各種計算ハードウェアの活用（D5）
～CPU の性能を引き出す手法～

担当者　長谷 芳樹（神戸市立工業高等専門学校 電子工学科）

1　はじめに

従来より，複数のコンピュータをネットワークで接続して演算速度やメインメモリ（主記憶装置）容量を向上させる並列化技術が利用されてきた。また，近年のコンピュータやスマートホンには複数の CPU あるいはマルチコア CPU を搭載していることがほとんどであり，演算速度向上に大きく寄与している。メモリそのものについても，比較的安価なコンピュータでもデュアルチャンネルあるいはクアッドチャンネルで接続されるようになってきており，ノイマン型コンピュータにおけるボトルネックになりがちなメモリバス幅も日進月歩で改善している。さらに，GPU（GPGPU）を用いた並列計算も一般的になっている。GPU を用いると，対象とするモデルサイズとアルゴリズムによっては CPU による演算と比して数倍～数 100 倍，あるいはそれ以上の演算速度を得られる場合もあり，日常的に利用されるようになっている。

これらの技術は，本実験の前半で扱った FPGA と共に，同一コストでの，あるいは同一の消費電力あたりの演算速度向上に寄与している。本実験ではこれらの技術のうち，共有メモリ型のマルチコア CPU の性能を引き出す基礎的な手法について実習をおこなう。

2　スケジュール

- 第 1 週　5 のおわりまでと 6.1～6.5
- 第 2 週　6.6～6.8，自由課題着手
- 第 3 週　FPGA 自由課題の制作，プレゼンテーション準備
- 第 4 週　FPGA 自由課題のプレゼンテーションとディスカッション
- **第 5 週　CPU の性能を引き出す手法（この資料）**
- 第 6 週　GPU（GPGPU）で並列演算をおこなう手法（資料は別途配布）

3　使用するハードウェアとソフトウェア

本実験では下記の環境を利用する。

- PC： 4 コアの CPU （Intel Core i5-4460 プロセッサ：キャッシュ 6 MB, 動作クロック 3.2 GHz） およびデュアルチャンネルメモリ（DDR3 1600 MHz, 4 GB × 2）を搭載したもの
- コンパイラ： gcc (Cygwin64 [1] 上で利用)
- 可視化ソフト： Scilab [2]

4　ベースとなるプログラム（FDTD 法）のダウンロードと実行

ベースとなるサンプルプログラムをダウンロードし，実行してみる。

- ファイルを https://github.com/nagataniyoshiki/kcct_d/ [3] からダウンロードし，c:￥cygwin64￥home￥jikken￥ などのフォルダに保存する（ファイル名とパス名は任意だが，半角英数字のみ）。
- Cygwin64 を起動する。（cygwin の使い方は末尾の付録等を参照）
- ファイルを保存したフォルダに移動する（cd /cygdrive/c/保存したフォルダ/ など）。
- gcc acoustic_2d_fdtd.c -lm と打ち込む（コンパイル：“a.exe” が生成される）。
- ./a.exe と打ち込む（実行）。

◆課題：各時刻の音場（音圧分布）を記録した “field*.txt” ファイルが生成されるので，可視化スクリプト field_1.sci あるいは field_2.sci を Scilab [2] で実行して音波が伝搬する様子を確認する。

5　FDTD 法の概要（参考）

このプログラムは 2 次元の音響 FDTD 法（finite-difference time-domain method）[4, 6] のサンプルプログラムである。2 次元空間を正方形のグリッドに区切り，図 1 に示すように格子点に音圧値（p: pressure），そこから 1/2 ずれた位置に粒子速度値（v: particle velocity）を定義し，音波の伝搬を計算する手法である。

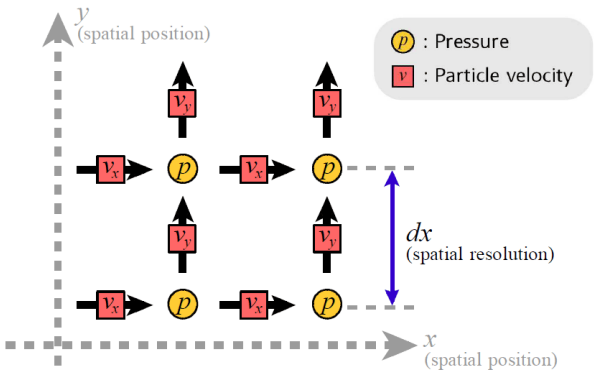

図 1: 2 次元音響 FDTD 法の空間グリッド配置

音響 FDTD 法（流体中の音波伝搬解析手法）の支配方程式は，電磁波におけるマクスウェルの波動方程式と同等であり，電界と磁界を音圧と粒子速度に置き換えればよい：

$$\frac{\partial p}{\partial t} = -\kappa \left(\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} \right), \tag{1}$$

$$\frac{\partial v_x}{\partial t} = -\frac{1}{\rho}\frac{\partial p}{\partial x}, \quad \frac{\partial v_y}{\partial t} = -\frac{1}{\rho}\frac{\partial p}{\partial y}. \tag{2}$$

ここで，p は音圧，v は粒子速度である。また，x, y は空間座標，t は時刻，κ は体積弾性率，ρ は密度である。式 (1) で音圧を，式 (2) で粒子速度をそれぞれ計算する。次に，差分化の例を示す（中心差分）：

音圧：　`P[i][j] += -(kappa*dt/dx) * ( (Vx[i+1][j]-Vx[i][j]) + (Vy[i][j+1]-Vy[i][j]) );`
粒子速度（x 方向）：　`Vx[i][j] += -dt/(rho*dx) * (P[i][j]-P[i-1][j]);`
粒子速度（y 方向）：　`Vy[i][j] += -dt/(rho*dx) * (P[i][j]-P[i][j-1]);`

音場の更新に際しては，この 3 つの差分式を用いて，図 2 に示すように音圧値→粒子速度値→音圧値→粒子速度値 … のように交互に計算する。この手法は「蛙跳び（leap-frog）差分」などと呼ばれることもある。

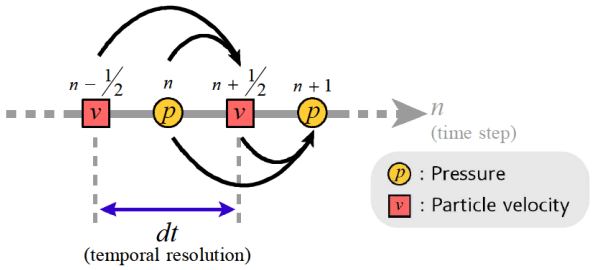

図 2: 2 次元音響 FDTD 法の時間ステップ

（音響 FDTD 法についての詳細は [5, 6, 7, 8] 等を参照のこと。）

本実験のサンプルプログラムでは，空間全体を空気（音速 331 m/s，密度 1.3 kg/m^3）で満たしている。空間サイズ NX × NY は 300 × 400 点，空間解像度 dx は 10 cm，計算の時間刻み dt は 20 マイクロ秒としている。空間上のある点から freq = 1 kHz の正弦波（二乗余弦関数をかけたもの）を 1 波長だけ音圧波形として送波している。端面は全反射となっている。
◆課題： 周波数（freq）や計算ステップ数（Nstep）を変えて実行してみる。

6 実行時間の測定

プログラムの実行時間はストップウォッチ等で測定してもよいが，ここでは time.h を利用して自動的に出力するようにプログラムを変更する。（必要に応じてファイル出力の有無や計算ステップ数を調整する。）

●ヘッダファイルを include する

```
01: #include<time.h>
```

●プログラム開始時と終了時の時刻を記録して，差を出力する

```
変数定義: time_t t_start,t_end;
開始時刻: t_start = time(NULL);
終了時刻: t_end = time(NULL);
実行時間: fprintf(stderr,"Computation time: %d s. \n", (int)(t_end-t_start));
```

これでプログラムの実行時間が表示されるので，再度コンパイルして実行してみる。
◆課題：演算時間を記録する

7 最適化コンパイル

最も容易に利用できる高速化手法は，コンパイラに用意されている最適化オプションを用いることである。コンパイル時に最適化オプションを指定すると自動的にコードが解析され，ループ等が最適化されたバイナリファイルが生成される。本実験で使用する gcc にも何段階ものオプションがあるが，ここでは最も一般的な -O から -O3 までを試してみる（“ゼロ” ではなく “オー”）。

●最適化コンパイル（デフォルトでは “a.exe” が生成される。出力ファイル名は -o オプションで変更可。）

```
gcc ファイル名.c -lm
gcc ファイル名.c -lm -O
gcc ファイル名.c -lm -O2
gcc ファイル名.c -lm -O3
```

◆課題：各オプション毎に演算時間を記録する。

8 配列確保時の留意点

一般に，メモリ– CPU 間のデータ転送のレイテンシー（待ち時間）は CPU のクロックに比べて極めて遅いため，そのままでは CPU 側で長い待ち時間が発生して演算速度が上がらないという問題が生ずる。この問題を軽減するため，CPU 側により高速な（ただし小容量の）“ キャッシュ ” が用意されていることが多い。例えば，ある番地（アドレス）に格納されているデータが読み込まれると，連続した番地に格納されているデータも自動的にキャッシュに転送される。この工夫により，次に近くのアドレスのデータにアクセスする場合（すなわち，キャッシュに “ ヒット ” した場合），高速に読み込むことができ，結果として CPU の待ち時間を減らすことが出来る。

従って，効率的なプログラミングをおこなう際には，できる限りメモリ上に連続した番地に配置したデータに連続してアクセスするように注意する必要がある。

たとえば以下のようにすると，各変数のアドレスが確認できる。

●番地（アドレス）の表示

```
01: for (i = 0; i < NX; i++) {
02:     for (j = 0; j < NY; j++) {
03:         printf(" P[%d][%d] : %p\n", i, j, &P[i][j]);
04:     }
05: }
06: printf("\n");
```

（※このとき，`#define` で指定している `NX` と `NY` は一時的に小さくしておくと見やすい）

◆課題：アドレスが連続していることを確認する。（float でも試してみる）

次に，アドレスの連続しない順にデータにアクセスし，演算速度がどのように変化するかを確かめる。

●ループ順序の変更（ **j → i** の順にループを回す）

```
01: for(j=0;j<NY;j++){
02:     for(i=0;i<NX+1;i++){
03:         Vx[i][j] += - dt / (rho * dx) * ( P[i][j] - P[i-1][j] );
04:     }
05: }
06:
07: for(j=0;j<NY+1;j++){
08:     for(i=0;i<NX;i++){
09:         Vy[i][j] += - dt / (rho * dx) * ( P[i][j] - P[i][j-1] );
```

```
10:     }
11: }
12:
13: for(j=0;j<NY;j++){
14:     for(i=0;i<NX;i++){
15:         P[i][j] += - ( kappa * dt / dx )
16:                       * ( ( Vx[i+1][j] - Vx[i][j] ) + ( Vy[i][j+1] - Vy[i][j] ) );
17:     }
18: }
```

◆課題：演算時間を記録する（コンパイル時の最適化オプションに注意）

なお，本サンプルプログラムでは静的に配列確保をおこなっているが，実際には malloc 関数で動的に確保することが一般的である。余裕があれば試してみて，アドレスがどのように割り当てられるかを確認してみる。

9 MPIによる並列化（参考）

並列化には，大きく分けて,「分散メモリ型」と「共有メモリ型」とがある。複数の計算機をネットワークで接続して計算する手法（特に安価な PC を用いた場合は“PC クラスター”などと呼ぶ場合もある）は前者の「メモリ分散型」である。これに対し，1 台の計算機に複数の CPU を搭載したものやマルチコアの CPU を用いる場合は，後者の「メモリ共有型」である。

いずれの場合でも，メッセージ通信の API 仕様である MPI（Message Passing Interface）が利用できる [9]。MPI はプロセッサ間でメッセージ（転送の指示や同期の指定など）をやりとりしながら並列処理をおこなう方式であり，移植性も非常に高い。実際，小さな研究室の PC クラスタから，京コンピュータのような大規模な計算機まで，広く MPI が採用されている。

MPI の実装としては，MPICH [10] や Open MPI [11] など，いくつかのものが存在する。実装により，BSD，Linux，OSX，Windows，Cygwin などで動作するものが提供されている。

● MPI プログラムの最低限の枠組み（参考）

```
01: #include "mpi.h" /* ヘッダファイルの読み込み */
02: int main(int argc, char **argv)
03: {
04:     int numprocs, myid;
05:
06:     MPI_Init(&argc, &argv); /* MPI を利用するための準備（初期化）*/
07:     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
08:     /* コミュニケータ内のプロセス数を取得（numprocs にプロセス数が入る）。*/
09:     MPI_Comm_rank(MPI_COMM_WORLD, &myid);
10:     /* コミュニケータ内の各プロセスが自分の rank を取得（myid に自分の rank 番号が入る）。*/
11:
12:     /* 並列処理をここに記述する*/
13:
14:     MPI_Finalize(); /* MPI の終了処理 */
15:     return 0;
16: }
```

10　OpenMP による並列化

共有メモリ型の場合でも，MPI を用いてコーディングすると自由度が高く，性能を追求することが可能である。また，コードの変更なしに大規模な並列計算機（京コンピュータなど）でそのまま実行することが出来るため，汎用性が高い。反面，コミュニケータ（並列計算を一緒におこなうノードの集まり）を適切に管理したりメッセージ通信の順序を考慮する必要があるなど，簡単とはいえ，ある程度の手間がかかる。また，MPI 用に構築したコードはそのままでは MPI に対応していない計算機用にはコンパイルすることが出来ないという制約もある。

この制約を持たない別の手法として，共有メモリ型の計算機に限定して，極めて容易にコーディングできるように，さらに，非対応環境でもそのままコンパイルできるように考案された OpenMP という並列化手法がある。これは，ソースコード内にコンパイラのための“ヒント（ディレクティブ）”を追記することにより簡易的に並列化を実現するものであり，コードそのものは一切変更することなく，コンパイル時のオプションを変更するだけで並列化の有無を選択することができるという特徴がある。本実験では，OpenMP の最も基本的な手法のみを紹介する。

OpenMP の基本的な利用は極めて簡単であり，並列化したいループ文（for 文や while 文）の直前に，コンパイラのためのヒント情報をコメントとして追記するだけである。その際に，並列化した後の各スレッド間で相互に参照しない変数（すなわち，各スレッドに並列化しても差し支えないプライベートな変数）のリストを private() で与える。

● OpenMP のためのヘッダファイルを include する

```
01: #include<omp.h>
```

● Vx, Vy, P の更新ループを OpenMP で並列化する

```
00:     #pragma omp parallel for private(i,j)
01:     for(i=1;i<NX;i++){
02:         for(j=0;j<NY;j++){
03:             Vx[i][j] += - dt / (rho * dx) * ( P[i][j] - P[i-1][j] );
04:         }
05:     }
06:
07:     #pragma omp parallel for private(i,j)
08:     for(i=0;i<NX;i++){
09:         for(j=1;j<NY;j++){
10:             Vy[i][j] += - dt / (rho * dx) * ( P[i][j] - P[i][j-1] );
11:         }
12:     }
13:
14:     #pragma omp parallel for private(i,j)
15:     for(i=0;i<NX;i++){
16:         for(j=0;j<NY;j++){
17:             P[i][j] += - ( kappa * dt / dx )
18:                         * ( ( Vx[i+1][j] - Vx[i][j] ) + ( Vy[i][j+1] - Vy[i][j] ) );
19:         }
20:     }
```

このように，各 for ループの前に #pragma の行を追加するだけで並列化できる。この例では i と j は各スレッド間で共有される必要がない（独立である）ため， private(i,j) と指定している。逆に，Vx, Vy,

P は他のスレッドから更新されることがないために共有することが出来るので，並列化が有効に機能するわけである。

●並列化コンパイル

並列化なし： `gcc ファイル名.c -lm -O3`
並列化あり： `gcc ファイル名.c -lm -O3 -fopenmp`

◆課題：並列化の有無によって実行結果に差異が無いことを確認する。
◆課題：各オプション毎に演算時間を記録する。また，並列化有効時に複数のコアが使用されていることを Windows のタスクマネージャーで確認する。
◆課題：4 コア使用時に性能は何倍になったか？ 4 倍にならなかったとすればそれは何故か？

11 簡易レポート

第 5 週の課題はレポートは不要である。この替わりに，実験終了時に簡易レポートを提出する（A4 用紙 1 枚程度）。PC 等で作成した場合はファイルを nagatani 宛に送信しても良い。

- 各条件で実行したプログラムの実行時間（あるいは速度の向上率）を列挙する。
- 簡単なコメントを記す（簡易考察）。
- その他，アイデアや感想等があれば記す。

12 発展課題

ここまでで紹介した手法は並列化手法のごく一部である。各自，興味があれば深めて欲しい。

- さらに効率的なコードの追求（メモリ配置の工夫，計算量の低減，アムダールの法則 etc.）
- MPI を用いた並列化
- GPGPU を用いた並列化
- FPGA を用いた並列化
- 他のコンパイラや他の OS，あるいは他の言語との比較
- 可視化手法の検討

13 注意事項

- ソフトウェア工学実験室の PC は随時ファイル削除等をおこなうので，作成・更新したプロジェクトやスライドはこまめに各自の USB メモリ等に保存すること。
- ソフトウェア工学実験室の PC，プリンタ，ネットワーク機器などはていねいに扱うこと（**これは担当者からの強いお願いです！**）。

14　付録：cygwin のコマンド

cygwin [1] は UNIX ライクな環境を Windows 上に再現したものであり，コマンド等も UNIX 系 OS と似通っている。ここでは最低限だけ紹介しておく。

- man コマンド名 ： マニュアルの参照
- dir または ls または ls -l または ls -la ： ファイルリストの表示
- cd ディレクトリ名 ： ディレクトリの移動
- pwd ： カレントディレクトリ（現在地）の表示
- cat ファイル名 ： ファイル内容の表示
- rm ファイル名 ： ファイルの削除（確認メッセージは表示されない）
- mkdir ： ディレクトリの作成
- rmdir ： ディレクトリの削除 （※ rm -R でディレクトリの内容も問答無用で全削除 ）
- gcc 入力ファイル名.c -lm -O3 -fopenmp -o 出力ファイル名.exe ： コンパイル（一例）
- ./ファイル名 ： カレントディレクトリにある “ファイル名” を実行（※ “./” を付けない場合は path が検索される）

参考文献

[1] “Cygwin,” https://www.cygwin.com/ .

[2] “Scilab,” https://www.scilab.org/ .

[3] “Documents for the Experiments in the Department of Electronics, Kobe City College of Technology,” https://github.com/nagataniyoshiki/kcct_d/ .

[4] K.S. Yee: “Numerical solution of initial boundary value problems involing Maxwell’s equations in isotoropic media,” IEEE Transactions on Antenna and Propagation, AP-14(3) (1966) pp.302-307.

[5] 豊田政弘編, “FDTD 法で視る音の世界 （音響サイエンスシリーズ 14）,” コロナ社 (東京, 2015).

[6] 佐藤雅弘, “FDTD 法による弾性振動・波動の解析入門,” 森北出版 (東京, 2003).

[7] 日本建築学会編, “はじめての音響数値シミュレーション プログラミングガイド,” コロナ社 (東京, 2012).

[8] “FDTD Simulation Movie & Demo,” https://ultrasonics.jp/nagatani/fdtd/ .

[9] “Message Passing Interface Forum,” https://www.mpi-forum.org/ .

[10] “MPICH — High-Performance Portable MPI,” https://www.mpich.org/ .

[11] “Open MPI: Open Source High Performance Computing,” https://www.open-mpi.org/ .

(URL は 2019 年 5 月 22 日閲覧)
(rev. 201905b)