FDTD法の並列化技術とオープンソース化

株式会社EEM　大賀明夫

2014年6月12日

電子情報通信学会　アンテナ・伝播研究会

並列化技術 (CPU編)

1. OpenMP
・共有メモリー型
・並列化可能なfor文の直前にディレクティブを記入し、コンパイラが自動並列化を行う
・ほとんどのコンパイラが対応している

2. マルチスレッド
・共有メモリー型
・スレッド生成関数で複数のスレッドを起動し計算処理を分割する
・OSとコンパイラが標準でサポートしている

3. MPI
・分散メモリー型
・複数のプロセスを起動し計算処理を分割する
・プロセス間のデータのやりとりはMPI関数を用いた通信により行う

〇共有メモリー
一台のコンピュータのマルチコア・マルチCPU環境で複数のスレッドを起動する。
すべての変数はどのスレッドからもアクセスできるが競合するときはロックが必要。
〇分散メモリー
複数台のコンピュータで複数のプロセスを起動する。
各プロセスのメモリー空間は独立であるからデータをやりとりするには通信が必要。

並列化技術 (GPU編)

1. CUDA
・NVIDIAのグラフィックスボードを用いて高速に計算する
・計算処理を多数のスレッドに分割する、軽量スレッド
・メモリーアクセス速度はCPUより一桁速い

2. OpenCL
・アーキテクチャはCUDAと同じ
・プログラムもCUDAとほぼ1対1に対応する
・各種のデバイス(PC以外も含む)で動作することが特長

3. MPI
・GPUクラスタ用

○GPGPU (General-Purpose computing on Graphics Processing Units)
グラフィックスボードの高い演算性能を汎用の科学技術計算に応用すること

FDTD法

電界のX成分を更新する式

$$E_x^{n+1}\left(i+\frac{1}{2},j,k\right)=c_1\left(i+\frac{1}{2},j,k\right)E_x^{n}\left(i+\frac{1}{2},j,k\right)$$
$$+c_2\left(i+\frac{1}{2},j,k\right)\left\{\frac{\eta H_z^{n+\frac{1}{2}}\left(i+\frac{1}{2},j+\frac{1}{2},k\right)-\eta H_z^{n+\frac{1}{2}}\left(i+\frac{1}{2},j-\frac{1}{2},k\right)}{\Delta y_j/(c\,\Delta t)}\right.$$
$$\left.-\frac{\eta H_y^{n+\frac{1}{2}}\left(i+\frac{1}{2},j,k+\frac{1}{2}\right)-\eta H_y^{n+\frac{1}{2}}\left(i+\frac{1}{2},j,k-\frac{1}{2}\right)}{\Delta z_k/(c\,\Delta t)}\right\}$$
$$\left(i=0,\cdots,N_x-1,\ \ j=0,\cdots,N_y,\ \ k=0,\cdots,N_z\right) \qquad (7)$$

FDTDプログラムの主要部

表1 Ex成分を更新するプログラム(逐次版)

```
 1 #define NA(i,j,k) ((i)*Ni+(j)*Nj+(k)*Nk+N0)
 2 void updateEx(void) {
 3   int i, j, k;
 4   size_t n;
 5   for (i = 0; i <  Nx; i++) {
 6   for (j = 0; j <= Ny; j++) {
 7   for (k = 0; k <= Nz; k++) {
 8     n = NA(i, j, k);
 9     Ex[n] = C1[iEx[n]] * Ex[n]
10           + C2[iEx[n]] * (RYn[j] * (Hz[n] - Hz[n - Nj])
11                          - RZn[k] * (Hy[n] - Hy[n - Nk]));
12   }
13   }
14   }
15 }
```

Ex[n]へのアクセスはループ間で競合しない
→3重forループはどのように並べ替えても結果は変わらない
→FDTD法は極めて並列計算向きのアルゴリズムである

OpenMP

表2 Ex成分を更新するプログラム（OpenMP版）

```
 1 void updateEx(void) {
 2   int i, j, k;
 3   size_t n;
 4 #pragma omp parallel for private(i,j,k,n)
 5   for (i = 0; i <  Nx; i++) {
 6   for (j = 0; j <= Ny; j++) {
 7   for (k = 0; k <= Nz; k++) {
 8     n = NA(i, j, k);
 9     Ex[n] = C1[iEx[n]] * Ex[n]
10           + C2[iEx[n]] * (RYn[j] * (Hz[n] - Hz[n - Nj])
11                          - RZn[k] * (Hy[n] - Hy[n - Nk]));
12   }
13   }
14   }
15 }
```

違いはこの1行だけ

OpenMPプログラムの特徴

(1)ほとんどのコンパイラがサポートしている
(2)OSに無関係な規格（ソースコードに互換性がある）
(3)並列化可能なプログラムについては簡単に並列化できる
　　（指示文のprivate, reductionについての知識は必要）
(4)コンパイルオプションを付けないと指示文(#pragma)はコメントとみなされる
　　→逐次プログラムとの切り替えが簡単
(5)個別に並列化できる（共有メモリーの特徴、データ構造が変わらないため）
　　→開発効率が高い

private：
変数をスレッドごとに持ちスレッド間で共有しない
reduction（還元演算）:
全スレッドの共同作業（例：配列の和を求める）

〇コンパイルオプション：
VC++ : /openmp
gcc : -fopenmp
〇リンクオプション：
VC++ : なし
gcc : -fopenmp
〇リンクするライブラリ：
なし
〇実行時に必要なライブラリ：
VC++ : vcomp110.dll
gcc : なし

マルチスレッド

・複数のスレッドを起動し、計算処理をスレッドに分割する
・プログラム内ではスレッド番号に応じた処理を記述する

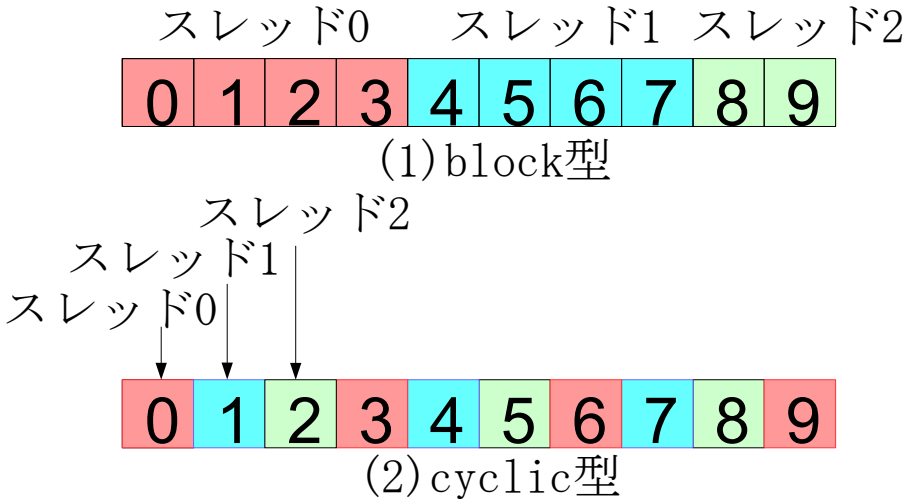


図2 2種類のループ分割法（ループ長=10,スレッド数=3）

表3 Ex成分を更新するプログラム（マルチスレッド版）

```
 1 typedef struct {
 2   int    tid;
 3 } update_arg_t;
 4 void updateEx_thread(void *arg) {
 5   update_arg_t *targ = (update_arg_t *)arg;
 6   int    tid = targ->tid;
 7   int    i, j, k;
 8   size_t n;
 9   int    nu, i0, i1;
10   nu = ((Nx + 0) + (nThread - 1)) / nThread;
11   i0 = tid * nu;
12   i1 = MIN(i0 + nu, Nx + 0);
13   for (i = i0; i < i1; i++) {
14   for (j = 0; j <= Ny; j++) {
15   for (k = 0; k <= Nz; k++) {
```

スレッド引数用構造体

スレッド番号を取得する

block型

（以下同じ）

```
    for (i = tid; i <  Nx; i += nThread) {
```

cyclic型

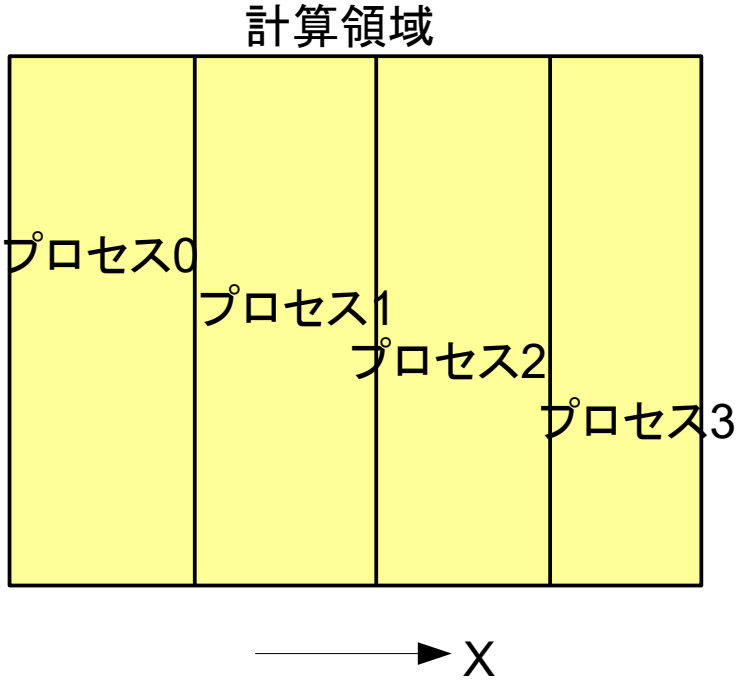
マルチスレッドプログラムの特徴

(1)すべてのOSとコンパイラが標準でサポートしている
　　(ただしプログラムはOSで異なる)
(2)プログラミング量は増えるが、並列計算に関する細かい調整が可能
　　（ただしFDTD法では細かい調整は不要）
(3)スレッドの起動は比較的コストがかかる

〇コンパイルオプション:
なし
〇リンクオプション:
なし
〇リンクするライブラリ:
VC++ : なし
gcc : -lpthread
〇実行時に必要なライブラリ:
なし

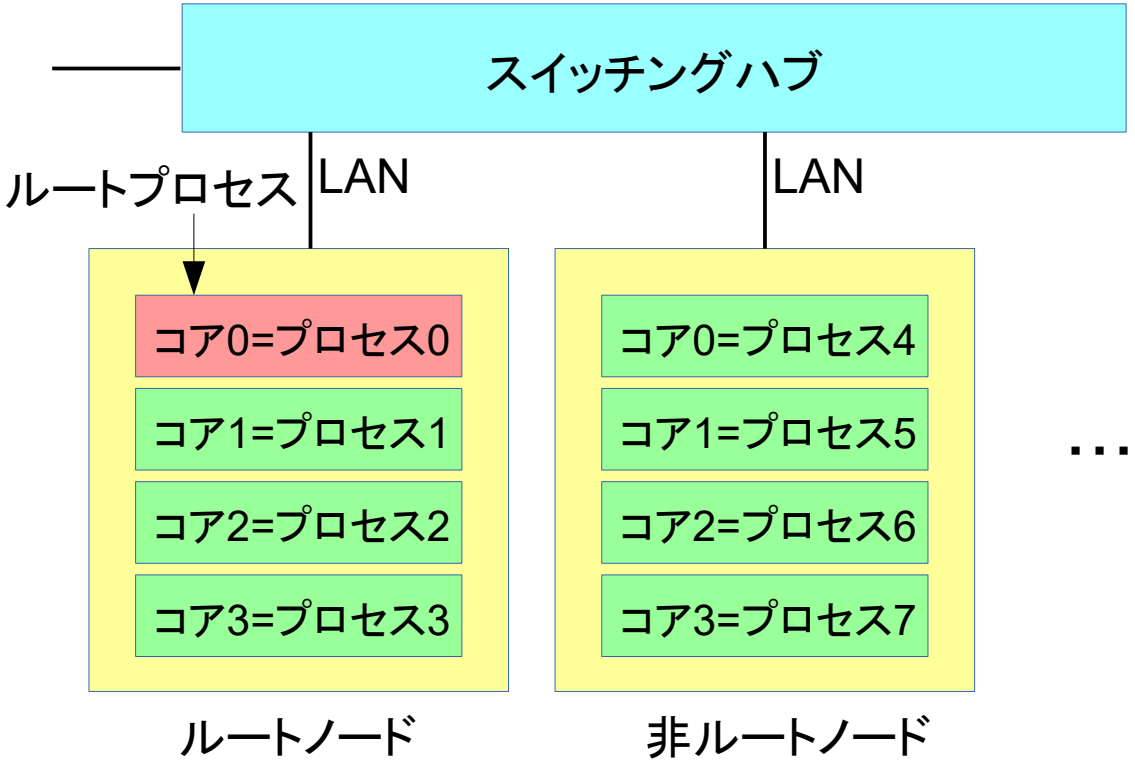
MPI

・複数のプロセスを起動する
　　→メモリー空間が独立
・領域を分割する
　　→プロセス数を増やせば計算モデルに上限がない（スパコン向き）
・プロセス間のデータのやりとりはMPI関数による通信で行う
　　→性能がネットワーク速度に影響される
・SPMD（Single Program Multiple Data）プログラミングモデル
　　→動くプログラムは同じであるが同じ変数の内容がプロセスですべて異なる



MPIの領域分割(4プロセスの場合)

PCクラスタとMPI



MPIプログラムの処理の流れ:

(1)ファイル入力
ルートプロセス

(2)入力データの配信(broadcast)
ルートプロセス→全プロセス

(3)計算
全プロセス

(4)計算結果の集計(gather)
全プロセス→ルートプロセス

(5)ファイル出力
ルートプロセス

MPIの通信：
・タイムステップ毎に不足成分を隣のプロセスから通信によって取得する
・不足成分：領域境界の半セル外側のHy,Hz
・重複成分：領域境界のEy,Ez,Hx

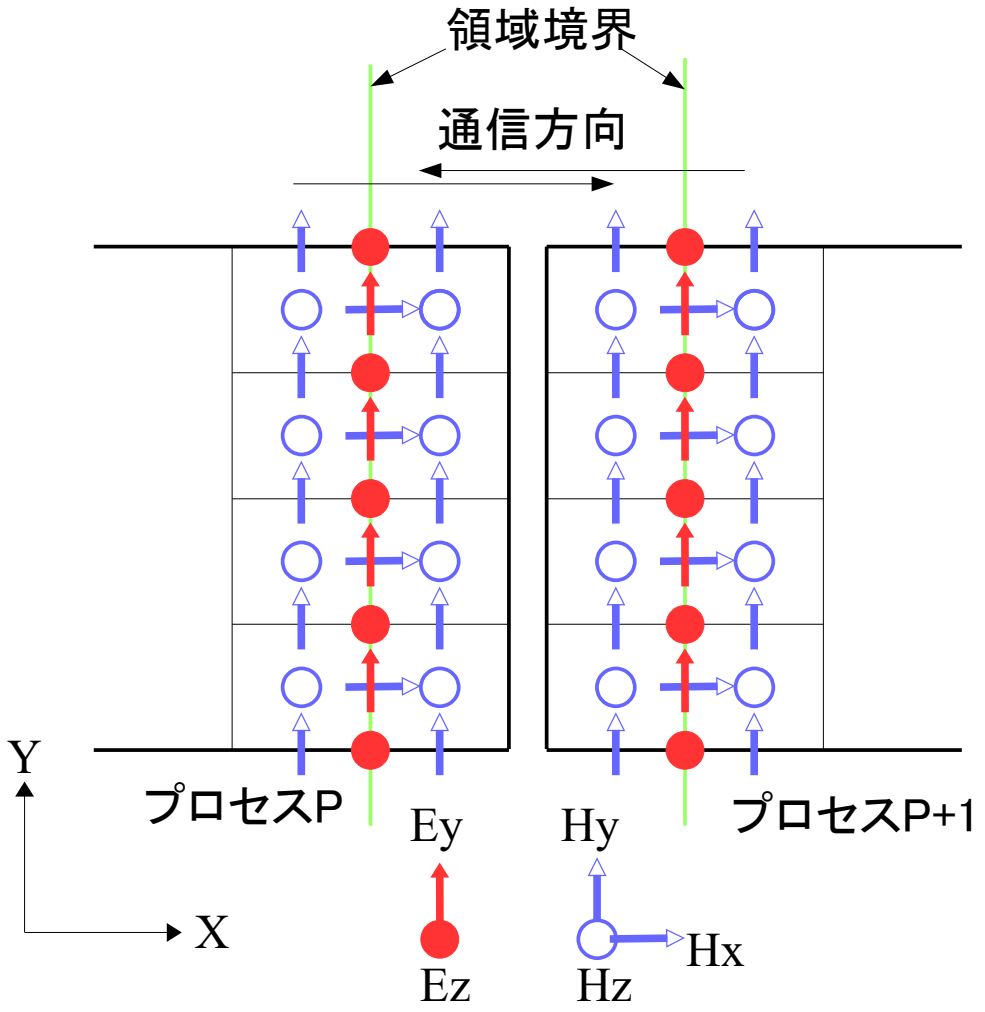


図3 MPIの通信

MPIクラスタの評価

使用メモリー　= 30 Nx Ny Nz / Np　[byte]　　(Np : プロセス数)

計算時間 = 5e-8 Nt Nx Ny Nz / Np [sec]　(標準的なCPU、Nt : タイムステップ数)

通信時間 = Nt　(T_L + (16 Ny Nz) / B[byte/sec])

　　T_L : 通信遅延(latency) ～ 10μsec　: FDTD法では無視できる
　　B : バンド幅 [byte/sec]

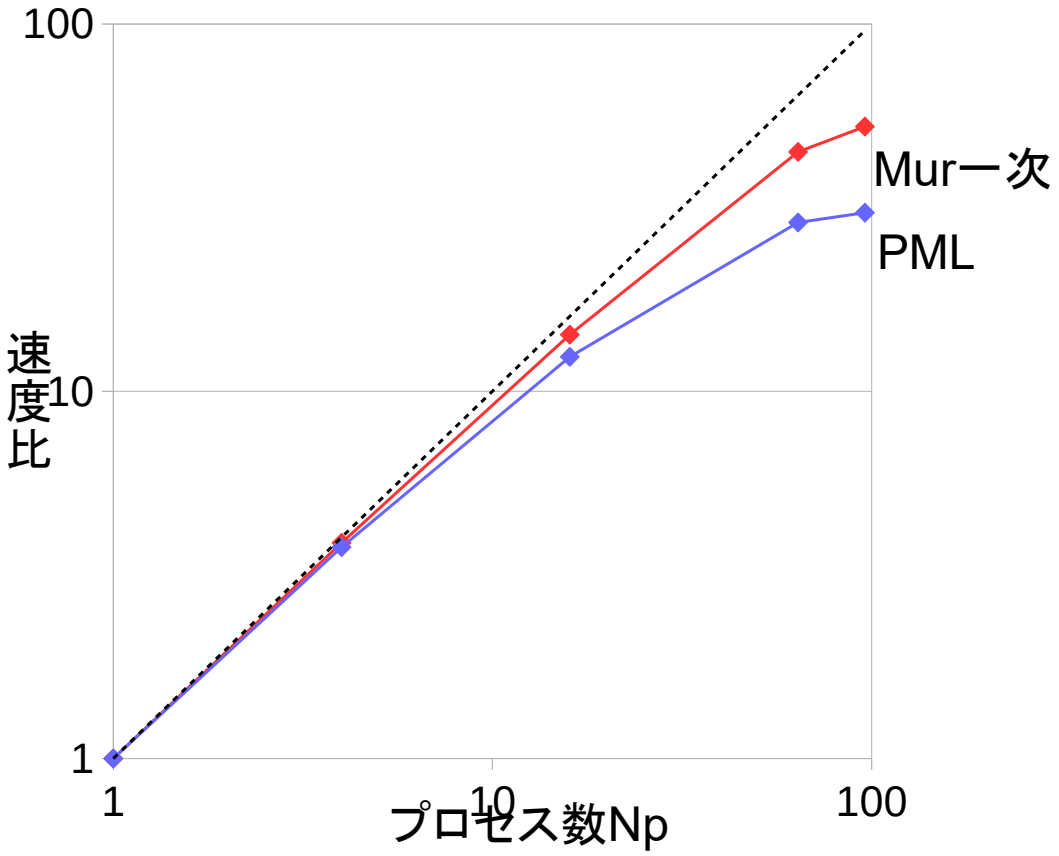
計算時間 / 通信時間 =　3e-9 B Nx / Np

ここで　B = 40[MB/sec] (ギガビットイーサネット)　とすると

計算時間 / 通信時間 =　0.1 Nx / Np

結論 :
・Nx / Np >> 10 のときクラスタにより計算時間が短縮できる
（例えばNx = 500 のときは Np < 10程度まで）
・通信が速くなるとNpの上限が大きくなる

プロセス数Npと速度比の関係(MPI)
・96プロセスで30倍(PML)～50倍(Mur一次)速くなる
・プロセス数が増えると通信量/計算量の比（～Np/Nx）が上がりネットワーク速度がネックになる
・吸収境界条件は負荷分散が不均一（両端のプロセスの負荷が大きい←改善可能）
・PMLは計算量自体が多いので影響が大きい



プロセス数と速度比の関係（MPI）
Nx=Ny=Nz=500
Fujitsu FX1 InfiniBand

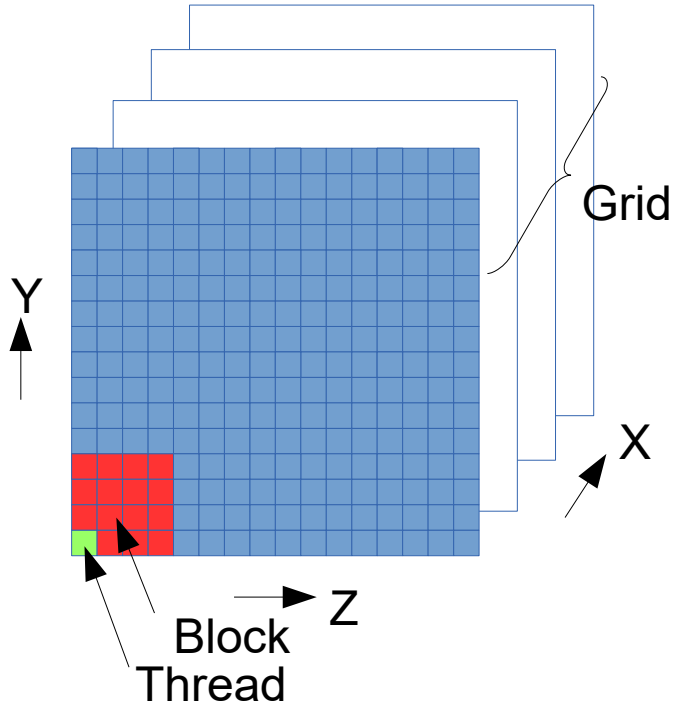
CUDA

並列化の必須条件

(1)並列化できるアルゴリズムであること
　　並列度がスケーラブルに大きくできる←FDTD法はそのようになっている

(2)CPU/GPU間のメモリー転送を最小限にする(律速にならないこと)
　　FDTD法では反復計算の途中でメモリー転送は不要

(3)スレッド番号順にメモリーにアクセスする(Coalescing of memory)
　　i→j→kループのときは配列のアドレスもi→j→kの順にする



FDTD法における
Thread,Block,Grid

CUDAプログラミングの作業手順

(1)最初に逐次版を作成し十分テストする。
　　（計算部にバグがあると並列化の作業効率が大きく下がる）

(2)計算の主要部をCPU/GPUから呼び出せるように書き換える。
　　（関数修飾子　”__device__ __host__”）

(3)カーネル関数（GPUで行う処理）とCPU/GPU間のメモリー転送を実装する。

表4 Ex成分を更新するプログラム（**CUDA版**）

```
1 __device__ __host__ void updateEx(
2   size_t n, size_t nj, size_t nk,
3   float *ex, float *hy, float *hz,
4   float c1, float c2, float ryn, float rzn) {
5   ex[n] = c1 * ex[n]
6         + c2 * (ryn * (hz[n] - hz[n - nj])
7                 - rzn * (hy[n] - hy[n - nk]));
8 }
```

アドレスnはBlock,Gridから次式で計算される。

```
int i = threadIdx.z + (blockIdx.z * blockDim.z);
int j = threadIdx.y + (blockIdx.y * blockDim.y);
int k = threadIdx.x + (blockIdx.x * blockDim.x);
size_t n = (i * ni) + (j * nj) + (k * nk) + n0;
```

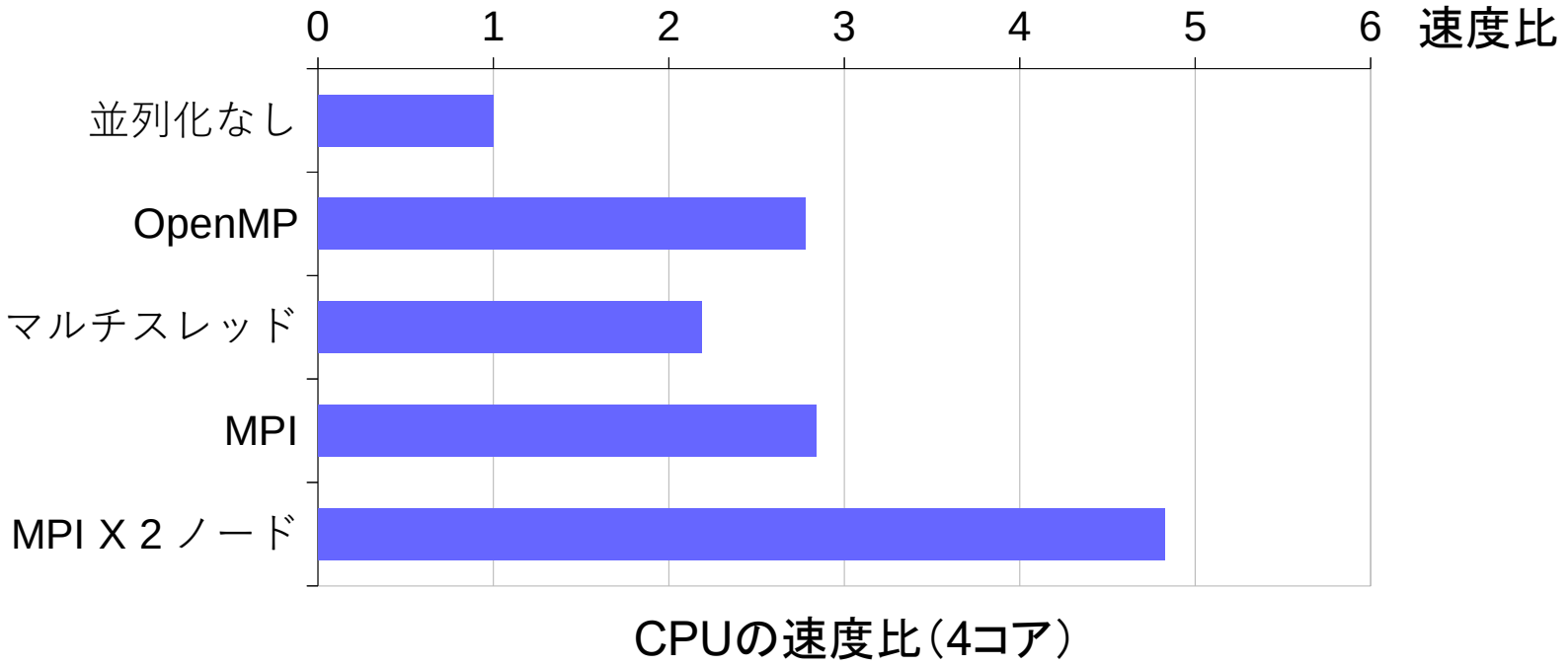
OpenCL

表5 Ex成分を更新するプログラム(OpenCL版)

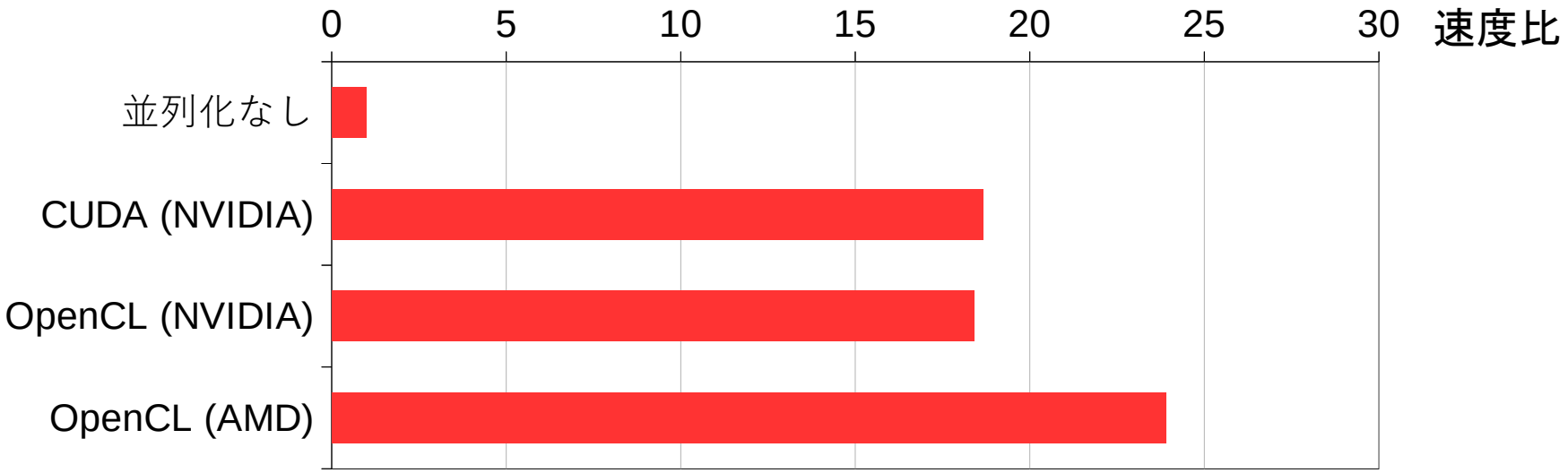
```
 1 #define LA(p,i,j,k) ((i)*((p)->ni)+(j)*((p)->nj)+(k)*((p)->nk)+((p)->n0))
 2 __kernel void updateEx(
 3  constant index_t *p,
 4  global float *ex, global float *hy, global float *hz, global unsigned char *iex,
 5  global float *c1, global float *c2, global float *ryn, global float *rzn) {
 6  int i = get_global_id(2);
 7  int j = get_global_id(1);
 8  int k = get_global_id(0);
 9  if ((i <  p->nx) &&
10      (j <= p->ny) &&
11      (k <= p->nz)) {
12      int n = LA(p, i, j, k);
13      ex[n] = c1[iex[n]] * ex[n]
14            + c2[iex[n]] * (ryn[j] * (hz[n] - hz[n - p->nj])
15                          - rzn[k] * (hy[n] - hy[n - p->nk]));
16  }
17 }
```

__kernel : カーネル関数(GPUで実行するコード)を表す関数修飾子
constant : contant memory (read専用、キャッシュされる)
global : global memory (通常の)デバイスメモリー
get_global_id : ワークアイテムの番号を取得する

ベンチマーク結果



CPUの速度比（4コア）



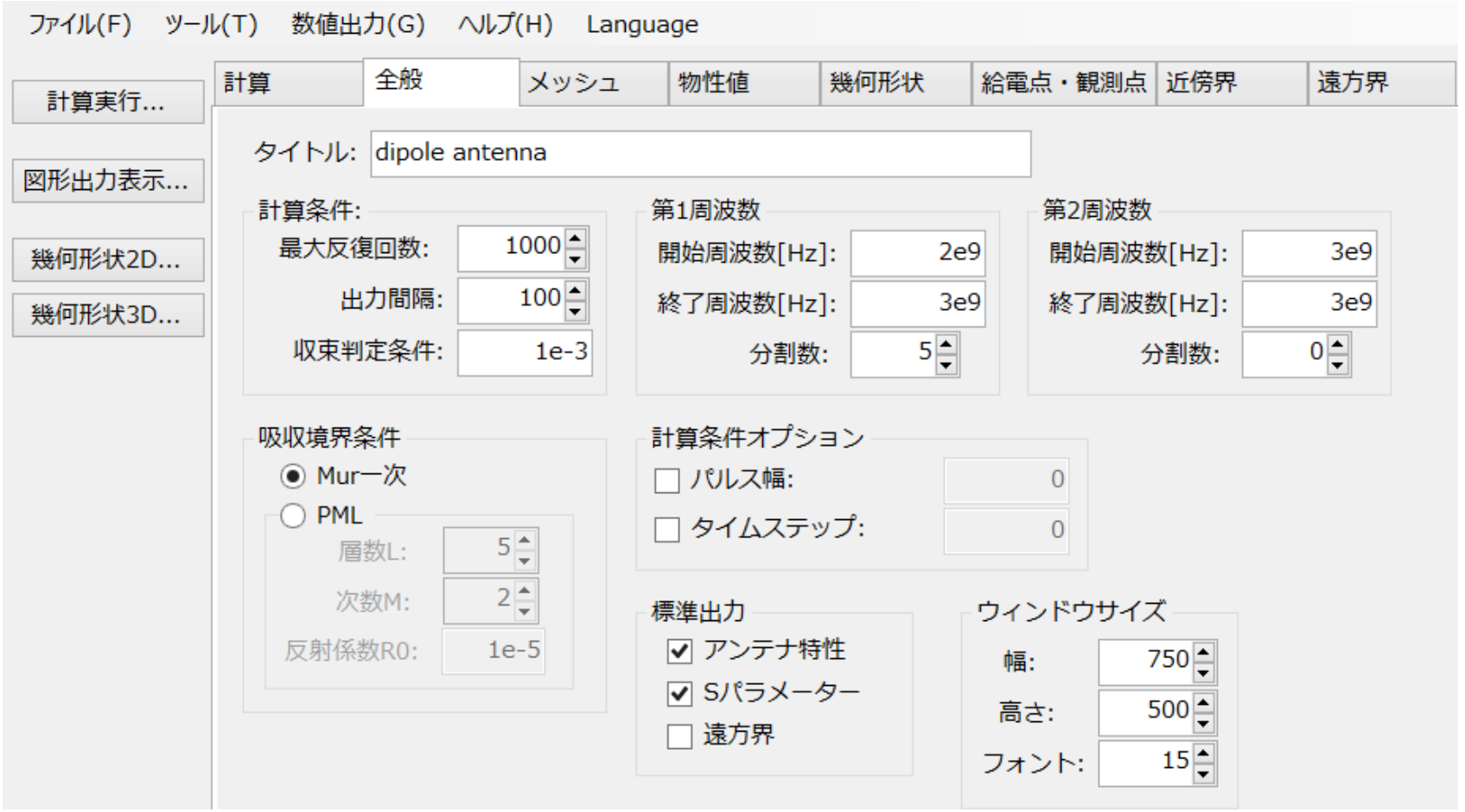
GPUの速度比

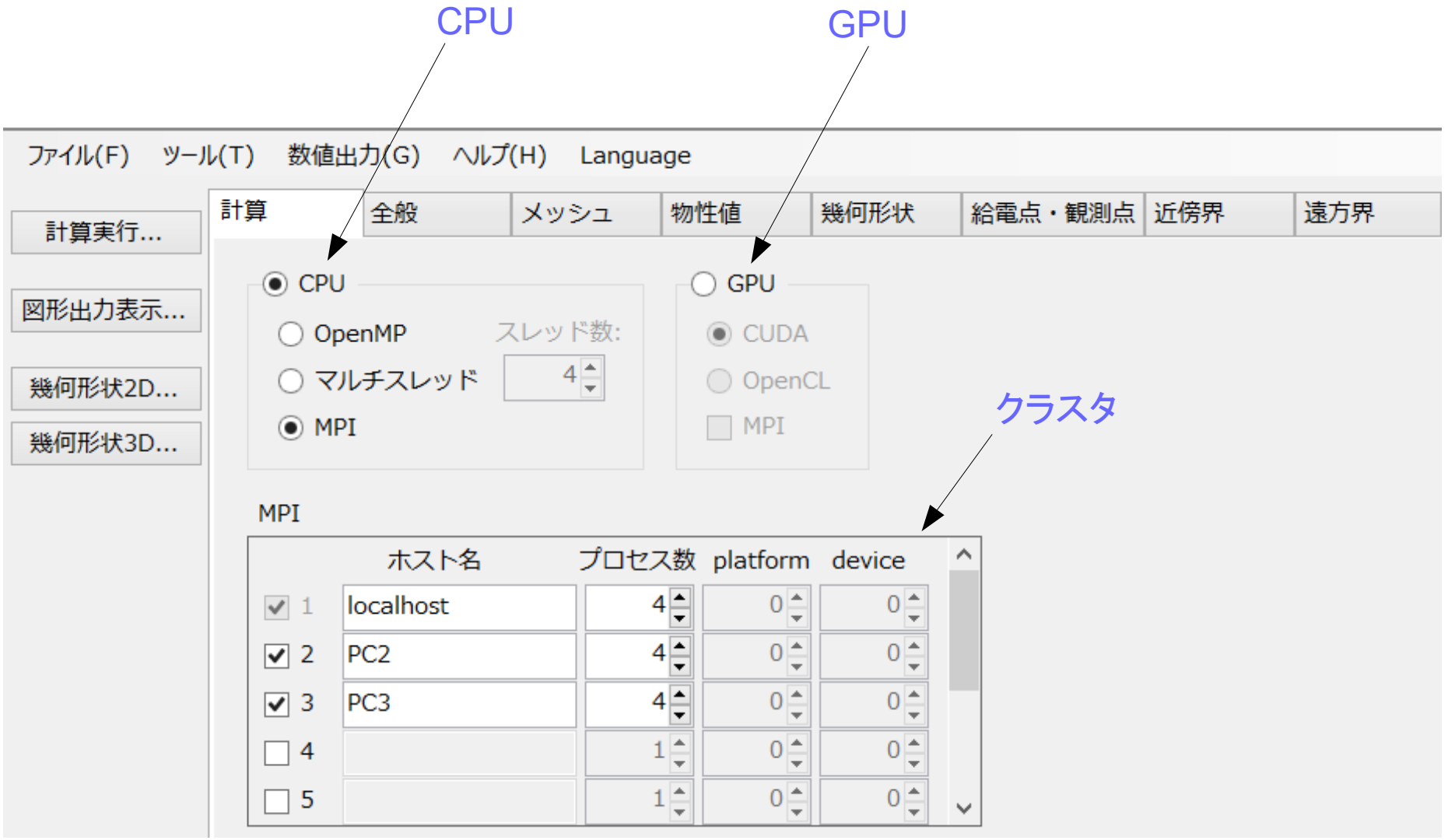
クラスタ使用時のネットワーク速度と並列化上限ノード数の関係

演算ハードウェア	ネットワーク	
	1Gbpsイーサネット	10Gbpsイーサネット/ InfiniBand
CPU	≲ 10	≲ 100
GPU	≲ 1	≲ 10

OpenFDTD

・オープンソース
・動作環境：Windows/Linux
・計算機能：
　　・アンテナ・伝送線路の周波数特性
　　・指定した周波数での近傍界と遠方界の分布図
・図形出力：HTML5のCanvas
・数値出力
・簡易GUI付属





計算ハードウェアの設定