電磁界シミュレーターOpenFDTDのSX-Auroraでの性能評価

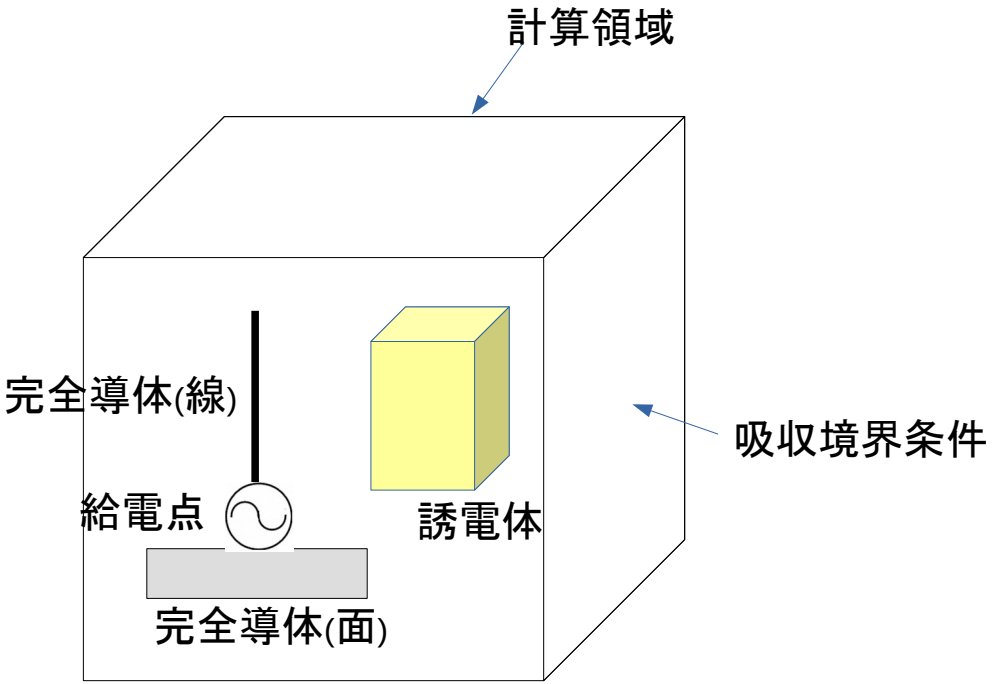
2019年11月7日

株式会社EEM　大賀明夫

電磁界シミュレーターOpenFDTD [1]

・FDTD法（時間領域差分法）[2]による汎用的な電磁界シミュレーター
・オープンソース（フリーソフト）
・Windows環境では実行プログラム付き（簡易GUI付き）
・Linux環境ではソースコードからコンパイルする
・図形出力はHTML（OSによらない）
・各種の高速化技術を利用し、大規模問題の計算を想定している
・使用言語：C
・2014年5月初版リリース

電磁界シミュレーション



電磁界シミュレーションのイメージ
（計算領域内の電磁界分布を計算する）

FDTD法とは

マックスウェル方程式（電磁気の基本法則）

$$\nabla \times \mathbf{H} = \epsilon \frac{\partial \mathbf{E}}{\partial t} + \sigma_e \mathbf{E}$$

$$-\nabla \times \mathbf{E} = \mu \frac{\partial \mathbf{H}}{\partial t} + \sigma_m \mathbf{H}$$

E:電界

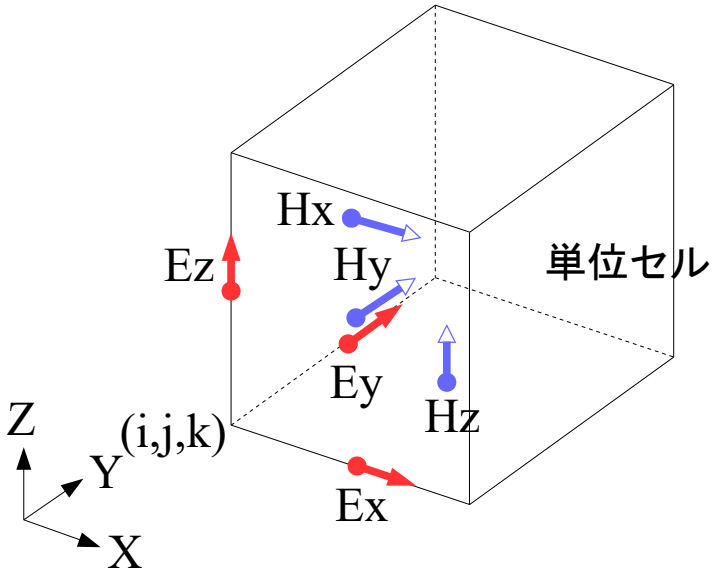
H:磁界

ε:誘電率

μ:透磁率

σ_e:導電率

σ_m:導磁率



計算領域を多数のセルに分割し

Yee格子上で時間領域と空間領域で離散化する

Yee格子

（マックスウェル方程式と相性がよい）

FDTD法の離散化

Exの離散化

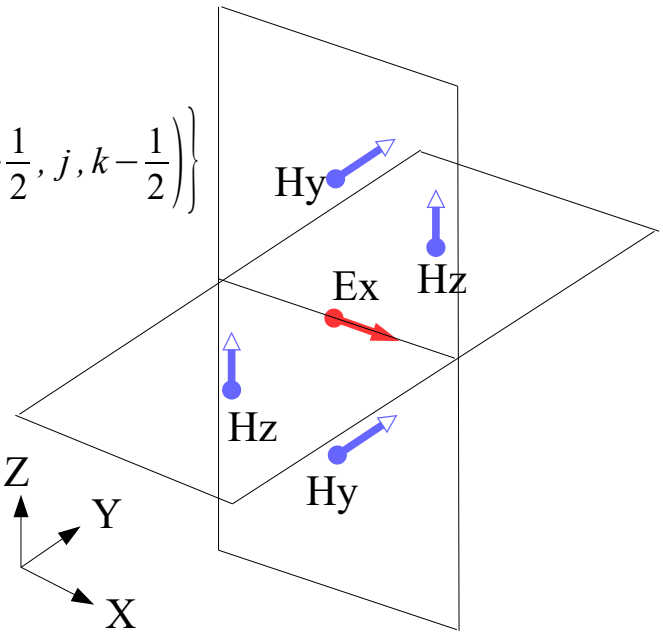
$$E_x^{n+1}\left(i+\frac{1}{2},j,k\right)=c_1 E_x^n\left(i+\frac{1}{2},j,k\right)$$

$$+c_{2y}\left\{H_z^{n+\frac{1}{2}}\left(i+\frac{1}{2},j+\frac{1}{2},k\right)-H_z^{n+\frac{1}{2}}\left(i+\frac{1}{2},j-\frac{1}{2},k\right)\right\}-c_{2z}\left\{H_y^{n+\frac{1}{2}}\left(i+\frac{1}{2},j,k+\frac{1}{2}\right)-H_y^{n+\frac{1}{2}}\left(i+\frac{1}{2},j,k-\frac{1}{2}\right)\right\}$$

Hxの離散化

$$H_x^{n+\frac{1}{2}}\left(i,j+\frac{1}{2},k+\frac{1}{2}\right)=d_1 H_x^{n-\frac{1}{2}}\left(i,j+\frac{1}{2},k+\frac{1}{2}\right)$$

$$-d_{2y}\left\{E_z^n\left(i,j+1,k+\frac{1}{2}\right)-E_z^n\left(i,j,k+\frac{1}{2}\right)\right\}+d_{2z}\left\{E_y^n\left(i,j+\frac{1}{2},k+1\right)-E_y^n\left(i,j+\frac{1}{2},k\right)\right\}$$



Exの離散化

n:タイムステップ

c_1,c_2,d_1,d_2:場所の関数（既知量）

OpenFDTDのデータ作成法

(1) テキストファイル編集

・エディタを用いてテキストファイルを編集する

・複雑なデータを入力することが難しい

(2) GUI

・Windows環境のGUIでデータを作成し、Linux環境に転送する

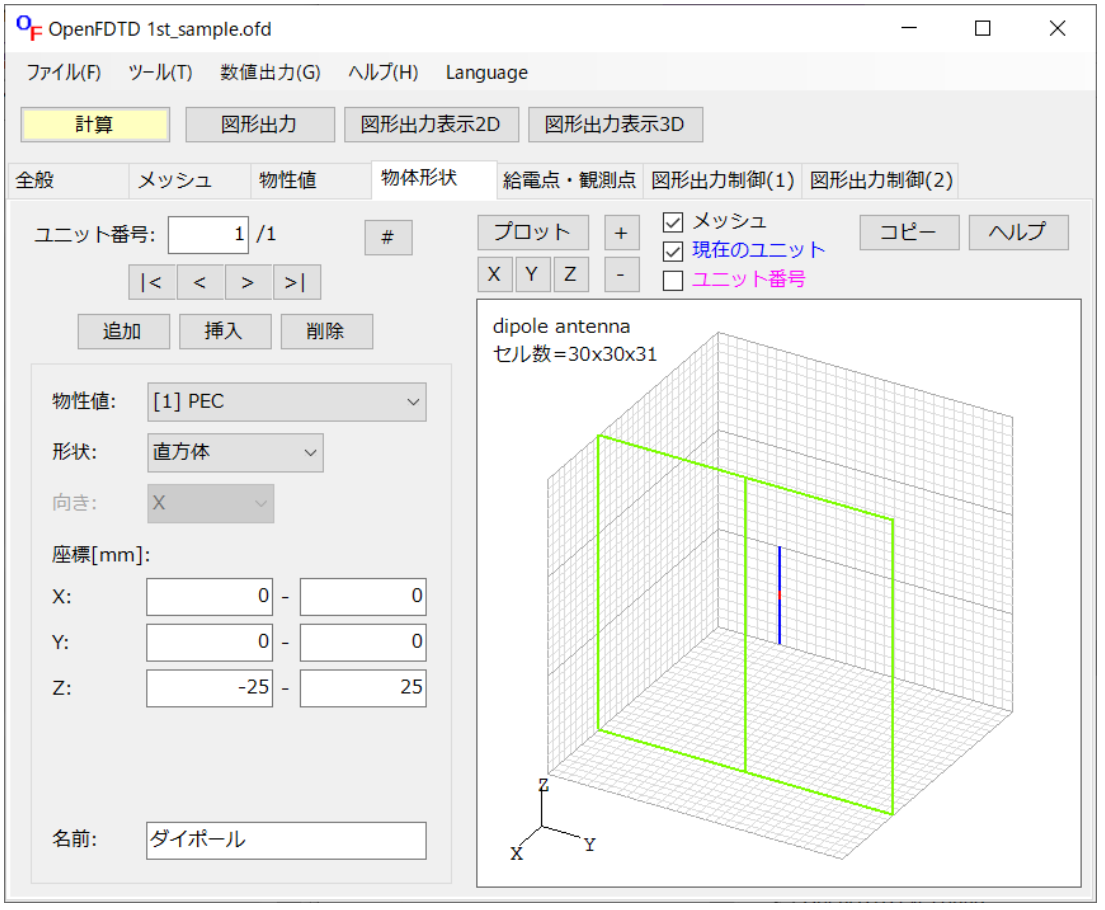
・データ入力が容易であるが、データ量が増えると作業量も増える

(3) プログラミング

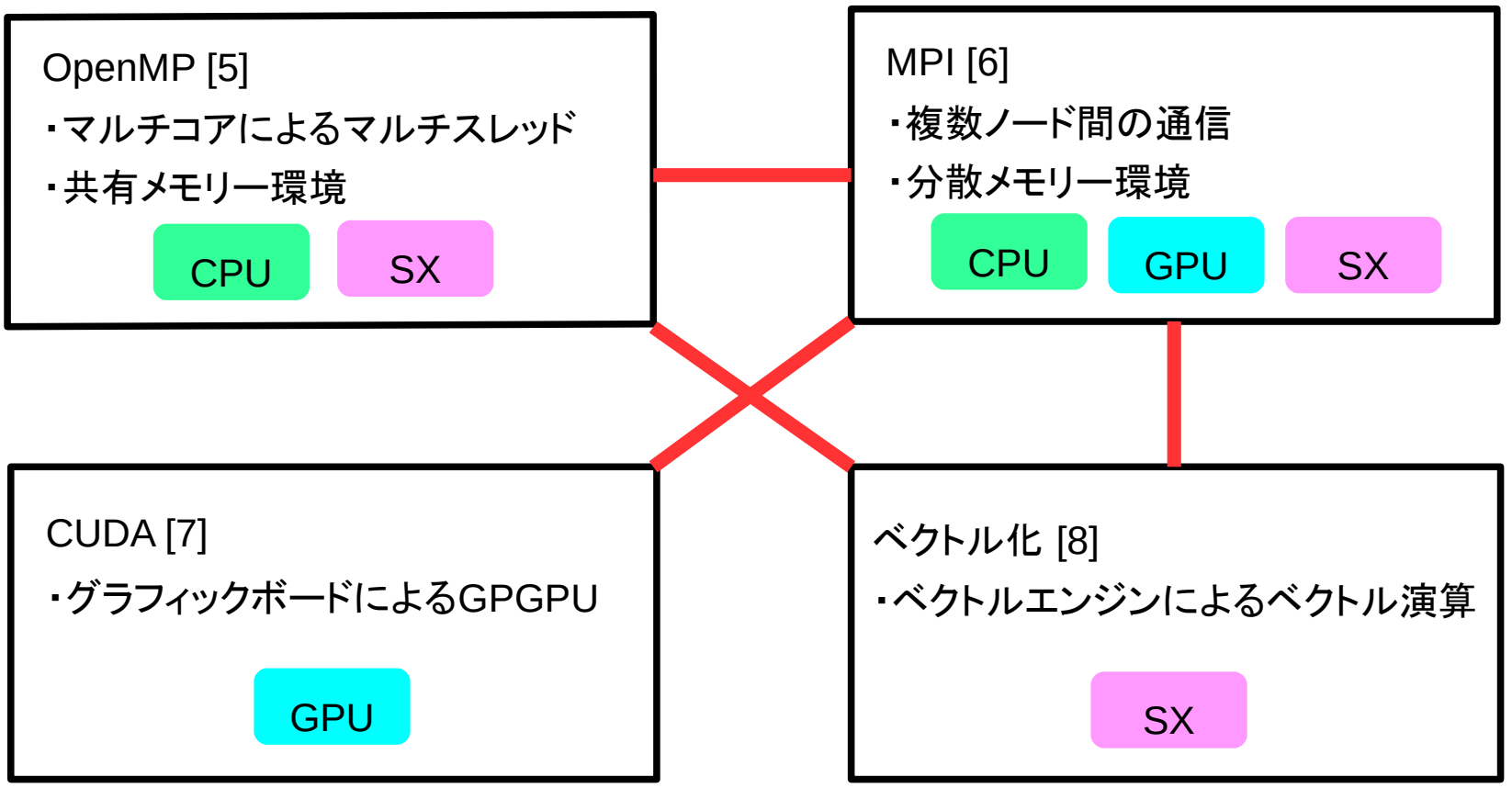
・OpenFDTDに同梱されている「データ作成ライブラリー」を使用してデータを作成するプログラムを作る

・プログラミングが必要であるが、大量のデータやパラメーターを変えながら繰り返し計算することに向いている

GUI（Graphical User Interface:Windows環境）[3]

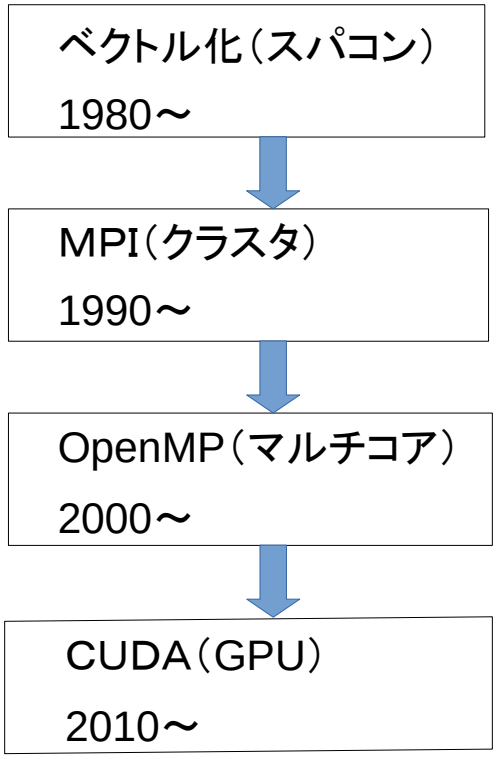


OpenFDTDの高速化技術 [4]

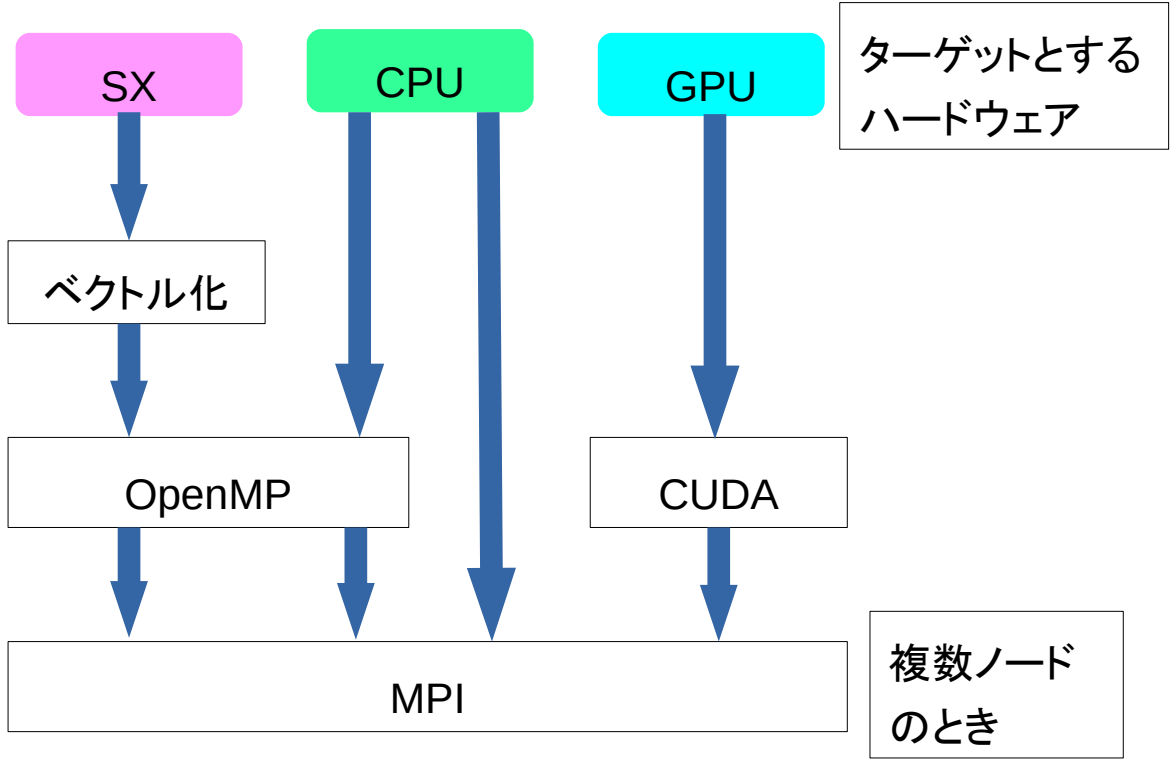


高速化技術の流れ

高速化技術の歴史



プログラミング作業の流れ



プログラミングの作業量

OpenMP  並列化は容易

MPI  プログラムの大幅な書き換えが必要になる

CUDA  プログラムの大幅な書き換えが必要になる

ベクトル化  自動ベクトル化が可能なアルゴリズムでは作業は簡単であるが、そうでないときはプログラムの大幅な書き換えが必要になる

※すべて高速化可能なアルゴリズムであることが前提

FOCUSスパコン [9]

FOCUSスパコンとは

・公益財団法人計算科学振興財団の運営するスパコンの時間貸しサービス

・産業界にスパコンを提供し、日本のものづくりに役立てる

システム (略称)	ハードウェア	コア数 (ノード当たり)	メモリー (ノード当たり)	ノード間ネットワーク	総ノード数	使用料金 1ノード1時間 (令和1年度)
F (CPU-1)	Intel Xeon E5-2698v4 × 2 (Broadwell)	40	128GB	InfiniBand 56Gbps	62	500円
F (GPU)	NVIDIA Tesla P100 × 1	3584	16GB	InfiniBand 56Gbps	2	500円
V (CPU-2)	Intel Xeon Gold 6148 × 1 (Skylake)	20	96GB	InfiniBand 56Gbps	2	300円
V (SX)	NEC SX-Aurora TSUBASA Type10B	8	48GB	InfiniBand 56Gbps	2	300円

SX-Aurora TSUBASAの評価を行うことができる

FOCUSスパコンの使い方

(1) アカウント作成
・アカウント料：年間1万円
・コンパイル作業と6分以内のジョブは無料
(2) 必要なソフトのインストール
・putty: 端末、鍵作成
・WinSCP: ファイル転送
(3) OpenFDTD
・zipファイルを展開してOpenFDTDフォルダに置く
(4) コンパイル

```
$ ssh fvpu1
$ module load PrgEnv-nec
$ module load MPI-nmpi
$ cd ~/OpenFDTD/src
$ make -f Makefile_ncc clean; make -f Makefile_ncc
$ cd ../mpi
$ make -f Makefile_ncc clean; make -f Makefile_ncc
```

（共用領域にある実行プログラムを使用するときはコンパイル作業は不要）
(5) スクリプトファイルを作成して実行する
$ sbatch スクリプトファイル

1ノードで実行するときのスクリプトファイル例

```
#!/bin/bash
#SBATCH -p v024h
#SBATCH -t 30
#SBATCH -J ncc
#SBATCH -o stdout.%J
./ofd_ncc -n 8 data/benchmark/benchmark100.ofd
```

複数ノードで実行するときのスクリプトファイル例

```
#!/bin/bash
#SBATCH -p v024h
#SBATCH -N 2
#SBATCH -n 2
#SBATCH -t 30
#SBATCH -J ncc_mpi
#SBATCH -o stdout.%J
module load PrgEnv-nec
module load MPI-nmpi
NODEFILE=`generate_pbs_nodefile`
mpiexec -machinefile $NODEFILE ./ofd_ncc_mpi -n 8
data/benchmark/benchmark100.ofd 2> /dev/null
```

高速化プログラミングのための開発環境

()内はコンパイルコマンド

コンパイラーはすべて無料

	必要なハードウェア	Windows	Linux
CPU	CPUなら何でもよい （現在のCPUはすべてマルチコア,MPIのテストも可）	Microsoft Visual Studio Community (cl, OpenMP含む) MS-MPI	GCC (gcc, OpenMP含む) OpenMPI (mpicc)
GPU	NVIDIAのグラフィックスボード（中位機種でよい）	CUDA (nvcc) MS-MPI	CUDA (nvcc) OpenMPI
SX	FOCUSスパコン	N.A.	NEC C (ncc, OpenMP含む) NEC MPI (mpincc)

高速化作業に高い費用は不要

高速化に成功したら必要に応じてハードウェアを買う

SX-Aurora TSUBASAでの作業の流れ

(0) 準備
・アルゴリズムがベクトル化と並列化に適していることを確認する。
・必要ならコードを書き換える。
・これらの作業は経験が必要なので不明なときは省略する。

(1) ベクトル化
・コンパイラーのメッセージから自動ベクトル化されていることを確認する（ベクトル化の対象は一番内側のfor文）
・ベクトル化可能なのに自動ベクトル化されないときはプログラマの責任でfor文の前に”#pragma _NEC ivdep”を入れる。ただし性能はあまり期待できないので、できれば自動ベクトル化できるようにコードを書き換える。
・ベクトル長が256なのでfor文の長さが256より十分大きいことが望ましい。

(2) OpenMP並列化
・OpenMPは粒度が大きい方が望ましいので一番外側のfor文に”#pragma omp parallel for”を記入する。
・コンパイラーのメッセージで並列化されていることを確認する。
・ベクトルエンジンは8コアなのでfor文の長さが8より十分大きいことが望ましい。

(3) MPI並列化
・複数ノードで実行するときはMPIを用いて問題を分割し、MPI関数による通信を実装する。

高速化作業とプログラムの構成

(1) コア部

・全体の計算時間の90数%を占める
・この部分をベクトル化と並列化することが絶対条件
・OpenFDTDでは電磁界の更新部分

(2) 周辺部

・全体の計算時間の数%を占める（入力データによる）
・コア部を高速化するとこの部分が無視できなくなる（SXはシリアル性能が遅いので特に）
・最低でもベクトル化と並列化のどちらかが必要
・OpenFDTDでは吸収境界条件、DFT（離散フーリエ変換）、収束判定など

(3) その他

・計算時間<<1%
・チューニング不要

電磁界更新部のソースコードとコンパイル

ソースコード（updateEx.c：Ex更新部）

```
 1  #include "ofd.h"
 2
 3  void updateEx()
 4  {
 5       int i;
 6  #ifdef _OPENMP
 7  #pragma omp parallel for
 8  #endif
 9       for (    i = 0; i <  Nx; i++) {
10       for (int j = 0; j <= Ny; j++) {
11            int64_t n = (i) * Ni + (j) * Nj + (0) * Nk + N0;
12            int64_t n1 = n - Nj;
13            int64_t n2 = n - Nk;
14            for (int k = 0; k <= Nz; k++, n++, n1++, n2++) {
15  #ifdef __NEC__
16                 Ex[n] = K1Ex[n] * Ex[n]
17                       + K2Ex[n] * (RYn[j] * (Hz[n] - Hz[n1])
18                       - RZn[k] * (Hy[n] - Hy[n2]));
19  #else
20                 int m = iEx[n];
21                 Ex[n] = C1[m] * Ex[n]
22                       + C2[m] * (RYn[j] * (Hz[n] - Hz[n1])
23                                - RZn[k] * (Hy[n] - Hy[n2]));
24  #endif
25            }
26       }
27       }
28  }
```

コンパイルオプション-fopenmpをつけるとマクロ_OPENMPが定義されOpenMPの指示文が有効になる

並列化

ベクトル化

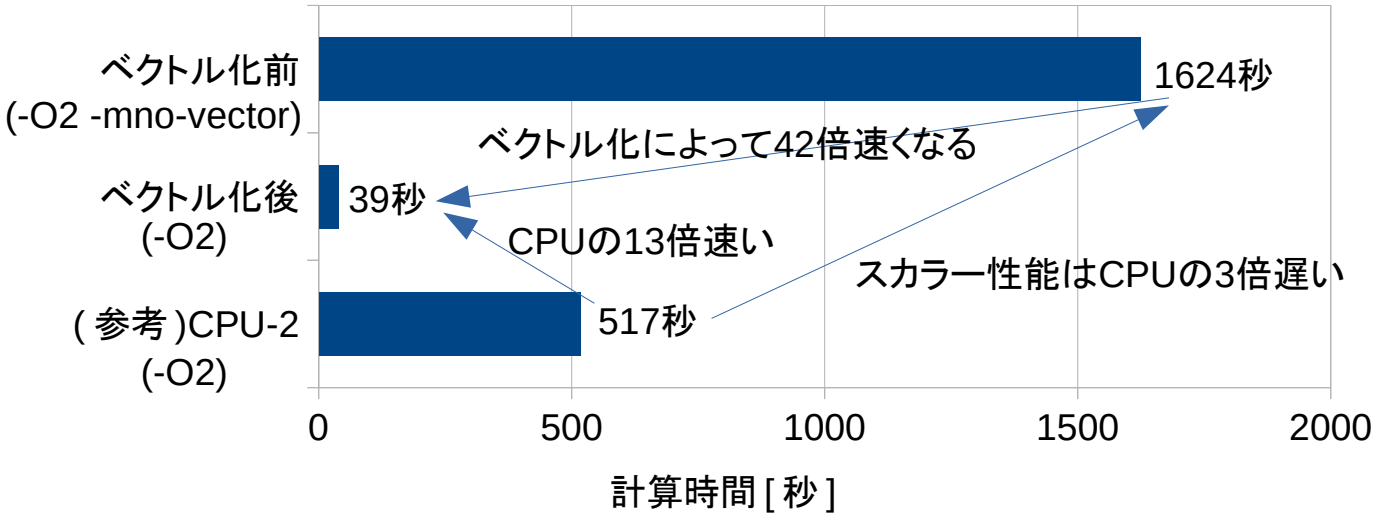
コンパイル結果

```
$ ncc -c -O2 -fopenmp -I../include -Wall updateEx.c
ncc: par(1801): updateEx.c, line 7: Parallel routine
generated.: updateEx$1
ncc: par(1803): updateEx.c, line 9: Parallelized by "for".
ncc: vec( 103): updateEx.c, line 10: Unvectorized loop.
ncc: vec( 101): updateEx.c, line 14: Vectorized loop.
```

SX用コード（nccではマクロ__NEC__が定義される）
通常コードの中間変数mがベクトル化の妨げになるので変数K1Ex,K2Exを新たに作成する
→必要メモリーが12/5=2.4倍になる

それ以外（通常のCPU）のコード

ベクトル化の効果

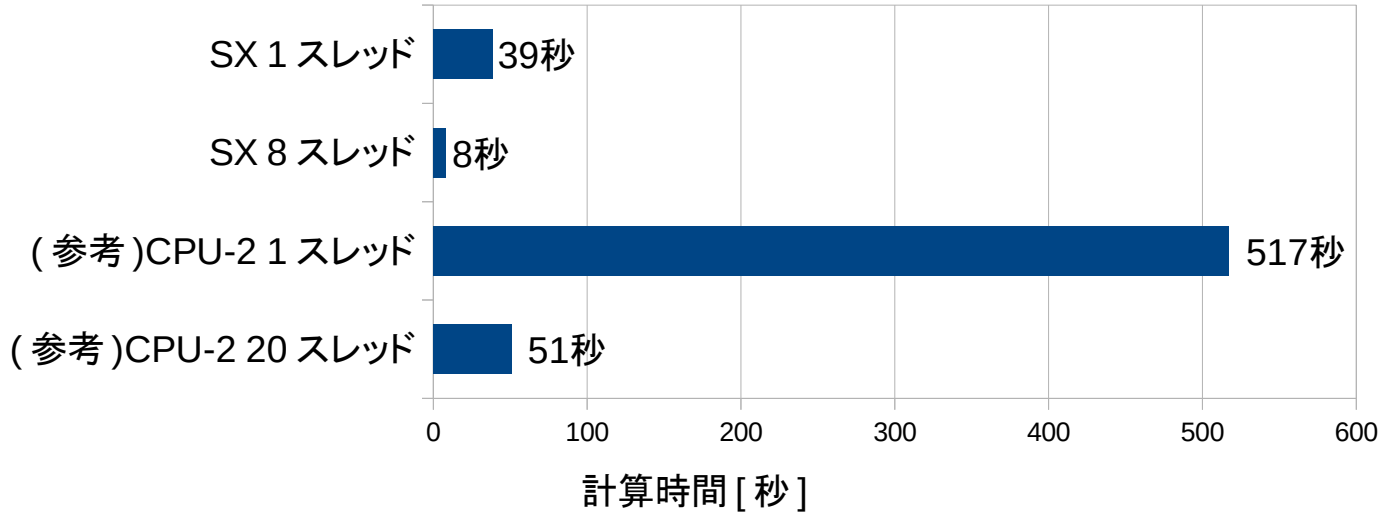


・コア部の計算時間

・()内はコンパイルオプション

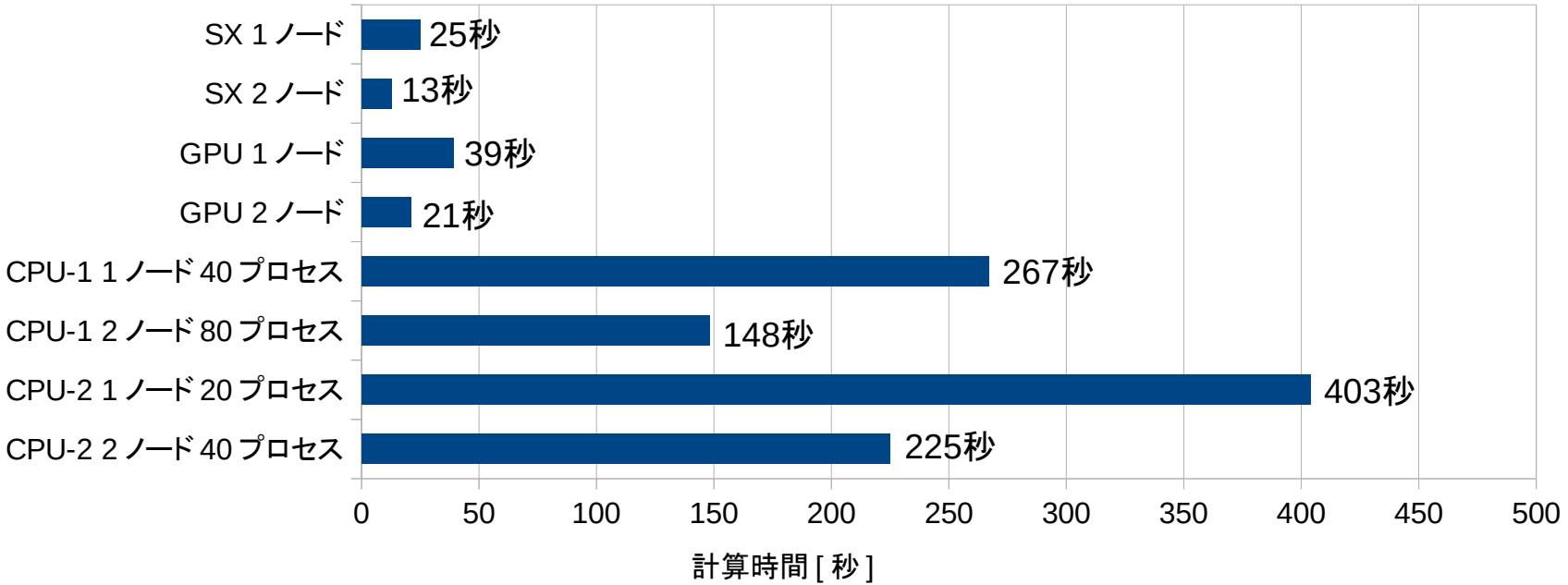
・計算条件：1スレッド共通、Nx=Ny=Nz=300、1000step

OpenMP並列化の効果



- SXはOpenMP並列化で5倍速くなる
- CPUはOpenMP並列化で10倍速くなる（コア数による）
- 計算条件：Nx=Ny=Nz=300、1000step、1ノード

MPI並列化の効果（SX/GPU/CPUの比較）



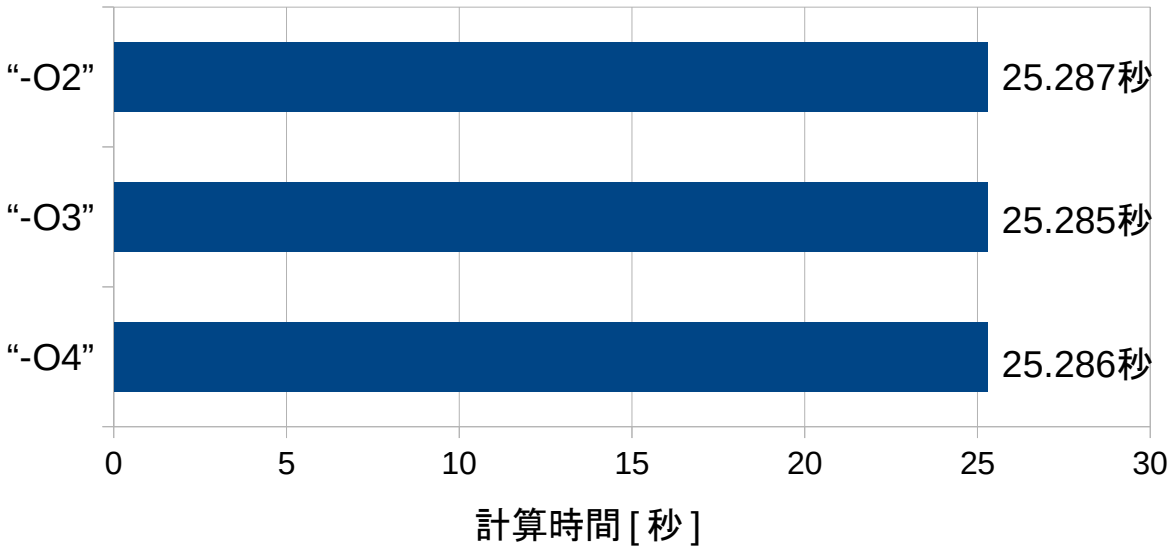
・SXはGPUの1.5倍速い

・SXはCPUの10～15倍速い（CPUの構成による）

・2ノードで2倍近く速くなる

・計算条件：Nx=Ny=Nz=500、1000step

最適化オプションの比較

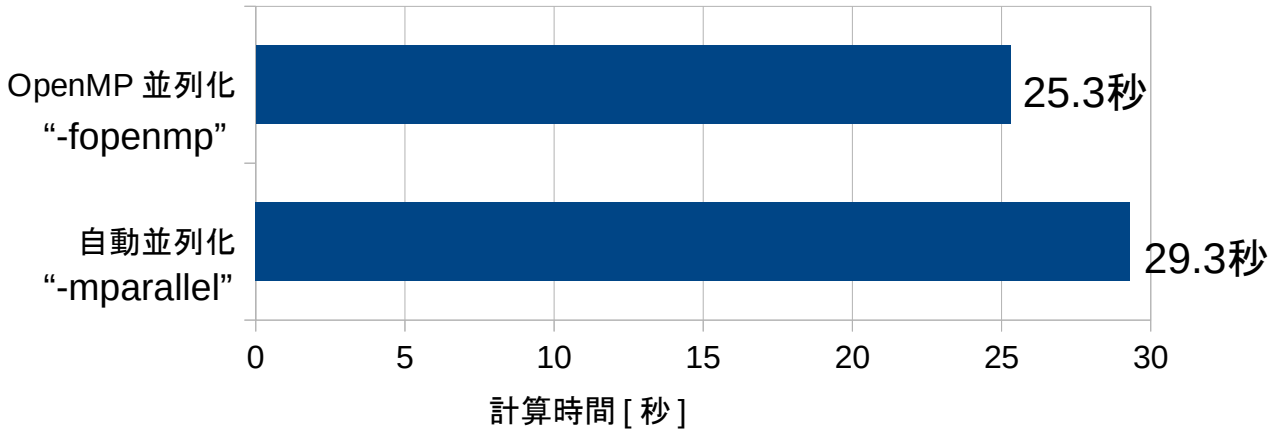


・計算時間は最適化オプションによらない→安全のために-O2とする

・計算結果は同じ

・計算条件：Nx=Ny=Nz=500、1000step、8スレッド、1ノード

自動並列化



・SXではコンパイル・リンクオプションに”-mparallel”をつけると自動並列化される

・OpenMPで陽に並列化することに比べて少し遅くなるがOpenMPプログラミングが不要になる

・計算条件：Nx=Ny=Nz=500、1000step、8スレッド、1ノード

ループ長とベクトル長の関係

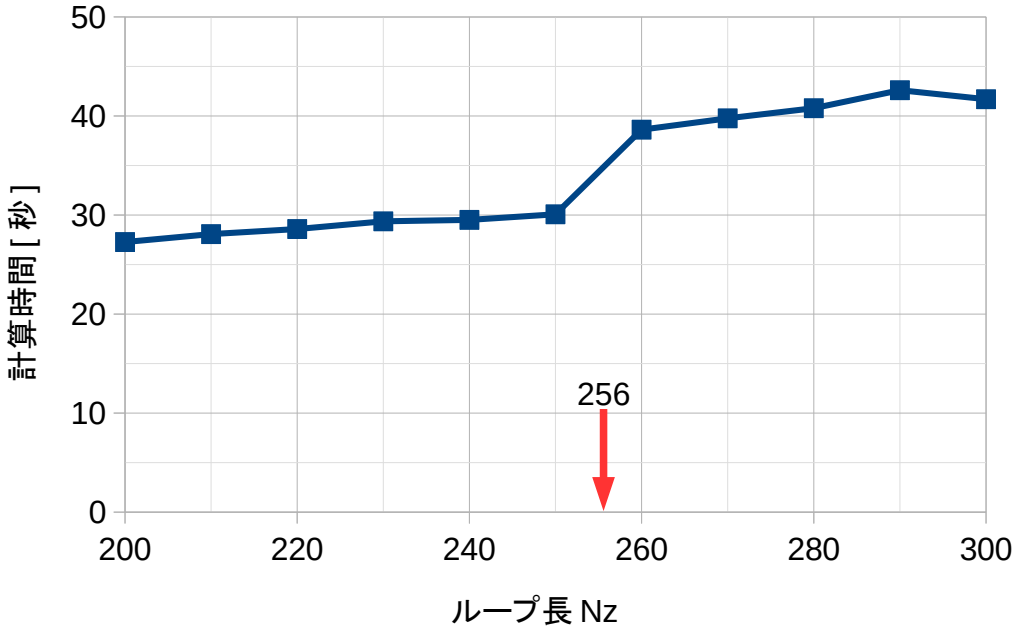
SXではベクトル長が256であり一番内側のループがベクトル化される。

OpenFDTDでは一番内側のループ長はNzまたはNz+1になる。

Nzを変えたときの計算時間は右の通り、256で不連続に増える。

ベクトル演算の回数 $\left[\frac{Nz+255}{256}\right]$ が256を境に1から2になるため。

繰り上げ商：$\left[\frac{n+d-1}{d}\right]$



・計算条件：Nx=Ny=500、Nz可変、1000step、8スレッド、1ノード

強制ベクトル化

Mur一次吸収境界条件は下記のようになるが、コンパイルすると下のように部分ベクトル化になる。アルゴリズム上、左辺のHXは右辺の別の場所に現れないことがわかっているので右のように強制ベクトル化してよい。

```
 1  #include "ofd.h"
 2
 3  void murHx()
 4  {
 5        int64_t n;
 6  #ifdef _OPENMP
 7  #pragma omp parallel for
 8  #endif
 9        for (n = 0; n < numMurHx; n++) {
10              int i  = fMurHx[n].i;
11              int j  = fMurHx[n].j;
12              int k  = fMurHx[n].k;
13              int i1 = fMurHx[n].i1;
14              int j1 = fMurHx[n].j1;
15              int k1 = fMurHx[n].k1;
16              HX(i, j, k) = fMurHx[n].f
17                 + fMurHx[n].g * (HX(i1, j1, k1) - HX(i, j, k));
18              fMurHx[n].f = HX(i1, j1, k1);
19        }
20  }
```

```
$ ncc -c -O2 -I../include -fopenmp murHx.c
ncc: par(1801): murHx.c, line 7: Parallel routine generated.: murHx$1
ncc: par(1803): murHx.c, line 9: Parallelized by "for".
ncc: vec( 102): murHx.c, line 9: Partially vectorized loop.
```

```
 1  #include "ofd.h"
 2
 3  void murHx()
 4  {
 5        int64_t n;
 6  #ifdef _OPENMP
 7  #pragma omp parallel for
 8  #endif
 9  #ifdef __NEC__
10  #pragma _NEC ivdep
11  #endif
12        for (n = 0; n < numMurHx; n++) {
13              int i  = fMurHx[n].i;
14              int j  = fMurHx[n].j;
15              int k  = fMurHx[n].k;
16              int i1 = fMurHx[n].i1;
17              int j1 = fMurHx[n].j1;
18              int k1 = fMurHx[n].k1;
19              HX(i, j, k) = fMurHx[n].f
20                 + fMurHx[n].g * (HX(i1, j1, k1) - HX(i, j, k));
21              fMurHx[n].f = HX(i1, j1, k1);
22        }
23  }
```

強制ベクトル化

```
$ ncc -c -O2 -I../include -fopenmp murHx.c
ncc: par(1801): murHx.c, line 7: Parallel routine generated.: murHx$1
ncc: par(1803): murHx.c, line 12: Parallelized by "for".
ncc: vec( 101): murHx.c, line 12: Vectorized loop.
```

還元演算(reduction)

OpenFDTDでは収束判定に全領域の平均電磁界が必要になる。このような演算の並列化を還元演算と呼ぶ。
右の関数をOpenMPと自動並列化でコンパイルすると以下のメッセージが出る。どちらも並列化されていることがわかる。

```
$ ncc -c -O2 -I../include -fopenmp -fdiag-vector=2 -fdiag-parallel=2 average.c
ncc: par(1801): average.c, line 9: Parallel routine generated.: average$1
ncc: par(1803): average.c, line 11: Parallelized by "for"
ncc: opt(1097): average.c, line 13: This statement prevents loop optimization.
ncc: vec( 101): average.c, line 13: Vectorized loop.
ncc: vec( 126): average.c, line 14: Idiom detected.: Sum
ncc: vec( 126): average.c, line 30: Idiom detected.: Sum
ncc: par(1809): average.c, line 42: Barrier synchronization.
$ ncc -c -O2 -I../include -mparallel -fdiag-vector=2 -fdiag-parallel=2 average.c
ncc: par(1801): average.c, line 11: Parallel routine generated.: average$1
ncc: par(1803): average.c, line 11: Parallelized by "for"
ncc: vec( 101): average.c, line 13: Vectorized loop.
ncc: vec( 126): average.c, line 14: Idiom detected.: Sum
ncc: vec( 126): average.c, line 30: Idiom detected.: Sum
```

```
 1  #include "ofd.h"
 2
 3  void average(double fsum[])
 4  {
 5          double se = 0;
 6          double sh = 0;
 7          int    i;
 8  #ifdef _OPENMP
 9  #pragma omp parallel for reduction(+ : se, sh)
10  #endif
11          for (    i = 0; i < Nx; i++) {
12          for (int j = 0; j < Ny; j++) {
13          for (int k = 0; k < Nz; k++) {
14                  se +=
15                          + fabs(
16                                  + EX(i    , j    , k    )
17                                  + EX(i    , j + 1, k    )
18                                  + EX(i    , j    , k + 1)
19                                  + EX(i    , j + 1, k + 1))
20                          + fabs(
21                                  + EY(i    , j    , k    )
22                                  + EY(i    , j    , k + 1)
23                                  + EY(i + 1, j    , k    )
24                                  + EY(i + 1, j    , k + 1))
25                          + fabs(
26                                  + EZ(i    , j    , k    )
27                                  + EZ(i + 1, j    , k    )
28                                  + EZ(i    , j + 1, k    )
29                                  + EZ(i + 1, j + 1, k    ));
30                  sh +=
31                          + fabs(
32                                  + HX(i    , j    , k    )
33                                  + HX(i + 1, j    , k    ))
34                          + fabs(
35                                  + HY(i    , j    , k    )
36                                  + HY(i    , j + 1, k    ))
37                          + fabs(
38                                  + HZ(i    , j    , k    )
39                                  + HZ(i    , j    , k + 1));
40          }
41          }
42          }
43
44          fsum[0] = se / (4.0 * Nx * Ny * Nz);
45          fsum[1] = sh / (2.0 * Nx * Ny * Nz);
46  }
```

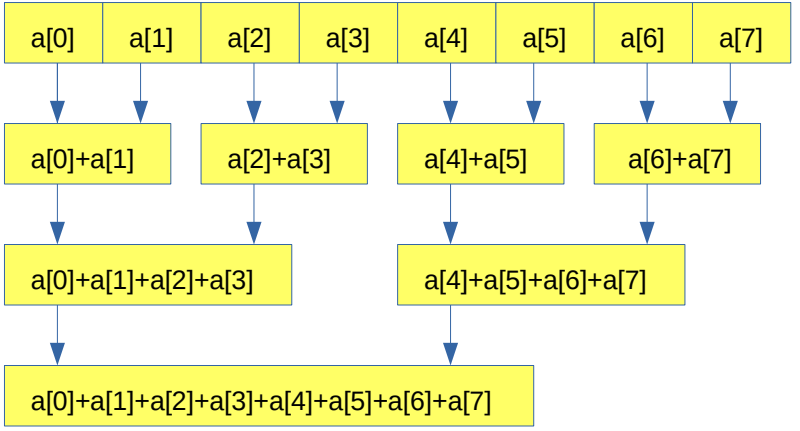
OpenMP

還元演算(reduction)のプログラミング

総和などを複数のスレッドで協力して計算することを還元演算と呼ぶ（SXの自動並列化ではマクロ演算と呼ぶ）。環境によって以下のプログラミングが必要になる。

- CPU : OpenMPのreduction指示文を使用する
- MPI : MPI_Reduce関数を使用する
- CUDA : 自作する（サンプルあり）
- SX : OpenMPまたは自動並列化を使用する

還元演算によってN個の演算が$\log_2 N$ステップで処理される（N個の演算器があるとき）。



8個の総和の還元演算（3ステップで終了する）

プロファイル(1) (ngprof)

ngprofの使い方

・コンパイルオプションとリンクオプションに”-p”をつけてビルドする

・通常と同じく実行する

（gmon.outができる）

$ ngprof ofd_ncc

(標準出力に出力される)

出力例(Nx=Ny=Nz=300, 3000step, 1スレッド)

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 15.27     28.18    28.18     3001     0.01     0.01  updateHy
 14.94     55.76    27.58     3001     0.01     0.01  updateEy
 14.32     82.19    26.43     3001     0.01     0.01  updateHx
 14.27    108.53    26.34     3001     0.01     0.01  updateEx
 14.03    134.43    25.90     3001     0.01     0.01  updateEz
 13.74    159.79    25.36     3001     0.01     0.01  updateHz
  6.09    171.04    11.25 26269230     0.00     0.00  in_geometry
  1.79    174.35     3.31     3001     0.00     0.00  murHz
  1.57    177.25     2.90     3001     0.00     0.00  murHy
  1.54    180.09     2.84     3001     0.00     0.00  murHx
  1.02    181.97     1.88        1     1.88    13.13  setupId
  0.29    182.51     0.54        2     0.27     0.35  setupMurHx
  0.27    183.01     0.50  1083600     0.00     0.00  factorMur
  0.27    183.50     0.49        2     0.25     0.33  setupMurHy
  0.26    183.98     0.48        2     0.24     0.32  setupMurHz
  0.21    184.37     0.39       16     0.02     0.02  average
  0.04    184.45     0.08        1     0.08   169.35  solve
  0.03    184.50     0.05        1     0.05     0.05  setup_material
  0.02    184.53     0.03     3001     0.00     0.00  efeed
（以下略）
```

コア部で87%

計算時間計測のためのオーバーヘッドが含まれていることに注意

プロファイル(2) (PROGINF)

PROGINFの使い方

・通常と同じ方法でビルドする

$ export VE_PROGINF=DETAIL

・通常と同じく実行する

(標準エラーに出力される)

※OpenMPのマルチスレッドを使用すると結果が正しくないことがあるので1スレッドで実行する(マニュアルより)

出力例(Nx=Ny=Nz=300, 3000step, 1スレッド)

```
          ********  Program  Information  ********
 Real Time (sec)                          :          119.565017
 User Time (sec)                          :          119.502737
 Vector Time (sec)                        :          116.728085
 Inst. Count                              :        199441222251
 V. Inst. Count                           :         48436361963
 V. Element Count                         :       7328628071908
 V. Load Element Count                    :       3790260328882
 FLOP Count                               :       3927320832766
 MOPS                                     :         70804.522083
 MOPS (Real)                              :         70758.293450
 MFLOPS                                   :         32868.743335
 MFLOPS (Real)                            :         32847.283165
 A. V. Length                             :           151.304263
 V. Op. Ratio (%)                         :            98.215089
 L1 Cache Miss (sec)                      :             0.266222
 CPU Port Conf. (sec)                     :             0.186285
 V. Arith. Exec. (sec)                    :            50.999870
 V. Load Exec. (sec)                      :            65.579170
 VLD LLC Hit Element Ratio (%)            :            17.904995
 Power Throttling (sec)                   :             0.000000
 Thermal Throttling (sec)                 :             0.000000
 Memory Size Used (MB)                    :          2856.000000

 Start Time (date)         :        Sat Sep 28 17:43:29 2019 JST
 End   Time (date)         :        Sat Sep 28 17:45:29 2019 JST
```

32GFLOPS

ベクトル化率98.2%

プロファイル(3) (FTRACE)

FTRACEの使い方

・コンパイルオプションとリンクオプションに”-ftrace”をつけてビルドする

・通常と同じく実行する

（ftrace.outができる）

$ ftrace

(標準出力に出力される)

出力例(Nx=Ny=Nz=300, 3000step, 1スレッド)

```
*----------------------*
  FTRACE ANALYSIS LIST
*----------------------*

Execution Date : Sat Sep 28 17:33:07 2019 JST
Total CPU Time : 0:02'12"163 (132.163 sec.)
```

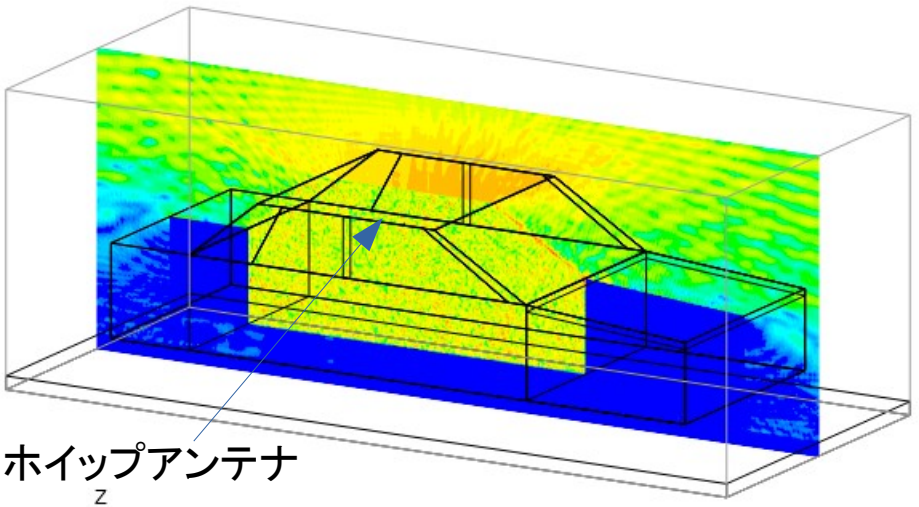
FREQUENCY	EXCLUSIVE TIME[sec](%)	AVER.TIME [msec]	MOPS	MFLOPS	V.OP RATIO	AVER. V.LEN	VECTOR TIME	L1CACHE MISS	CPU PORT CONF	VLD LLC HIT E.%	PROC.NAME
3001	19.596(14.8)	6.530	72007.8	33189.0	98.24	150.0	19.555	0.038	0.000	24.94	updateHy
3001	19.232(14.6)	6.408	73626.1	33930.6	98.23	150.5	19.174	0.050	0.000	24.97	updateEy
3001	18.234(13.8)	6.076	77149.9	35669.0	98.25	150.0	18.232	0.002	0.000	12.68	updateHx
3001	18.198(13.8)	6.064	77568.9	35857.9	98.23	150.5	18.196	0.002	0.000	12.81	updateEx
3001	17.828(13.5)	5.941	74742.8	36602.3	98.28	150.1	17.780	0.046	0.000	15.04	updateEz
3001	17.408(13.2)	5.801	76302.4	37361.0	98.26	150.5	17.352	0.053	0.000	15.85	updateHz
1	7.557(5.7)	7557.004	1010.4	0.0	14.95	150.3	0.017	4.166	0.000	16.70	setupId
26269230	5.688(4.3)	0.000	835.9	46.2	0.00	0.0	0.000	3.230	0.000	0.00	in_geometry
3001	2.228(1.7)	0.742	18533.3	1459.7	99.76	256.0	2.226	0.002	0.000	35.58	murHz
3001	2.035(1.5)	0.678	20289.1	1598.0	99.76	256.0	2.034	0.001	0.092	19.18	murHx
3001	2.006(1.5)	0.668	20586.0	1621.4	99.76	256.0	2.005	0.001	0.094	23.43	murHy
2	0.493(0.4)	246.615	1523.9	0.7	0.00	0.0	0.000	0.119	0.000	0.00	setupMurHz
2	0.439(0.3)	219.644	1457.4	0.8	0.00	0.0	0.000	0.074	0.000	0.00	setupMurHx
1083600	0.413(0.3)	0.000	614.5	47.3	0.00	0.0	0.000	0.259	0.000	0.00	factorMur
2	0.405(0.3)	202.410	1852.5	0.9	0.00	0.0	0.000	0.045	0.000	0.00	setupMurHy
16	0.242(0.2)	15.100	92829.2	35248.0	99.04	151.2	0.242	0.000	0.000	42.94	average
（中略）											
27399880	132.163(100.0)	0.005	64078.5	29715.8	98.11	151.3	116.852	8.164	0.186	17.95	total

ループ長=300なので平均=300/2=150

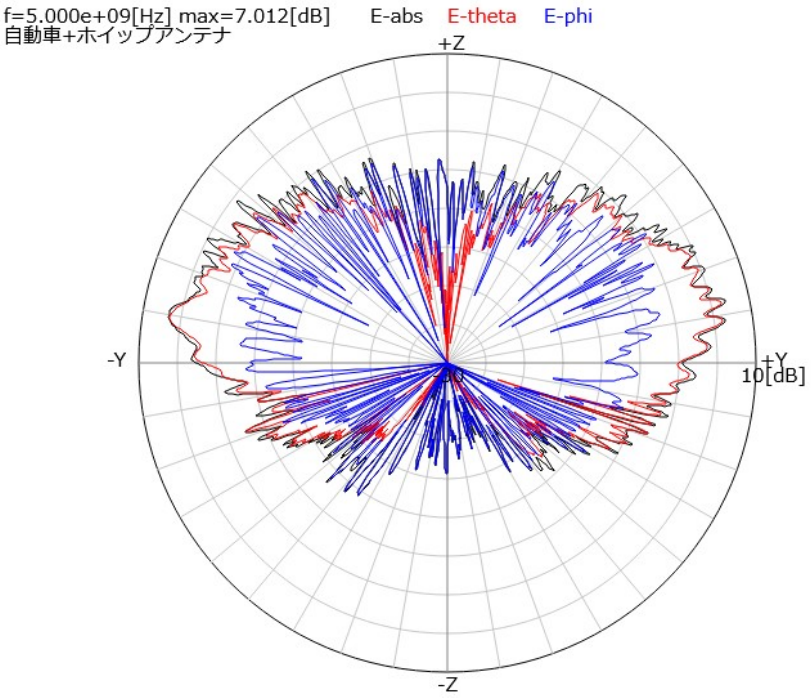
コア部で84%

ベクトル化されない関数が10%消費している

計算例(1)



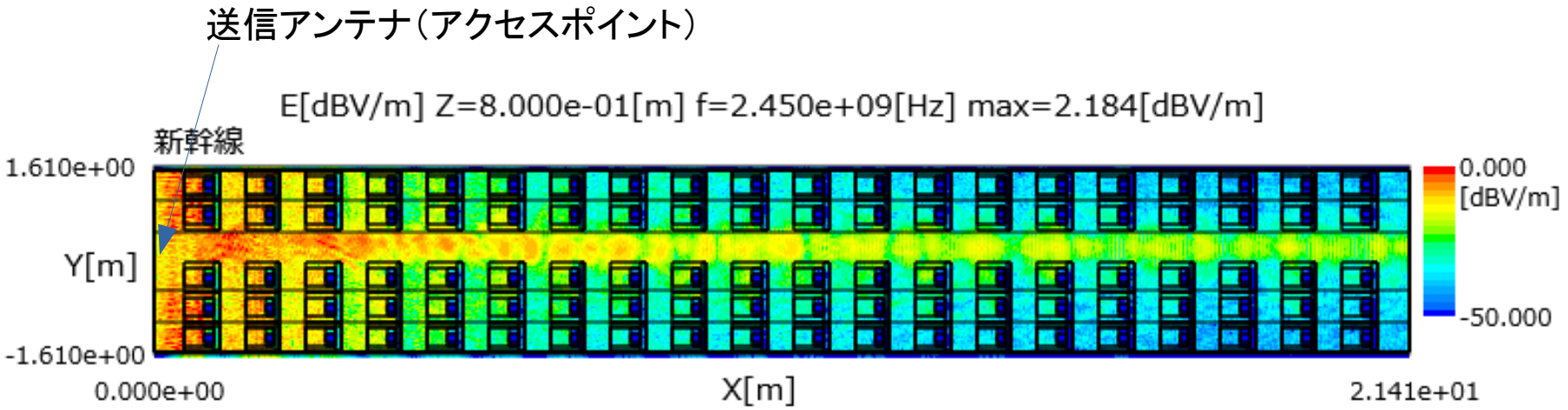
(a)電界分布(dB表示)



(b)遠方界パターン(YZ面）

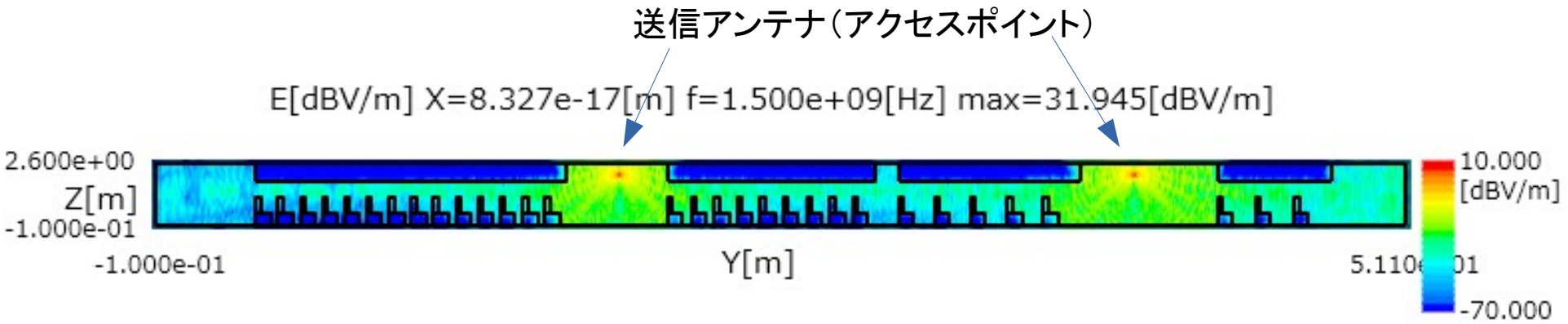
車周囲の電界分布(5GHz)
セル数=520×1,180×440=269,984,000
タイムステップ数=10,000
総使用メモリー=25.7GB
計算時間=8分(2ノード)

計算例(2)



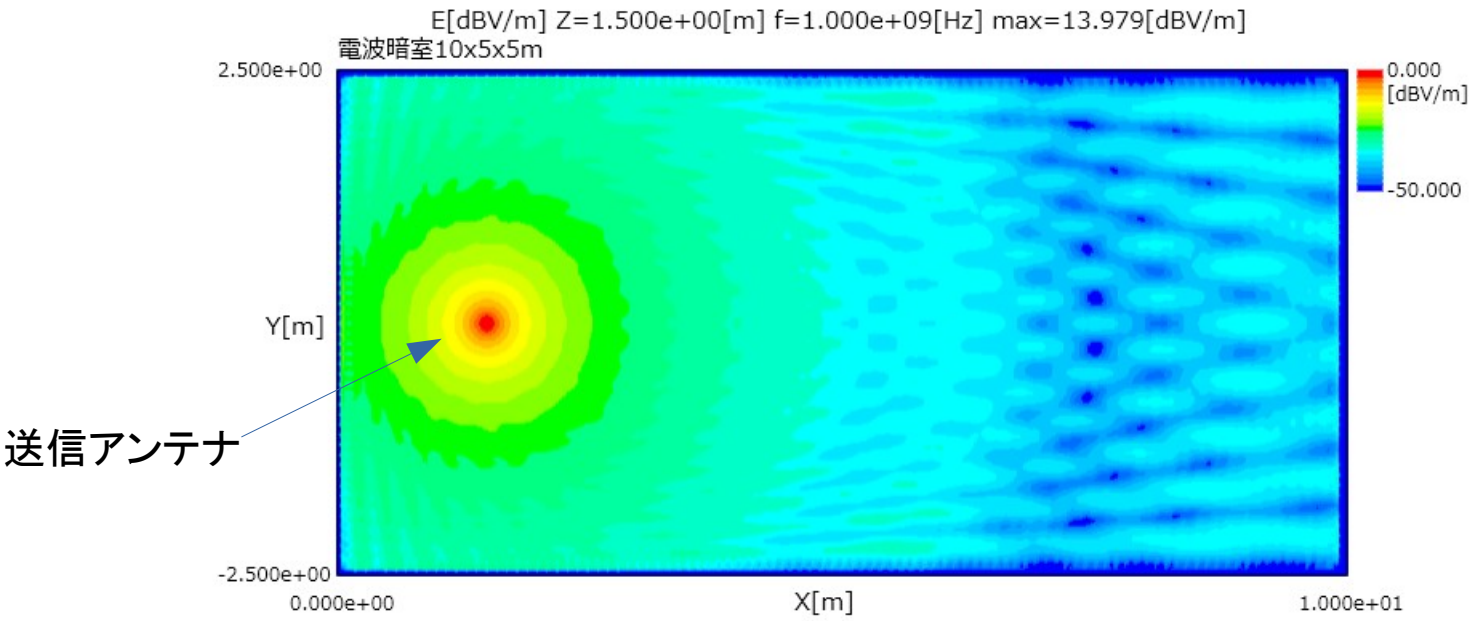
新幹線車内の電界分布(2.45GHz，床上0.8m面)
セル数=2,141×322×217=149,600,234
タイムステップ数=10,000
総使用メモリー=14.2GB
計算時間=6分(2ノード)

計算例(3)



飛行機内の電界分布(1.5GHz)
セル数=400×2,560×135=138,240,000
タイムステップ数=20,000
総使用メモリー=13.4GB
計算時間=9分(2ノード)

計算例(4)



電波暗室内の電界分布(1GHz)
セル数=1,000×500×500=250,000,000
タイムステップ数=5,000
総使用メモリー=23.4GB
計算時間=3分(2ノード)

まとめ

・電磁界シミュレーターOpenFDTDをSX-Aurora TSUBASAに移植しその性能を評価した。
・OpenFDTDはベクトル化とOpenMP並列化に適したアルゴリズムなのでSXで高い性能が得られる。
・OpenFDTDはMPI並列化されており、そのままSXの複数ノードに対応できる。

・ベクトル化と並列化に適したアルゴリズムであればSXに容易に移植することができる。
（これは常にコードの大幅な変更が必要になるCUDAに比べると優れている）
・MPIで並列化されたプログラムであればノード数を増やせばさらに速くなる。
（多ノードをあまり想定していないGPUに比べるとSXはスケーラビリティが高い）

文献

[1] OpenFDTD, http://www.e-em.co.jp/OpenFDTD/

[2] 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

[3] 大賀明夫, "OpenFDTDとOpenMOMで始める はじめてのアンテナ・シミュレーション," RFワールド, no.39, pp.7-82, CQ出版社, 2017

[4] 大賀明夫, "FDTD法の並列化技術とオープンソース化," 電子情報通信学会技術研究報告, AP2014-41, pp.7-12, 2014年6月

[5] 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 第2版,” カットシステム, 2012

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

[7] John Cheng, Max Grossman, Ty McKercher (株式会社クイープ訳), "CUDA C プロフェッショナル プログラミング," インプレス, 2015

[8] NEC SDK, https://www.hpc.nec/documents/sdk/

[9] 公益財団法人計算科学振興財団, FOCUSスパコン, https://www.j-focus.or.jp/