一般社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
A・P2014-41(2014-6)

FDTD 法の並列化技術とオープンソース化

大賀 明夫

株式会社 EEM

E-mail: oga@e-em.co.jp

あらまし FDTD 法の各種並列化技術について説明し実装する。並列化技術として OpenMP、マルチスレッド 、MPI、CUDA 、OpenCL を取り上げる。プログラムはオープンソースとして公開する。

キーワード FDTD 法，並列化、OpenMP、MPI、CUDA、OpenCL

Parallelization of FDTD and its open source program

Akio OGA

EEM Inc.

E-mail: oga@e-em.co.jp

Abstract Several techniques parallelizing FDTD are presented. They include OpenMP, Multi-threading, MPI, CUDA and OpenCL. Programs are opened with source code.

Keyword FDTD, Parallelization, OpenMP, MPI, CUDA, OpenCL

1. はじめに

FDTD(Finite Difference Time Domain)法を用いた電磁界シミュレータは電磁界の各種の用途に広く用いられている。しかし問題の規模が大きくなりまた形状が複雑化すると計算時間が長くなり、その高速化が求められている。

本報告では高速化のために技術として並列化を取り上げる。各種手法の並列プログラムの実例を挙げ、プログラミング上の注意点や効率的な開発方法について述べる。最後にベンチマーク問題の計算時間を計測し各手法について考察する。

プログラムは OpenFDTD という名前のオープンソースとして公開する。[1]

2. 並列化技術

並列化の技術としては以下の手法がある。

(1) OpenMP [2]

これはプログラムの for 文の前に固有のディレクティブ(指示文)を記入し、コンパイラがその指示文をもとに並列プログラムを作成するものである。現在のほとんどの C/C++コンパイラは OpenMP をサポートしている。1 台のマルチコア、マルチ CPU のコンピュータで並列計算するときに使用する。この環境を共有メモリーと呼ぶ。

(2) マルチスレッド [3]

これは共有メモリー環境で複数のスレッドを生成し、計算処理を各スレッドに分割するものである。ユーザーはスレッド関数を起動しスレッド番号に応じた処理を記述する。マルチスレッドは OS とコンパイラが標準でサポートしている。(1)と比べてプログラミングの作業量は増えるが細かい並列処理にも対応できる。(1)もマルチスレッドの一種であるがプログラミング手法が異なるのでここでは別に扱う。

(3) MPI (Message Passing Interface) [4]-[7]

これは複数台のコンピュータで並列計算するものである。この環境を分散メモリーと呼ぶ。マルチコア、マルチ CPU による並列計算も含んでいる。処理の単位はプロセスとなりメモリー空間が独立しているので、プロセス間のデータのやりとりは MPI の関数を呼び出すことによって通信により行う。

(4) CUDA [8]-[10]

これは NVIDIA のグラフィックスボードの高い計算能力を汎用の科学技術計算に用いるものである(GPGPU : General-

Purpose computing on Graphics Processing Units)。計算処理を多数のスレッドに分割して並列計算を行う。CPU と GPU 間は関数を通してメモリー転送を行う。

(5) OpenCL [11]-[16]

これは NVIDIA または AMD のグラフィックスボードを用いて並列計算を行うものである。プログラミング手法は CUDA とほぼ同じであるが、各種のデバイスで動作することが特長である。

3. FDTD 法

Maxwell 方程式に構成方程式を代入して係数の次元を揃えると式(1)(2)となる。

$$\epsilon_r \frac{\partial \boldsymbol{E}}{c\,\partial t} = \nabla \times \eta \boldsymbol{H} - (\eta\sigma)\boldsymbol{E} \tag{1}$$

$$\mu_r \frac{\partial \eta \boldsymbol{H}}{c\,\partial t} = -\nabla \times \boldsymbol{E} - \left(\frac{\sigma_m}{\eta}\right)\eta \boldsymbol{H} \tag{2}$$

ここで、 $\boldsymbol{E}$:電界 $\boldsymbol{H}$:磁界 ϵ_r :比誘電率 μ_r :比透磁率 σ :導電率 σ_m :導磁率 $\eta = \sqrt{\frac{\mu_0}{\epsilon_0}}$:真空の波動インピーダンス $c = \frac{1}{\sqrt{\epsilon_0 \mu_0}}$:真空の光速である。

式(1)(2)を時間的に離散化すると次式が得られる。右上の n は時間ステップである。

$$\boldsymbol{E}^{n+1} = c_1 \boldsymbol{E}^{n} + c_2 (c\Delta t) \nabla \times \eta \boldsymbol{H}^{n+\frac{1}{2}} \tag{3}$$

$$\eta \boldsymbol{H}^{n+\frac{1}{2}} = d_1 \eta \boldsymbol{H}^{n-\frac{1}{2}} - d_2 (c\Delta t) \nabla \times \boldsymbol{E}^{n} \tag{4}$$

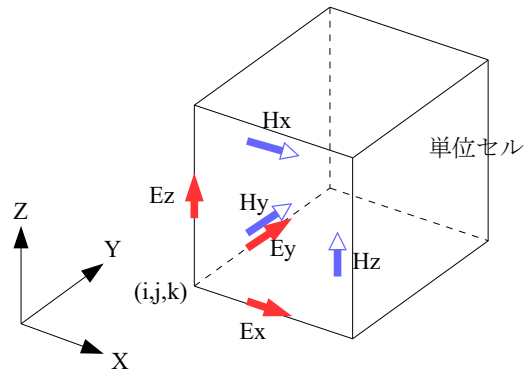

図 1 Yee 格子

ここで c_1, c_2, d_1, d_2 は次式で計算される無次元量である。

$$\left.\begin{aligned} c_1 &= \frac{\epsilon_r}{\epsilon_r + (\eta\sigma) c\Delta t} \\ c_2 &= \frac{1}{\epsilon_r + (\eta\sigma) c\Delta t} \end{aligned}\right\} \tag{5}$$

$$\left.\begin{aligned} d_1 &= \frac{\mu_r}{\mu_r + \left(\frac{\sigma_m}{\eta}\right) c\Delta t} \\ d_2 &= \frac{1}{\mu_r + \left(\frac{\sigma_m}{\eta}\right) c\Delta t} \end{aligned}\right\} \tag{6}$$

式(3)の X 成分を図 1 の Yee 格子で空間的に離散化すると次式が得られる。[17]

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

適当な波源のもとで式(7)とその他の成分を反復計算することによって電磁界の時間波形が求まり、それを Fourier 変換することにより周波数領域の電磁界が得られる。

4. FDTD 法プログラム

電界の X 成分を更新する式(7)のプログラムは以下のようになる。

表 1 Ex 成分を更新するプログラム

```
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 }
```

ループは外側から i,j,k の順にとる。従って電磁界配列のアドレスも外側から i,j,k の順に配置する必要がある。係数(C1,C2)は物性値番号を通して間接的に参照する。物性値の種類を256個までに限定すると1バイトで表せる。このとき全体の必要メモリーは30NxNyNzバイトになる。表1では左辺の変数への書き込みがループ間で競合しないために3重ループはどのように並べ替えても結果が変わらない。この性質のために FDTD 法は極めて並列計算向きのアルゴリズムになっている。

5. OpenMP

OpenMP では並列化可能な for 文の直前に指示文（#pragma omp で始まる）を代入し、コンパイラの自動並列化機能を利用する。表2に表1を OpenMP 対応にしたプログラムを示す。違いは4行目のみである。ここで private は括弧内の変数を各スレッドで独自に持ちスレッド間で共有しないことを示す。なお、並列化するループは3つのうちどれでもよいが一番粒度（単位スレッドの計算量）の大きい外側のループが最も効率がよい。このように FDTD 法は OpenMP を用いると簡単に並列化することができる。

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 }
```

6. マルチスレッド

表3にマルチスレッドのプログラムを示す。ループの中は表1と同じであり省略する。スレッド関数には void*型の引数を1つ指定することができる。スレッド番号以外の変数を渡すには構造体に入れるかグローバル変数を使用する。スレッドを起動/終了する関数は、Windows では CreateThread / WaitForMultipleObjects、Linux では pthread_create / pthread_join である。

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++) {
```

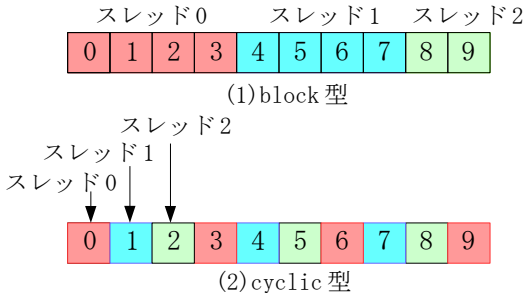

図2 2種類のループ分割法（ループ長=10, スレッド数=3）

ループを分割するには図2の2種類があるが、表3では他の並列化に合わせて block 型を用いている。cyclic 型にするには9-13行目が以下の一行になる。

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

ここで変数 tid はスレッド番号、nThread はスレッド数である。なお、10 行目は繰り上げ商を計算するものであり並列計算でよく用いられる定型句である。

7. MPI

ここでは計算領域を X 方向に分割する。各プロセスの i の下限と上限を iMin,iMax とすると表 1 の 5 行目が以下に変わるだけである。

```
for (i = iMin; i < iMax; i++) {
```

計算領域がプロセスに分割されるのでプロセス数を増やせば計算できるサイズに上限はない。

タイムステップごとに図 3 の通り不足する成分を隣のプロセスから通信により取得する。不足する成分は領域境界の半セル外側の Hy,Hz 成分である。通信には MPI_Sendrecv 関数を用いる。通信量は 8NyNz バイトである。なお、領域境界上の Ey,Ez,Hx 成分は両プロセスで重複して持ち重複して計算する。

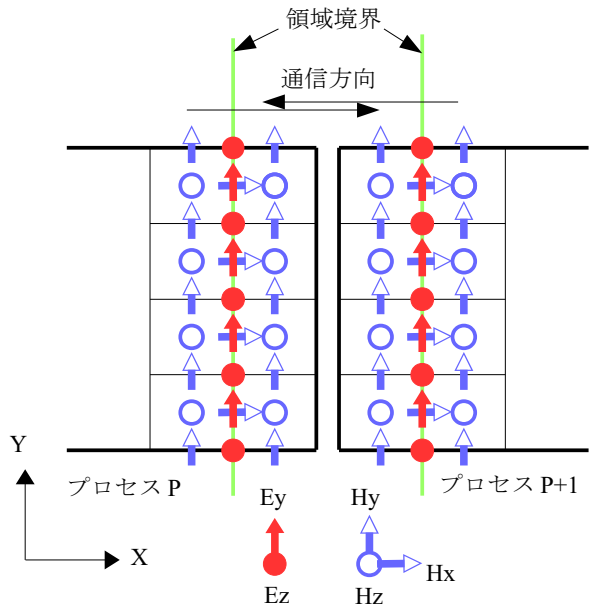

図 3 MPI の通信

8. CUDA

CUDA では表 1 の 3 重ループを多数のスレッドに分割し並列計算する。ここでは Block=(32,8,1)=256 スレッド、Grid=(Nz/32,Ny/8,Nx)（正確には繰り上げ商）とする。CUDA ではスレッドの順に変数にアクセスすることが必須(coalescing of memory[10])であるためにループは外側から i,j,k とする。このようにすると変数のアドレスの並びはこれまで述べた CPU 版と同じでよい。

CUDA では CPU/GPU 間のメモリー転送を少なくすることが大切であるが、FDTD 法では反復計算の間はすべて GPU メモリー(デバイスメモリー)内で計算できるので都合がよい。表 4 に CUDA プログラムを示す。

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 }
```

ここでアドレス 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;
```

CUDA プログラミングの作業手順は以下のようになる。

(1)逐次版を作成し十分テストする。

(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。表 4 の関数修飾子”__device__ __host__”がこれを表している。

(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。

(2)で計算処理を CPU/GPU で共通化することにより、(3)の作業が形式的なもので済み作業効率が上がる。

9. OpenCL

OpenCL のアーキテクチャは CUDA とほぼ同じであるからプログラム設計法は CUDA と同じである。表 5 に OpenCL プログラムを示す。

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 }
```

以下、OpenCL 固有の注意点を挙げる。

(1)OpenCL はいろいろな環境で動かすためにやや複雑な前処理が必要であるが、定型的な処理なのでモジュール化して計算部と切り離しておく。

(2)オンラインコンパイルを使用するときはカーネルコードは"Intel Kernel Builder"[12]を用いて文法をチェックしておく。カーネルコードでは#include は使えない（#define は可）。

(3)OpenCL では CUDA と異なり CPU/GPU でコードを共通化することができないので、CPU コードを GPU コードにコピーするときに機械的な作業で済むように CPU コードを十分チューニングしておく。

(4)カーネルに引数を渡すには clSetKernelArg 関数を使用し、カーネルを起動するには clEnqueueNDRangeKernel 関数（データ並列）を使用する。

(5)ワークアイテムの設定とパフォーマンスは CUDA と同じである。

10. ベンチマーク

以上で述べたプログラムの並列化の性能を評価するために図 4 のベンチマーク問題を計算する。誘電体ブロックの上にグラウンド板とモノポールアンテナが乗っているモデルである。計算条件は以下の通りである。

・セル数：200 x 200 x 200

・タイムステップ数：2000

・周波数：20GHz

・吸収境界条件：Mur 一次

計算環境は以下の通りである。

・CPU-1：Intel Core i7-4770K, 4 コア, 4.0GHz(OC)

・CPU-2：Intel Core i5-2500K, 4 コア, 4.0GHz(OC)

・GPU-1：NVIDIA GeForce GTX 670, 1344 コア, 4GB

・GPU-2：AMD Radeon R9 270, 1280 コア, 2GB

・OS：Windows7 64 ビット

・コンパイラ：Microsoft Visual Studio 2012

・ネットワーク：ギガビットイーサネット

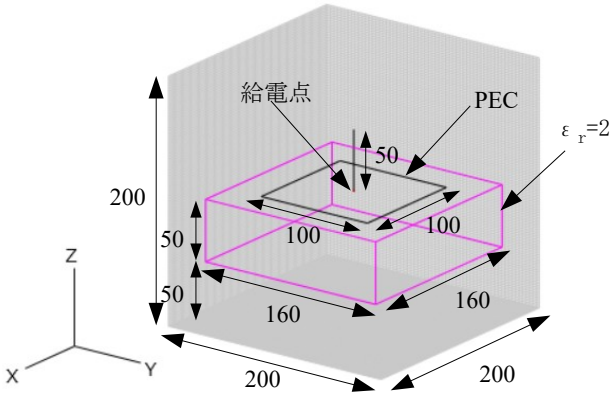

図 4　ベンチマーク問題（単位：mm）

計算時間は表 6 の通りである。

表 6 計算時間

No.	ハードウェア	並列化手法	計算時間	速度比
1	CPU-1	並列化なし	399.1 秒	1
2	CPU-1	OpenMP	143.6 秒	2.78
3	CPU-1	スレッド	182.4 秒	2.19
4	CPU-1	MPI	140.6 秒	2.84
5	CPU-1+CPU-2	MPI	82.7 秒	4.83
6	GPU-1	CUDA	21.4 秒	18.6
7	GPU-1	OpenCL	21.7 秒	18.4
8	GPU-2	OpenCL	16.7 秒	23.9

No.2-4 は CPU による並列化であり、4 コアで 3 倍弱速くなる。その中で No.3 のマルチスレッドが比較的遅いがスレッド起動のオーバーヘッドが比較的大きいことによる。No.5 は 2 台によるクラスタであり、1 台に比べて 2 倍弱速くなる。No.6-8 は GPU による並列化であり、No.1 に比べて 18-24 倍速くなるなお No.6-7 より同じ GPU では CUDA と OpenCL の性能はほぼ同じである。

11. オープンソース化

本プログラムは OpenFDTD という名前でオープンソースとして公開している[1]。動作環境は Windows または Linux である。主な計算機能としては、アンテナと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などがある。計算結果は数値出力と図形出力が行われる。図形出力は HTML5 の Canvas 形式であり Web ブラウザで見ることができる。本プロプラムはエディタで入力データを作成し、コマンドラインで計算を実行することを前提としているが、Windows 用の簡易 GUI も添付している(図 5)。また、複雑な入力データをプログラミングで作成するためのツール「データ作成ライブラリー」も付属している。

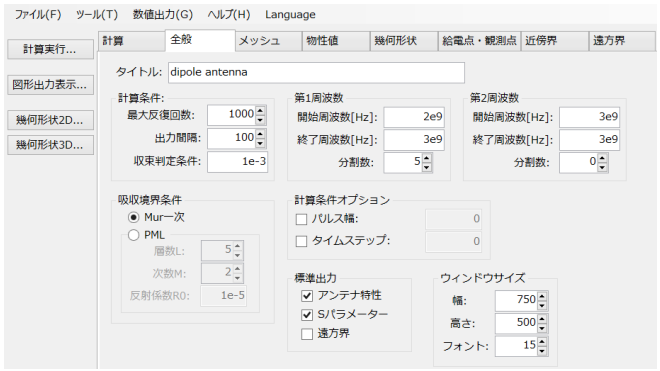

図 5 GUI プログラム

12. まとめ

FDTD 法を並列化するために、OpenMP、マルチスレッド 、MPI、CUDA、OpenCL について述べた。FDTD 法は並列化向きのアルゴリズムであり、並列化することによって計算時間を短縮することができる。MPI を用いるとクラスタまたはスーパーコンピュータで大規模な計算を行うことができる。また 、GPU を用いると安価で高速な計算環境を構築することができる。なお、GPU についても MPI を用いてクラスタ化することができるが、ギガビットイーサネットでは性能は出ず、より高速なネットワーク環境が必要になる。[1]

文 献

[1] 株式会社 EEM、"OpenFDTD"、
http://www.e-em.co.jp/OpenFDTD/

[2] OpenMP.org, http://openmp.org/wp/

[3] 株式会社フィックスターズ、マルチコア CPU のための並列プログラミング、秀和システム、2006.

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

[5] P.パチェコ（秋葉博訳）、MPI 並列プログラミング、培風館、2001.

[6] Open MPI, http://www.open-mpi.org/

[7] Microsoft, "HPC Pack 2012 MS-MPI",
http://www.microsoft.com/ja-jp/download/details.aspx?id=36045

[8] NVIDIA, "CUDA Downloads",
https://developer.nvidia.com/cuda-downloads/

[9] NVIDIA, "CUDA C Programming Guide",
http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[10] NVIDIA, "CUDA C Best Practices Guide",
http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/

[11] Khronos group, "OpenCL", https://www.khronos.org/opencl/

[12] Intel, "Intel SDK for OpenCL Applications 2013",
http://software.intel.com/en-us/vcsource/tools/opencl-sdk/

[13] NVIDIA, "OpenCL | NVIDIA Developer Zone",
https://developer.nvidia.com/opencl/

[14] AMD, "OpenCL Zone | AMD",
http://developer.amd.com/tools-and-sdks/opencl-zone/

[15] 株式会社フィックスターズ、改定新版 OpenCL 入門、インプレスジャパン、2012.

[16] 奥薗隆司、OpenCL 入門、秀和システム、2010.

[17] 宇野亨、FDTD 法による電磁界およびアンテナ解析、コロナ社、1998.