のコンピュータ利用では常に大規模の数値計算を高速に行うことが要求され、HPC（High Performance Computing）と呼ばれる分野において研究が行われている[1]。その最先端においてはスーパーコンピュータが用いられるが、中小規模のレベルではPCが用いられることもある。いずれにおいても、演算の基本単位であるCPUあるいは演算コアそのものの性能を向上させるにはほぼ限界に達していて、複数のCPUあるいはコアを同時並列的に利用することにより、扱う計算規模の拡大と計算時間の短縮を可能とする並列コンピューティングが主流となっている。

最近話題となったスーパーコンピュータ『京』では８個のコアを持つCPUが８万個以上用いられていて、その性能を最大限に引き出すためにプログラミング環境として、後述するマルチプロセス型とマルチスレッド型の並列計算のハイブリッド型のプログラミングモデルが採用されている[2]。

この論文では、最近のPCを用いた並列コンピューティングの技術について概観し、その内容、特徴を具体的な並列計算のプログラムを用いて比較検討し、また今後の方向について議論する。

2 並列コンピューティング

2．1 ハードウェア

汎用のCPUを用いた並列コンピューティングのシステムは大きく分散メモリ型と共有メモリ型に分けられる。分散メモリ型システムは、CPUとそのアクセスできる固有のメモリ空間が複数個分散しているものである。実際の機器構成としては、複数のPCをネットワークで接続したシステム（PCクラスタ）[3]が一般的である。一方、共有メモリ型システムではすべてのCPUが共有メモリにアクセスできる。複数のCPUを持つ１台のPC、複数コアのCPUを持つ１台のPCがその代表的な例である。分散メモリ型システムでは複数のプロセスを、共有メモリ型システムでは複数のスレッドを作成し、それらで計算を並列に実行させるものである。

それとは別に、近年では本来3Dグラフィックス用の目的で開発されてきたGPUを数値計算に用いるGPGPU（General Purpose computing on Graphics Processing Units）が盛んになってきている。その理由は、GPUがリアルタイムの動画表示を可能にするために、画面上の各点で表示するデータを並列に処理することができるよう演算コアを多数搭載しているからである。この多数の演算コアをグラフィックス以外の一般の処理にも利用しようというのである。

２．２　ソフトウェア

分散メモリ型システムと共有メモリ型システムでは当然動作プログラムの内容が異なってくる。分散メモリ型システム用の動作プログラムでは、それぞれの演算単位（CPUあるいはコア）でプロセスを生成し、プロセス間のデータ共有はプロセス間通信によって行う。このようなプログラム作成の枠組みとしてMPI（Message Passing Interface）が広く用いられている[4-7]。共有メモリ型システムの場合は、複数のプロセスで並列化することも可能であるが、メモリを共有する複数のスレッドによって並列化するのが一般的である。このようなプログラム作成の枠組みとしてはOpenMPが広く用いられている[10-13]。

並列処理のタイプには大きく分けて、データ並列とタスク並列がある。データ並列とは、処理対象のデータを分割し、それぞれに対して各実行単位（プロセスまたはスレッド）が同じ処理を並列に行うものである。一般的に処理の記述は比較的簡単であるケースが多いが、各実行単位の扱うデータ間に依存性がある場合は並列実行ができない場合がある。

一方、タスク並列では各プロセスまたはスレッドがそれぞれ異なる処理を並列に行うものである。流れ作業であるパイプライン処理もこれに含まれる。データ並列の場合と比較して処理の記述が複雑となるケースが多い。

利用する機会の多いデータ並列処理の一つにレデュース（Reduce）処理がある。レデュース処理とは複数のデータから一つの値（レデュース値）を得る処理で、総和、最大／最小値などの計算である。分散メモリ型の計算では、各プロセスでのレデュース値を一つのプロセスに集めるための処理が必要であり、共有メモリ型の計算では、各プロセスが共通のレデュース値へアクセスする際の競合に注意が必要となるので、以下に紹介する並列計算システムではレデュース計算のための機能が用意されている。

GPGPUの場合、GPUの構造はメーカーによって異なるためにプログラム作成方法もGPUによって違い、NVIDIA社はCUDA、AMD社はFireStreamというようにメーカーごとに開発環境を提供している。この状態はプログラム開発者にとっては望ましくないためGPUによらないプログラム作成の手段としてOpenCLが制定されている。

２．３　テスト用プログラム

実際に並列コンピューティングを行うには逐次処理プログラムをどのように変更する必要があるか、それによりどれだけパフォーマンスが上がるか、を見るために２つのfloat型のn×n行列A、Bの積C=ABを計算するC/C++言語でのプログラムを例にとる。この計算はデータ並列の典型的な例で、比較的簡単に並列処理を記述できる。

実行した環境は

CPU：Intel(R) Core(TM)i7 870 2.93GHz

メモリ：4.0 GB
GPU：NVIDIA GeForce GTX 460
OS：Windows 7
開発環境：マイクロソフトC/C++ コンパイラ
である。

このCPUは４つのコアを持ち、ハイパースレッディング・テクノロジーを実装しているので最大で同時に8スレッドの処理が可能である。また、GPUのコア数は336である。

基本となる逐次計算を行うプログラムは以下のようになる。

行列A、B、Cを配列a、b、cで扱っている。

（matmul_seq.c）

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define SIZE 1024

float a[SIZE][SIZE], b[SIZE][SIZE], c[SIZE][SIZE];

int main(int argc, char *argv[])
{
    int i, j, k;
    clock_t start_t, end_t, elap_t;
    for (i=0; i<SIZE; i++)
        for (j=0; j<SIZE; j++) {
            a[i][j] = (rand() - RAND_MAX/2)/(float)RAND_MAX;
            b[i][j] = (rand() - RAND_MAX/2)/(float)RAND_MAX;
            c[i][j] = 0.0;
        }
    start_t = clock();
    for (i=0; i<SIZE; i++)
        for (k=0; k<SIZE; k++)
            for (j=0; j<SIZE; j++)
                c[i][j] += a[i][k]*b[k][j];
```

```
    end_t = clock();
    elap_t = end_t - start_t;
    for (i=0; i<SIZE; i+=100)
        printf("%f ", c[i][i]);
    printf("\ntime = %f\n", (float)elap_t/CLOCKS_PER_SEC);
    return 0;
}
```

行列のサイズが1024でコンパイラの最適化をしていない場合の計算時間は5.6秒である。

3　分散メモリ型システム

分散メモリ型並列コンピューティングを行うには、複数のプロセスを生成し各演算単位がそれらを並列に実行させるマルチプロセスプログラムを作成する必要がある。また、プロセス間ではメモリを共有できないため、複数の演算単位に共通のデータを処理させるには、プロセス間でデータのやり取りを行うことも必要となる。これらの処理を行うコードをユーザがすべて記述するのは煩雑で多くの労力と時間が必要となる。そこで、並列コンピューティングをより少ない労力でできるよう、これらの処理の記述の標準を制定する試みが行われるようになった。

3．1　MPI

分散メモリ型並列コンピューティングの現在の主流はMPI（Message Passing Interface）という仕様に基づくものである[3-7]。MPIの標準は、多くの組織を代表するメンバーからなるMPI Forumグループによって管理されている。1994年にMPI-1.0が発表されたのち、1997年にMPI-2.0、2009年９月にMPI2.2が、2012年８月にMPI3.0のドラフトがPublic Comment用に発表されている。

3．1．1　MPIの実装

MPIは仕様を定めたものであり、その仕様を実装した実際のソフトウェアパッケージにはいくつかのものがある。有償のものだけでなくUNIX、Linux、MSWindows用の無償で利用できるものもある。プログラミング言語としては、CおよびFORTRANが基本となっている。

（1）MPICH

アルゴンヌ国立研究所で開発されたもので、現在MPICH2として広く利用されている。

（2）LAM/MPI

LAM（Local Area Multicomputer）はインディアナ大学で開発されたものであり、完全なMPI 1.2標準とMPI-2標準の大部分を提供する。

現在では次のOpenMPI プロジェクトに引き継がれている。

（3）OpenMPI

LAMを引き継ぐもので、MPI-2 の標準を満たすものである。

（4）MSMPI

マイクロソフト社が販売しているWindows HPC Server 2008に含まれているMPIの実装である。

３．１．２　MPIのおもなAPI

MPIでは、「MPI_」で始まる種々の関数

・開始及び終了と環境データの取得

MPI_Init、MPI_Finalize、MPI_Comm_size、MPI_Comm_rank など

・一対一通信

MPI_Send、MPI_Recv、MPI_Sendrecv など

・集団通信

MPI_Bcast、MPI_Gather、MPI_Scatter、MPI_Reduce など

・その他

MPI_Wtime など

およびMPI_COMM_WORLD、MPI_FLOAT などの定数といったAPI（Application Program Interface）の仕様が定められていて、プログラム作成者がこれらを利用して並列処理のコードを記述する。

各プロセスが同じプログラムを実行するSPMD（Single Program Multiple Data）形式であるが、コードの中に各プロセスが自分の番号（rank）を取得してそれによって処理するデータの範囲を決めるなど、異なる処理をするような記述をすることになる。

３．１．３　MPI による並列プログラムの具体例

行列Cの行を各プロセスで分割して並列に計算し、各プロセスで扱うデータはMPIの通信関数を用いて送受信する。

①　各プロセスが自分の番号 rank をMPI_Comm_rankで取得する

②　このプログラムを実行するプロセス数sizeをMPI_Comm_sizeで取得する

③　行列Bに対応する配列bの要素をMPI_Broadcast関数で全プロセスに配信する

④ 行列Aに対応する配列aの要素を各プロセスの担当部分にで分けてMPI_Scatter関数で配列apに配信する

⑤ 各プロセスで計算した担当部分の配列cpをMPI_Gather関数でルートプロセスの行列Cに対応する配列cに集約する

プロセス間での行列データの送受信をしやすくするために、行列を1次元配列として扱っている。

MPIを用いるデータ並列タイプのプログラムは、各プロセスが自分のプロセスの番号を取得し、それに基づいて処理するデータの範囲を知るという形式になるのが一般的である。

（matmul_mpi.c）

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define SIZE 1024
#define IDX(i,j) (i)*n + (j)

void mat_mul(float *a, float *b, float *c, int n, int na)
{
    int i, j, k, ii, jj, kk;
    for (i=0; i<na; i++)
        for (j=0; j<n; j++)
            for (k=0; k<n; k++)
                c[IDX(i, j)] += a[IDX(i, k)] * b[IDX(k, j)];
}

int main(int argc, char *argv[])
{
    int rank, psize, n, nn, na, nna, i;
    double start_t, end_t;
    float *a, *b, *c, *ap, *cp;

    n = SIZE;
```

```
nn = n*n;
b = (float *)malloc(sizeof(float)*nn);
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);                                    ①
MPI_Comm_size(MPI_COMM_WORLD, &psize);                                   ②
na = SIZE/psize;
nna = na*n;
if (rank == 0) {
    printf("matrix size : %d x %d = %d\n", n, n, nn);
    a = (float *)malloc(sizeof(float)*nn);
    c = (float *)calloc(nn, sizeof(float));
    for (i=0; i<nn; i++) {
        a[i] = (rand() - RAND_MAX/2)/(float)RAND_MAX;
        b[i] = (rand() - RAND_MAX/2)/(float)RAND_MAX;
    }
}
ap = (float *)malloc(sizeof(float)*nna);
cp = (float *)malloc(sizeof(float)*nna);
MPI_Bcast(b, nn, MPI_FLOAT, 0, MPI_COMM_WORLD);                          ③
MPI_Scatter(a, nna, MPI_FLOAT, ap, nna, MPI_FLOAT, 0, MPI_COMM_WORLD); ④
if (rank == 0)
    start_t = MPI_Wtime();
mat_mul(ap, b, cp, n, na);
if (rank == 0) {
    end_t = MPI_Wtime();
    printf("time = %lf\n", end_t - start_t);
    start_t = MPI_Wtime();
}
MPI_Gather(cp, nna, MPI_FLOAT, c, nna, MPI_FLOAT, 0, MPI_COMM_WORLD);⑤
if (rank == 0) {
    for (i=0; i<n; i+=100)
        printf("%f ", c[IDX(i, i)]);
    printf("\n");
```

```
    }
    MPI_Finalize();
    return 0;
}
```

MPIではプログラムを実行する際に作成するプロセス数を指定する。MPICH 2 では以下のようにする。

> mpiexec –n プロセス数 –host ホスト名 実行ファイル名

このプログラムを４つのプロセスを並列実行させて実行した場合の計算時間は3.4秒である。１次元行列として扱ったことやプロセス作成等のオーバーヘッドのためにそれほどパフォーマンスの向上が得られていない。

３．２　HPF（High Performance Fortran）

HPFはFortranの文法を並列処理が簡潔に記述できるように拡張した言語仕様である[8-9]。その基本的な考え方は、ユーザが最小限の指示文（directiveディレクティブ）によってデータの分割配置の方法を指定すれば、残る作業（計算の分割と通信の生成）をコンパイラが自動的に行う、というものである。このディレクティブは通常のコンパイラではコメントとして扱われるので、既存の逐次実行プログラムとしてコンパイルすることも可能である。

その言語仕様は1991年より米国を中心とした産学の代表者によるHPFF（High Performance Fortran Forum）によって検討され、1993年にHPF1.0仕様が制定された。1997年にHPF 2.0が、また、1999年には、HPF推進協議会の前身であるHPF合同検討会（JAHPF）によって、HPFの実用性をさらに高める拡張仕様HPF/JA1.0が定められた。

HPFを利用するには対応するコンパイラが必要であるが、HPFPC（HPF推進協議会）がフリーで提供しているトランスレータ（hpfc）を用いてHPFのコードからMPIのAPIを用いたコードを生成する方法もある。

４　共有メモリ型システム

共有メモリ型並列コンピューティングでは、複数のスレッドを生成し、それらを並列に動作させるマルチスレッドプログラムが必要となる。また、共有メモリ型並列コンピューティングの場合は、複数のスレッドが同じデータにアクセスする際の競合の問題に注意深い検討が必要となる。スレッドの生成や廃棄、競合に対処する処理コードのすべて記述することな

しに、HPFの場合と同様にディレクティブで簡潔に記述できるようにするためのいくつかの仕様が発表されている。

４．１ OpenMP

共有メモリ型並列コンピューティングの仕様で最も一般的なものはOpenMP Architecture Review Boardにより管理されているOpenMPで、1997年にFortranバージョンが公表されて以降、2011年６月にはバージョン3.1が公表されている[10-13]。C/C++およびFortranバージョンがあり、現在ではマイクロソフト社のコンパイラ、GNUプロジェクトのgccなど多くのコンパイラが対応している。

４．１．１ OpenMPの記法

OpenMPではHPFと同じように逐次処理のコードの並列処理可能な部分に、並列処理用のディレクティブを記述する。そのため、逐次処理のプログラムを大きく変更することなく、並列処理プログラムを作成できることが大きな利点である。また、この指示文はC言語の場合は #pragmaプリプロセッサ指示文、Fortranの場合はコメント文の形式なので、OpenMPに対応していないコンパイラでは無視されるようになっている。

C言語の場合

#pragma omp directive-name [clauses]

と言う形式になり、ディレクティブおよび節（Clause）には以下のものがある。

① 並列実行制御（Directives）

・並列実行領域構文

parallel

・ワークシェアリング構文

ワークシェア制御

for sections single

タスク制御

task

② 同期制御

実行同期制御

barrier master ordered flush

データアクセス同期制御

atomic critical

③　データ属性制御（Clauses）

private firstfrvate threadprivate lastprivate copyprivate reduction

default shared copyin

④　その他

if nowait untied collapse schedule

４．１．２　OpenMPによる並列プログラムの具体例

（matmul_omp.c）

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define SIZE 1024

float a[SIZE][SIZE], b[SIZE][SIZE], c[SIZE][SIZE];
float a1[SIZE], b1[SIZE];

int main(int argc, char *argv[])
{
    int i, j, k, l, num_th = 1;
    clock_t start_t, end_t, elap_t;

    if (argc > 1)
        num_th = atoi(argv[1]);
    omp_set_num_threads(num_th);
    printf("number of threads : %d / %d\n", omp_get_max_threads(), num_th);
    for (i=0; i<SIZE; i++)
        for (j=0; j<SIZE; j++) {
            a[i][j] = (rand() - RAND_MAX/2)/(float)RAND_MAX;
            b[i][j] = (rand() - RAND_MAX/2)/(float)RAND_MAX;
        }
    start_t = clock();
#pragma omp parallel for private(i, j, k) shared(a, b, c)
```

```
    for (i=0; i<SIZE; i++)
        for (k=0; k<SIZE; k++)
            for (j=0; j<SIZE; j++)
                c[i][j] += a[i][k]*b[k][j];
    end_t = clock();
    elap_t = end_t - start_t;
    for (i=0; i<SIZE; i+=100)
        printf("%f ", c[i][i]);
    printf("\nelapsed time = %d\n", elap_t);
    return 0;
}
```

逐次処理のプログラムの繰り返しのfor文の前に

#pragma omp parallel for

というディレクティブを挿入するという簡単な変更でデータ並列処理を記述できる。ただし、各スレッドの扱うデータの間に依存関係がある場合には、並列化ができなかったり依存性を避ける工夫が必要であったりする。

privateとsharedという節はそのあとの括弧内の変数の属性を指定するもので、前者は変数i、j、kが各スレッド間で独立していることを、後者は配列a、b、cがスレッド間で共有している指示している。

上の例ではループの範囲を各プロセスでどう分担するかをコンパイラに任せているが、分担方法にはいくつかの選択肢があり、それを明示的に指定することもできる。

いくつのスレッドを作成するかをomp_set_num_threads関数で指定できるが、何も指定をしなければ実行環境で利用できる最大のスレッド数となる。

Microsoftのコンパイラでのコマンドラインからのコンパイルは以下のようにする。

```
> cl /openmp matmul_omp.c
```

スレッド数を4とした場合の計算時間は1.41秒であり、逐次処理の場合と比べて3倍程度のパフォーマンス向上が得られている。

4．2 TBB

TBB（Threading Building Blocks）はIntel社の提供するマルチスレッドプログラムを効率

よく作成するためのC++のライブラリである[14-15]。Intel Parallel Studioにサポート付の有償版が含まれているが、オープン化されている無償のものがインターネットからダウンロードして自由に利用できる。

目的はOpenMPと同じだが、実現方法がOpenMPがディレクティブを用いるのに対し、TBBではC++のテンプレート機能を用いたライブラリを提供するという方法となっている。そのため、OpenMPを利用するにはそれに対応したコンパイラが必要となるが、TBBは一般的なコンパイラで利用できる。

４．２．１　TBBの記法

TBBでは、データ並列処理とパイプラインという２種類の並列化の方法がある。

データ並列処理としてparallel_xxxという名前の次のテンプレート関数が用意されている。

parallel_for　parallel_reduce　parallel_scan　parallel_sort　parallel_while

単純なループをデータ並列処理として記述するには、演算子「()」を持つクラスまたは構造体を作成し、ループ内で実行したい処理をこの演算子の処理内容として記述し、parallel_forというテンプレート関数を利用する。

４．２．２　TBBによる並列プログラムの具体例

（matmul_tbb.cpp）

```
#include <iostream>
#include "tbb/task_scheduler_init.h"
#include "tbb/blocked_range2d.h"
#include "tbb/parallel_for.h"
#include "tbb/tick_count.h"

using namespace std;
using namespace tbb;

const size_t L=1024;

float a[L][L], b[L][L], c[L][L];

class matmul{
    float (*a)[L];
```

```
    float (*b)[L];
    float (*c)[L];
public:
    void operator() (const blocked_range2d<size_t>& range) const {
        float (*aa)[L] = a;
        float (*bb)[L] = b;
        float (*cc)[L] = c;
        for (size_t i = range.rows().begin(); i != range.rows().end(); i++){
            for (size_t j = range.cols().begin(); j != range.cols().end(); j++){
                float total  = 0;
                for (size_t k=0; k<L; ++k)
                    total += a[i][k] * b[k][j];
                c[i][j] = total;
            }
        }
    }
    matmul (float aa[L][L], float bb[L][L], float cc[L][L]):a(aa), b(bb), c(cc)
{}
};

int main(int argc, char *argv)
{
    task_scheduler_init init;
    tick_count start, finish;
    for (size_t i=0; i<L; i++)
        for (size_t j=0; j<L; j++) {
            a[i][j] = (rand() - RAND_MAX/2)/(float)RAND_MAX;
            b[i][j] = (rand() - RAND_MAX/2)/(float)RAND_MAX;
        }
    start = tick_count::now();
    parallel_for(blocked_range2d<size_t>(0, L, 0, L), matmul(a, b, c),
        auto_partitioner());
    finish = tick_count::now();
```

```
    for (size_t i=0; i<L; i+=100)
        cout << c[i][i] << " ";
    cout << endl;

    double duration = (finish - start).seconds();
    cout << "elapse=" << duration << endl;
    return 0;
}
```

クラスmatmulのオペレータ「()」の処理として並列化したい行列積の処理を記述し、parallel_forというテンプレート関数の引数にこの関数を与えている。また、並列処理の範囲を指定するため blocked_range というテンプレートクラスを用いている。

作成されるスレッド数をtask_scheduler_initクラスのコンストラクタの引数として与えることもできるが、省略すれば実行環境での可能な最大の数が用いられる。

スレッド数４で実行した時間は1.98秒である。

４．３　Cilk

1994年にMITがC言語向けの並列化言語としてCilk仕様を発表し、これをCilkArts社がCilk++としてリリースした後、IntelがCilkArts社を買収して自社のIntel Parallel Studio 2011でWindows環境のCilk Plusをサポートライブラリとして提供したものである[16]。GNUプロジェクトのコンパイラgccのバージョン4.7以降にはCilkの機能が組み込まれている。

４．３．１　Cilkの記法

Cilkのキーワードは以下の３個だけである。

cilk_for：forループを置き換えて、ループの反復を複数のタスクに分割し並列に処理することを指示する。

cilk_spawn：関数呼び出しを変更し、関数の呼び出し元と呼び出し先を並列に実行することを指示する。

cilk_sync：生成したスレッドが終了するまで現在の位置で待機することを指示する。

４．３．２　Cilkによる並列プログラムの具体例

ここではCilk++を用い、行列を１次元配列で扱っている。

(matmul.cilk)

```
#include <cilk.h>
#include <cilkview.h>
#include <iostream>

#define SIZE 1024

int cilk_main(int argc, char* argv[]) {
    int nn = SIZE;
    float* A = (float*) calloc(nn*nn, sizeof(float));
    float* B = (float*) calloc(nn*nn, sizeof(float));
    float* C = (float*) calloc(nn*nn, sizeof(float));

    for (int i=0; i<nn; i++)
        for (int j=0; j<nn; j++) {
            A[i*nn + j] = (rand() - RAND_MAX/2)/(float)RAND_MAX;
            B[i*nn + j] = (rand() - RAND_MAX/2)/(float)RAND_MAX;
        }
    cilk::cilkview cv;
    cv.reset();
    cv.start();
    cilk_for(unsigned int i=0; i<nn; i++) {
        int itn = i * nn;
        for (unsigned int j=0; j<nn; j++) {
            for (unsigned int k=0; k<nn; k++) {
                int ktn = k * nn;
                C[itn + j] += A[itn + k]*B[ktn + j];
            }
        }
    }
    cv.stop();
    for (int i=0; i<nn; i+=100)
        std::cout << C[i*nn + i] << " ";
```

```
    std::cout << std::endl;
    float par_time = cv.accumulated_milliseconds() / 1000.f;
    std::cout << "time = " << par_time << " seconds." << std::endl;
    free(A);
    free(B);
    free(C);
    return 0;
}
```

逐次処理におけるfor文をcilk_forに変更するという簡単な変更でデータ並列処理を記述できる。生成するスレッドの数はcilk::contextクラスのメソッドset_worker_count()で指定できるが、省略すると実行環境で利用できる最大数が用いられる。

Cilk++ではコマンドラインでのコンパイルを以下のようにする。

```
> cilkpp matmul.cilk
```

スレッド数４で実行した時間は3.82秒である。

５　GPGPU

GPGPU（General Purpose computing on Graphics Processing Units）は近年盛んになってきている技術で、本来グラフィックス処理のためのGPUを一般的な計算に利用するものである。最近のGPUには画面上の各画素に表示するデータの処理を並列に行うことで高速な描画を実現するために多くのプロセッシングコアが搭載されている。

これらのコアは一般的なCPUのコアと比較すると可能な処理は限定されるが、その数は数百と圧倒的に多いため、データ並列処理に対しては非常に大きなパフォーマンスの向上が期待される。また最近では、この技術を活かし、グラフィックス処理ではなくGPGPUを目的としたGPUも開発されている。

５．１　CUDA（Compute Unified Device Architecture）

CUDAはNVIDIA社のGPUを用いた並列コンピューティングのC言語による開発環境であり無償で提供される。CUDAに対応したGPUは、複数のSM（Streaming Multi-Processor）からなり、SMは８個のSP（Scalar Processor）を持っていて、各SPに一つのスレッドを実行させることで高い並列度の実現を可能にしている。また、SMにはその中のSPが共有する

高速にアクセスできるメモリがあり、有効に用いることで高速化を図ることができる。執筆時点での最新版はCUDA4.2であるが、現在CUDA5 Release Candidateが入手可能である。

５．１．１　CUDAによるプログラミング

CUDAにはGPUの構造に基づいたいくつかの制限があり、効率の良いプログラムを作成するにはGPUの構造を理解しておく必要がある[17-19]。

CUDAのプログラムは、CPUで実行される「ホストコード」とGPUで実行される「デバイスコード」から構成される。デバイスコードが書かれた関数をカーネル関数と呼び、この内容がSPによって並列に実行される。CUDAではGPUの制御をおこなう関数を提供するとともに、デバイスコード用にC言語の文法を拡張している。

一般的なCUDAプログラムは以下の手順をとる。

・CUDAの初期化を行う
・GPU側のメモリを確保する
・ホストからGPU側のメモリにデータを転送する
・GPUでデバイスコードを実行する
・計算結果をCPU側のメインメモリへ転送する
・後処理を行う

CUDAプログラミングの重要な要素として、各SPの実行するスレッドを特定するための仕組みがある。スレッドは階層的に、グリッド／ブロック／スレッド に分けられ、グリッド内のブロック、ブロック内のスレッドは最大３次元までの添え字で指定する。

一般的には各スレッドが配列の各要素に対する処理を並列に行うので、各スレッドの扱う配列要素を配列の次元数に合わせて指定しやすいようになっている。

GPUからアクセスできるメモリにはSM内にあるものとGPU外にあるDRAMとがあり、それぞれ容量・アクセス速度・アクセス可能範囲に大きな違いがあり、その使い方がパフォーマンスに対して大きな影響がある。

５．１．２　CUDAによる並列プログラムの具体例

void mat_mul(float* A, float* B, float* C)　がカーネル関数であり、必ず「__global__」というスコープ宣言をする。

ホスト側で行列A、Bに対応する配列hA、hBを与え、デバイス側のグローバルメモリ領域dA、dB にコピーしている。

matrixMul<<<grid, block>>>(dA, dB, dC);　がカーネル関数の呼び出しであり、gridとblockはそれぞれグリッドとブロックの次元を指定している。

カーネル関数で計算したグローバルメモリ領域にあるする結果dCをホスト側のメモリ用域にある行列Cに対応する配列hCにコピーしている。

（matmul.cu）

```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <cutil_inline.h>

#define SIZE 1024

__global__ void mat_mul(float* A, float* B, float* C)
{
    unsigned int ci = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int ri = blockIdx.y * blockDim.y + threadIdx.y;
    unsigned int si;
    float tmp = 0;
        for (si=0; si<SIZE; si++) {
        tmp += A[ci*SIZE + si]*B[si*SIZE + ri];
        __syncthreads();
    }
    C[ci*SIZE + ri] = tmp;
}

int main(int argc, char** argv)
{
    unsigned int e_size = sizeof(float)*SIZE*SIZE;
    unsigned int ci, ri, blsiz;
    unsigned int cudaTimer = 0;
    float *hA, *hB, *hC, *dA, *dB, *dC;
    if (argc > 1)
        blsiz = atoi(argv[1]);
    else {
        printf("give block size\n");
```

```
        exit(1);
    }
    hA = (float *)malloc(e_size);
    hB = (float *)malloc(e_size);
    for (ci=0; ci<SIZE; ci++)
        for (ri=0; ri<SIZE; ri++) {
            hA[ci*SIZE + ri] = (rand() - RAND_MAX/2)/(float)RAND_MAX;
            hB[ci*SIZE + ri] = (rand() - RAND_MAX/2)/(float)RAND_MAX;
        }
    cudaMalloc((void **)&dA, e_size);
    cudaMemcpy(dA, hA, e_size, cudaMemcpyHostToDevice);
    cudaMalloc((void **)&dB, e_size);
    cudaMemcpy(dB, hB, e_size, cudaMemcpyHostToDevice);
    cudaMalloc((void **)&dC, e_size);
    dim3 block(blsiz, blsiz);
    dim3 grid(SIZE/blsiz, SIZE/blsiz);
    cutCreateTimer(&cudaTimer);
    cutStartTimer(cudaTimer);
    mat_mul<<<grid, block>>>(dA, dB, dC);
    cudaThreadSynchronize();
    cutStopTimer(cudaTimer);
    printf("Time = %f milliseconds\n", cutGetTimerValue(cudaTimer));
    cutDeleteTimer(cudaTimer);
    hC = (float *)malloc(e_size);
    cudaMemcpy(hC, dC, e_size, cudaMemcpyDeviceToHost);
    for (ci = 0; ci<SIZE; ci+=100)
        printf("%f ", hC[ci*SIZE + ci]);
    printf("\n");
    free(hA); free(hB); free(hC);
    cudaFree(dA); cudaFree(dB); cudaFree(dC);
    cudaThreadExit();
    return 0;
}
```

このファイルのコマンドラインからのコンパイルは以下のようにする。

```
> nvcc –o matmul matmul.cu
```

ブロックサイズを16として実行した場合の実行時間は0.327秒でありCPUによる逐次処理と較べて大幅なパフォーマンスアップが得られる。

５．２　OpenCL

OpenCL（Open Computing Language）は、OpenCL C言語によるマルチコアCPUやGPUなどによる異種混在の計算資源（ヘテロジニアス環境）を利用した並列コンピューティングのためのフレームワークである[20-23]。OpenCLの仕様はアップル社によって提案されたのち、標準化団体クロノス・グループの作業部会OpenCL Working Groupによって策定されている。仕様はオープン標準として公開されており、仕様に基づいたフレームワークの実装はサードパーティによって行われている。

OpenCL 1.1が2010年６月に発表され、2012年２月現在の最新の正式版は2011年11月に発表されたOpenCL 1.2である。

CUDAを開発したNVIDIAは自社のGPU向けのOpenCLによる開発環境を無償で提供している。

５．２．１　OpenCL によるプログラミング

OpenCLでは処理環境を次の２つに分類する。

・ホスト

制御用のソフトウェアが動作する環境で、通常はCPUとメインメモリ

・OpenCLデバイス

演算用のソフトウェアが動作する環境で、GPUやCPUと付随するメモリ

CUDAと異なり、CPU内のコアをOpenCLデバイスとして扱うこともできる。

ソフトウェアについてもホスト側で動作するものをホストプログラム、デバイス側で動作するものをカーネルと呼び区別する。

ホスト用のソースコードはOpenCLランタイムAPIを利用して記述し、カーネルコードと呼ばれるデバイス用のソースコードはOpenCL C言語で記述する。

OpenCLでの処理はカーネルインスタンスであるワークアイテムという単位で行われる。複数のワークアイテムをワークグループという単位にまとめることができ、ワークアイテムとワークグループを２あるいは３次元のインデックスで識別するようにできる。

同じワークグループ内のワークアイテムは同期やローカルメモリの共有が可能となっている。データ並列のカーネルを実行する場合には、コード内でワークアイテムとワークグループの数を指定する必要がある。

ワークグループを構成するワークアイテムの最大値は実行環境によって制限される。

OpenCLのワークアイテムはCUDAにおけるスレッドに対応する。

ホストコード内でカーネルコード内の処理の実行を記述するために、以下の手続きが必要となる。

1．プラットフォームの特定
clGetPlatformIDs

2．デバイスの特定
clGetDeviceIDs

3．コンテキストの作成
clCreateContext

4．コマンドキューの作成
clCreateCommandQueue

5．メモリオブジェクトの作成
clCreateBuffer

6．カーネルファイルの読み込み

7．プログラムオブジェクトの作成
clCreateProgramWithSource　　オンラインコンパイル方式
または
clCreateProgramWithBinary　　オフラインコンパイル方式

8．カーネルコードのコンパイル
clBuildProgram

9．カーネルオブジェクトの作成
clCreateKernel

10．カーネル引数の設定
clSetKernelArg

11．カーネルの実行（コマンドキューへの投入）
clEnqueueNDRangeKernel　　データ並列指示の場合
または
clEnqueueTask　　タスク並列指示の場合

12．メモリオブジェクトからの読み込み

clEnqueueReadBuffer

13. 各オブジェクトの解放

clFlush(command_queue);

clFinish(command_queue);

clReleaseKernel(kernel);

clReleaseProgram(program);

clReleaseMemObject(memobj);

clReleaseCommandQueue(command_queue);

clReleaseContext(context);

カーネルコードはホストコード実行時に読み込まれるが、事前にコンパイルしておいたバイナリデータを読み込むオフラインコンパイル方式と、ソースのテキストデータを読み込んでコンパイルするオンラインコンパイル方式とが規定されている。

５．２．３　OpenCLによる並列プログラムの具体例

ここではオンラインコンパイル方式を用いてカーネルのソースコードをファイル（matmul.cl）から読み込んでいる。

（ホストコード matmul_ocl.c）

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <CL/cl.h>

#define SIZE 1024
#define MAX_SOURCE_SIZE 20000

int main(int argc, char **argv)
{
    cl_platform_id pl_id = NULL;
    cl_device_id dev_id = NULL;
    cl_context ctxt = NULL;
    cl_command_queue com_q = NULL;
```

```
cl_mem Amobj, Bmobj, Cmobj, ndim;
cl_program prg;
cl_kernel kern;
cl_uint num_pl, num_dev;
cl_int ret, n, nn;
int i, j;
float *A, *B, *C;
FILE *fp;
char *src_str, fn[] = "matmul.cl";
int src_siz;
size_t globalworksize, localworksize;

clock_t start_t, end_t;
char build_c[4096];
printf("%d %d\n", CL_INVALID_PROGRAM, CL_INVALID_PROGRAM_EXECUTABLE);
n = SIZE;
nn = n*n;
A = (float *)malloc(sizeof(float)*nn);
B = (float *)malloc(sizeof(float)*nn);
C = (float *)malloc(sizeof(float)*nn);
for (i=0; i<SIZE; i++)
    for (j=0; j<SIZE; j++) {
        A[i*SIZE + j] = (rand() - RAND_MAX/2)/(float)RAND_MAX;
        B[i*SIZE + j] = (rand() - RAND_MAX/2)/(float)RAND_MAX;
    }
if ((fp = fopen(fn, "r")) == NULL) {
    printf("file open error\n");
    exit(1);
}
src_str = (char *)malloc(MAX_SOURCE_SIZE);
src_siz = fread(src_str, 1, MAX_SOURCE_SIZE, fp);
start_t = clock();
ret = clGetPlatformIDs(1, &pl_id, &num_pl);
```

```
    ret = clGetDeviceIDs(pl_id, CL_DEVICE_TYPE_DEFAULT,1, &dev_id, &num_dev);
    ctxt = clCreateContext(NULL, 1, &dev_id, NULL, NULL, &ret);
    com_q = clCreateCommandQueue(ctxt, dev_id, 0, &ret);

    Amobj = clCreateBuffer(ctxt, CL_MEM_READ_WRITE, sizeof(float)*nn, NULL,
&ret);
    Bmobj = clCreateBuffer(ctxt, CL_MEM_READ_WRITE, sizeof(float)*nn, NULL,
&ret);
    Cmobj = clCreateBuffer(ctxt, CL_MEM_READ_WRITE, sizeof(float)*nn, NULL,
&ret);
    ndim = clCreateBuffer(ctxt, CL_MEM_READ_WRITE, sizeof(int), NULL, &ret);

    ret = clEnqueueWriteBuffer(com_q, Amobj, CL_TRUE, 0, nn*sizeof(float), A, 0,
NULL, NULL);
    ret = clEnqueueWriteBuffer(com_q, Bmobj, CL_TRUE, 0,  nn*sizeof(float), B, 0,
NULL, NULL);
    ret = clEnqueueWriteBuffer(com_q, Cmobj, CL_TRUE, 0,  nn*sizeof(float), C, 0,
NULL, NULL);
    ret = clEnqueueWriteBuffer(com_q, ndim, CL_TRUE, 0, sizeof(int), &n, 0,
NULL, NULL);

    prg = clCreateProgramWithSource(ctxt, 1, (const char **)&src_str, (const
size_t *)&src_siz, &ret);
    ret = clBuildProgram(prg, 1, &dev_id, NULL, NULL, NULL);
    if (ret < 0) {
        clGetProgramBuildInfo(prg, dev_id, CL_PROGRAM_BUILD_LOG, 4096, build_c,
NULL);
        printf("Build Log for program:\n%s\n", build_c );
        exit(1);
    }
    kern = clCreateKernel(prg, "my_matmul", &ret);
    ret = clSetKernelArg(kern, 0, sizeof(cl_mem), (void *)&Amobj);
    clSetKernelArg(kern, 1, sizeof(cl_mem), (void *)&Bmobj);
```

```
    clSetKernelArg(kern, 2, sizeof(cl_mem), (void *)&Cmobj);
    clSetKernelArg(kern, 3, sizeof(cl_mem), (void *)&ndim);
    localworksize = n;
    globalworksize = n;
    ret = clEnqueueNDRangeKernel(com_q, kern, 1, NULL, &globalworksize,
&localworksize, 0, NULL, NULL);

    clEnqueueReadBuffer(com_q, Cmobj, CL_TRUE, 0, nn*sizeof(float), C, 0, NULL,
NULL);
    clFlush(com_q);
    clFinish(com_q);
    clReleaseKernel(kern);
    clReleaseProgram(prg);
    clReleaseMemObject(Amobj);
    clReleaseMemObject(Bmobj);
    clReleaseMemObject(Cmobj);
    clReleaseCommandQueue(com_q);
    clReleaseContext(ctxt);
    end_t = clock();

    for (i=0; i<SIZE; i+=100)
        printf("%f ", C[i*SIZE + i]);
    printf("\n");
    printf("time = %f msec\n", (end_t - start_t)/(float)CLOCKS_PER_SEC*1000);
    free(src_str);
    free(A);
    free(B);
    free(C);
    return 0;
}
```

（カーネルコード matmul.cl）

```
__kernel void my_matmul(__global float *A, __global float *B, __global float *C,
```

```
__global int *n)
{
    int i, j, k, ik, kj, ij, nd;
    float tmp = 0.0;
    i = get_global_id(0);
    j = get_local_id(0);
    nd = *n;
    ij = i*nd + j;
    for (k=0; k<nd; k++) {
        ik = i*nd + k;
        kj = k*nd + j;
        tmp += A[ik]*B[kj];
    }
    C[ij] = tmp;
}
```

オンラインコンパイル方式の場合、カーネルコードのコンパイルエラーの情報は自動的には表示されないので、clGetProgramBuildInfo関数で取得する必要がある。

配列の大きさであるn個のワークアイテムを一つにまとめたワークグループをn個作成している。

一般的にCUDAで記述した場合と較べてソースコードがかなり長くなる。

計算時間は0.213秒と大幅なパフォーマンスの向上が得られた。

５．３　DirectCompute

Microsoft DirectComputeは、Windows VistaとWindows 7 OS上でGPGPUをサポートするAPIの１つである。最初DirectComputeはMicrosoft DirectXの一部として DirectX 11 API に含めてリリースされたが、その後、DirectX 10とDirectX 11に対応するGPU上で実行できるようになった。

CUDAやOpenCLに較べて詳しい情報が少ないために、あまり普及していないようである。

６　新しい進展

新しいハードウェア面での新技術としては、CPU内のコア数をさらに増やすものとしてIntel のMIC（Many Integrated Core）アーキテクチャーに基づくプロセッサがあり、50以上のコアを搭載する。Intelはまた、開発コードSandy Bridge という名前のGPU機能をダイレ

ベルでCPUに統合した第2世代のIntel Core i7/i5/i3シリーズの情報を発表している。

CPUから分離したGPUあるいはCPUと統合されたGPUなどのデータ並列用のデバイスはAccelerator と呼ばれ、今後それらの高性能化が進んでいくことが期待される。

GPUを利用するソフトウェアを開発するためのCUDAやOpenCLではGPUに並列に実行させる処理をデバイスコードとしてホストコードとは別に作成し、ホストコードにはデバイス側へのデータ転送処理などを記述するなど学習しなくてはならない内容が多く、GPGPUを行うには敷居が高いという面がある。また、従来の逐次処理プログラムを大きく書き換える必要があるため、過去の資産を有効に利用できない。ソフトウェア面では、これらの点を改善するための次に述べるような進展がある。

６．１　OpenHMPP、OpenACC、OpenMP4.0

OpenHMPP（Hybrid Multicore Parallel Programming）とOpenACCはこの敷居をできるだけ低くし、また過去の資産を有効活用できるようにすることを目的として制定された。これらは、OpenMPが逐次処理のプログラムの並列化したい部分にディレクティブを挿入することで並列処理のプログラムを作成できたように、GPUによる並列処理の記述をディレクティブの挿入でできるようにすることを目指したものである。ホスト―デバイス間のメモリ転送もディレクティブの節として指示することでコンパイラが管理する。

OpenHMPPはCAPS enterprise、GENCI、ICHEC、INRIA、PathScale Inc社がOpenHMPP Consortiumとして2011年６月にVersion1.0を発表し、製品として販売している。

OpenACCはNVIDIA、CAPS、PGI、Crayが共同で策定したもので、2011年11月にOpenACC API Version1.0が発表され、The Portland Group Inc (PGI) 社がこれに対応した「PGI Acceleratorコンパイラ」をFortranおよびC向けに販売している。

また、OpenMP Language CommitteeはOpenMP4.0としてAcceleratorを用いた並列コンピューティングも含める方向の検討を行っている。

６．１．１　OpenACCによる並列プログラムの具体例

（matmul_acc.c）

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define SIZE 1024
```

```
float a[SIZE][SIZE], b[SIZE][SIZE], c[SIZE][SIZE];

int main(int argc, char** argv)
{
    int i, j ,k;
    clock_t start_t, end_t, elap_t;
    for (i=0; i<SIZE; ++i)
        for (j=0; j<SIZE; ++j) {
            a[i][j] = (rand() - RAND_MAX/2)/(float)RAND_MAX;
            b[i][j] = (rand() - RAND_MAX/2)/(float)RAND_MAX;
            c[i][j] = 0.0f;
        }
    start_t = clock();
#pragma acc kernels copyin(a, b) copy(c)
    for (i = 0; i < SIZE; ++i) {
        for (k = 0; k < SIZE; ++k) {
            for (j = 0; j < SIZE; ++j) {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
    end_t = clock();
    elap_t = end_t - start_t;
    printf("\ntime = %d\n", elap_t);
    for (i=0; i<SIZE; i+=100)
        printf("%f ", c[i][i]);
    return 0;
}
```

#pragma acc kernelというディレクティブでデバイス側での処理内容を指示し、copyinあるいはcopy句でホスト－デバイス間のデータ転送の指示をしている。並列実行させるスレッドの数はコード中で指定しなければコンパイラが決めてくれる。CUDAでのgridDim、blockDimに対応したgang、worker、vectorといった節を使用してコード中で指定することも

できる。CUDAやOpenCLと較べてはるかに簡潔に記述できる。

現在OpenACCに対応したコンパイラはあまり多くないが、PGI社のコンパイラでは

> pgcc –acc matmul_acc.c

としてコンパイルする。計算時間は0.168秒であった。

６．２ C++ AMP（Accelerated Massive Parallelism）

Microsoft はVisual Studio 2012として、ヘテロジーニアス環境で並列実行するプログラムの開発を容易にするためにC++を拡張したC++AMPを提供している。OpenACCやOpenHMPPのようにディレクティブによって並列実行部分を指定するのではなく、C++言語自体を拡張している[24]。

６．２．１ C++AMPによる並列プログラムの具体例

行列A、B、Cに対して1次元配列v_a、v_b、v_cを用いている。

（matmul_amp.cpp）

```
#include <amp.h>
#include <iostream>
#include <time.h>

using namespace concurrency;

#define DATA_TYPE float
#define M 1024

template<typename _type>
void init_array(std::vector<_type> &v_a, std::vector<_type> &v_b, unsigned size)
{
    for (unsigned i=0; i<size; i++)
        for (unsigned j=0; j<size; j++) {
            v_a[i*size + j] = (rand() - RAND_MAX/2)/(_type)RAND_MAX;
            v_b[i*size + j] = (rand() - RAND_MAX/2)/(_type)RAND_MAX;
```

```
        }
}

template<typename _type>
void mat_mul(int m, const std::vector<_type>& va, const std::vector<_type>& vb,
std::vector<_type>& vc)
{
    extent<2> e_a(m, m), e_b(m, m), e_c(m, m);
    array_view<const _type, 2> av_a(e_a, va);
    array_view<const _type, 2> av_b(e_b, vb);
    array_view<_type, 2> av_c(e_c, vc);
    av_c.discard_data();
    parallel_for_each(av_c.extent, [=](index<2> idx) restrict(amp)
        {
            for(int i = 0; i < av_a.extent[1]; ++i) {
                index<2> idx_a(idx[0], i);
                index<2> idx_b(i, idx[1]);
                av_c[idx] += av_a[idx_a]*av_b[idx_b];
            }
        });
    av_c.synchronize();
}

int main(int argc, char** argv)
{
    accelerator default_device;
    std::wcout << L"Device : " << default_device.get_description() << "\n";
    std::vector<DATA_TYPE> v_a(M*M);
    std::vector<DATA_TYPE> v_b(M*M);
    std::vector<DATA_TYPE> v_c(M*M);
    init_array(v_a, v_b, M);
    printf("Matrix size = %d\n", M);
    clock_t start_t, end_t;
```

```
    start_t = clock();
    mat_mul(M, v_a, v_b, v_c);
    end_t = clock();
    for (int i=0; i<M; i+=100)
        printf("%f ", v_c[i*M + i]);
    printf("\ntime = %f msec\n", (end_t - start_t)/(float)CLOCKS_PER_SEC*1000);
    return 0;
}
```

並列処理の対象となる配列データをindex、extent、array_view といったクラスで扱い、parallel_for_each() というメソッドで並列処理を指示している。デバイスによる処理内容はラムダ表現と呼ばれる形式で記述し、parallel_for_each()の2つ目の引数として与えている。

計算時間は0.124秒である。

7 おわりに

計算を行う単位としてのCPUあるいはコア単体の性能向上が限界に近づいている中で、HPCの進展は複数の演算単位によって処理を並列分担して行うことでパフォーマンスを向上させる並列コンピューティングの方向に向かっている。

PCを用いた並列コンピューティングシステムにも、複数のPCをネットワークで接続したPCクラスタ、複数のCPUを持つPCあるいは複数のコアを持つCPUを用いるもの、グラフィックス演算用の多数のコアを持つGPUを用いるもの、あるいはこれらのハイブリッドなど多様な形態がある。一方、プログラム作成手法もそれを動作させるシステムの形態によってそれぞれ異なってくる。プログラム作成者の側からは、システムの違いによる動作プログラムに違いが少ないほうが望ましいと同時に、過去の逐次処理プログラムの膨大な資産を有効活用できることも重要である。

OpenMP、OpenCL、OpenHMPP、OpenACCあるいはC++AMPといった仕様はすべてこの点に沿って考案されてきたもので、今後の進展もこの方向で進められるものと思われる。

今後もさらに研究が進み、一般の科学者や技術者が並列コンピューティングを自らの目的のために抵抗なく利用できるようになることが期待される。

参考文献

［1］K. David、C. Severance、High Performance Computing、O'Reily、1998/7

［2］横川三津夫、特集　スーパーコンピュータ「京」、情報処理Vol.53 No8、2012/8

［3］T. L. Sterling、J. Salmon、D. J. Becker、D. F. Savarese、PCクラスタ構築法、産業図書、2001/3

［4］P. S. Pacheco、MPI並列プログラミング、培風館、2001/7

［5］W. Gropp、R. Thakur、E. Lusk、実践MPI－2―メッセージパッシング・インタフェースの上級者向け機能、ピアソンエデュケーション、2002/9

［6］G. E. Karaniadakis、R. M. Kirby II、Parallel Scientific Computing in C++ and MPI、Cambridge University Press、2003

［7］大塚　登、Symplectic Integratorによる運動方程式の数値解法とMPIによる並列計算、尾道大学経済情報論集、Vol3 No2、2003/12

［8］C. H. Koelbel、D. B. Loveman、R. S. Schreiber、G. L. Steele Jr.、M. E. Zosel、The High Performance FORTRAN Handbook、The MIT Press、1994

［9］岩下 英俊、坂上 仁志、妹尾 義樹、林 康晴、PCクラスタで並列プログラミング―High Performance Fortranで楽々並列化、培風館、2011/3

［10］R. Chandra、L. Dagum、D. Kohr、D. Maydan、J. McDonald、R. Menon、Parallel Programming in OpenMP、ACADEMIC PRESS、2001

［11］牛島 省、OpenMPによる並列プログラミングと数値計算法、丸善、2006/5

［12］北山 洋幸、OpenMP入門―マルチコアCPU時代の並列プログラミング、秀和システム、2009/8

［13］菅原 清文、C/C++プログラマーのためのOpenMP並列プログラミング、カットシステム、2012/6

［14］J. Reinders、インテル スレッディング・ビルディング・ブロック ―マルチコア時代のC++並列プログラミング、オライリー・ジャパン、2008/2

［15］日向 俊二、マルチコアのためのIntelスレッディング・ビルディング・ブロック入門―マルチコアCPUの能力を引き出すプログラミング技法、カットシステム、2008/3

［16］菅原 清文、Cilkがやってきた―C/C++プログラマーのための並列プログラミング言語、カットシステム、2011/1

［17］青木 尊之、額田 彰、はじめてのCUDAプログラミング、工学社、2009/11

［18］J. Sanders、E.Kandrot、CUDA by Example 汎用GPUプログラミング入門、インプレスジャパン、2011/2

［19］岡田 賢治、小山田 耕二、CUDA高速GPUプログラミング入門、秀和システム、2010/3

［20］池田 成樹、OpenCL並列プログラミング―マルチコアCPU/GPUのための標準フレームワーク、カットシステム、2010/1

［21］奥薗 隆司、OpenCL入門―GPU&マルチコアCPU並列プログラミング for MacOS Windows Linux、秀和システム、2010/5

［22］A. Munshi、OpneCL 詳説、Khronos OpenCL Working Group、カットシステム、2011/9

［23］株式会社フィックスターズ、土山 了士、中村 孝史、飯塚 拓郎、OpenCL入門 1.2対応 マルチコアCPU・GPUのための並列プログラミングインプレスジャパン、2012/3

［24］K. Gregory、A. Miller、C++AMP、Microsoft Press、2012/10