J. Plasma Fusion Res. Vol.92, No.6 (2016) 482-487

講座

今日からはじめるメニーコアアクセラレータ

Let's Start a Numerical Simulation Today Using the Many-Core Accelerator

1．メニーコアアクセラレータを研究室で使ってみよう

1. Let's Use Many-Core Accerelator in Your Laboratory

八柳祐一[1)]，田邊恵李[1)]，堀越将司[2)]
YATSUYANAGI Yuichi[1)], TANABE Eri[1)] and HORIKOSHI Masashi[2)]
[1)]静岡大学教育学部，[2)]インテル株式会社ソフトウェア＆ソリューション統括部

（原稿受付：2016年3月18日）

大学の研究室レベルで保有可能でありながら，マッシブな（大規模シミュレーションが可能な）計算環境を提供してくれるIntel® Xeon Phi™ コプロセッサ（アクセラレータ）の紹介を行う．アクセラレータはGPU（Graphics Processing Unit）と同様に，PCIeバスに挿す形で提供される並列計算機である．本章では，アクセラレータの特徴，及び導入の仕方を解説するとともに，非中性プラズマシミュレーションを例として取り上げ，計算可能なシミュレーション規模を視覚的に示す．

Keywords:
parallel computing, OpenMP, parallelize, vectorize, N-body simulation

1.1 初めに

1990年代のPCと言えば，CPUはi386（25 MHz）やi486DX2（66 MHz），Pentium（166 MHz），PentiumII（300 MHz），メインメモリは16 MB～256 MBが当たり前であり[1,2]，1000粒子程度のちょっとしたN体シミュレーションを行おうとすると，計算機センターに設置してあるスーパコンピュータを使わざるを得ず，長いジョブ待ち行列にうんざりしたものである．

その後，大学の1研究室レベルで保有可能なデスクトップスーパコンピュータの時代が，1990年代後半～2000年代に到来する．重力やクーロン力といった相互作用力の計算のみに特化した専用計算機（GRAPE/MDGRAPEシリーズ）により，10^4～10^5粒子のダイレクト・シミュレーションが現実的な時間で可能になった[3,4]．さらにその後，2010年代にかけてGPUが一般的になり，PC 1台分の大きさの計算機で1 TFLOPS*1の計算速度を実現するのも夢ではなくなった．

しかし，これらの専用計算機やGPUでは，それに特化したプログラミング言語（又は，API［Application Programming Interface］）を用いてコードを作成する必要がある．すなわち，インテルやAMD製のCPUを搭載するPCで動く並列コードは，そのままの形では，GPUなどで走らせることはできない*2．

このように，ハードウェア環境ごとにプログラミングやチューニングを行わなくてはならないのは，研究者にとって本質的ではない．そこで，当研究室では，PC用の並列コードを再コンパイルのみで超並列実行できる計算機，Intel® Xeon Phi™ コプロセッサ（以後，“アクセラレータ”と呼ぶ）に注目した．

本章では，アクセラレータの特徴，および使い方について簡単に紹介するとともに，非中性プラズマでのdiocotron不安定性を例にとり，どの程度の規模のシミュレーションが可能なのか解説をする．具体的な速度比較は次章で扱う（**図1**）．

1.2 アクセラレータの特徴，使い方

アクセラレータは，PCIeバスに挿すボードとして提供され，サイズは一般的なGPUと同じである．64ビット命令が

＊1　浮動小数点数の乗算などを1秒間に1 T=10^{12}回計算する能力．
＊2　近年，GPUとPC上でシームレスに動くようなAPI環境も整備されつつあるが，性能を引き出すのが困難なため，あまり普及していない．

Faculty of Education, Shizuoka University, Shizuoka City, SHIZUOKA 422-8529, Japan
corresponding author's e-mail: yatsuyanagi.yuichi@shizuoka.ac.jp

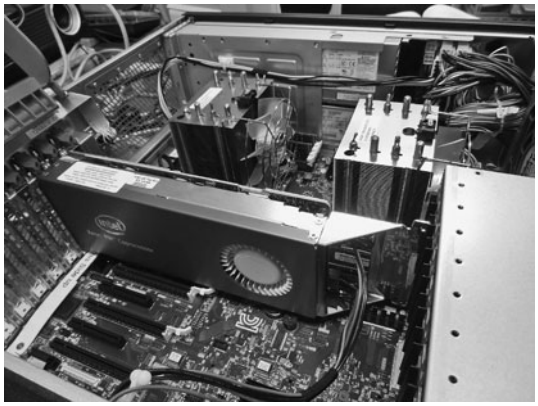

図1　PC に設置されたアクセラレータの様子.

実行可能なPentiumベースのCPUコアが61個，コアあたり4スレッド実行可能で，合計244並列の計算が可能である．GPU などの並列度を見慣れると，「244」並列というのは少ない印象を受けるが，GPU のストリーミングプロセッサと異なり，アクセラレータの CPU コアは，PC に搭載されるCPUコアと同様の能力を有する高機能なものである．各コアでは512ビット長のベクトル演算が可能で，倍精度浮動小数点数であれば 8 要素の同時計算が可能である．また，理論上は 1 コアあたり 2 スレッド以上の並列度でピーク性能を達成することが可能である．

GPU とアクセラレータの最も大きな違いは，アクセラレータ内で独立した OS が稼働していることだろう．すなわち，ホスト PC からは，LAN で接続された計算サーバのように見える．一般的に GPU では，初期化処理などをホストCPUで計算し，相互作用計算などの重たい計算部分のみを GPU にまかせる「オフロード」方式を採用する．このため，メモリバスに比べて低速な PCIe バス越しに大量の配列データをやり取りせざるを得ず，いかにこの通信を減らすかがGPU上のプログラミングでの腕の見せ所となる．一方，アクセラレータでは，重たい計算部分のみを「外注」するオフロード方式に加えて，アクセラレータに ssh などでリモートログインし，アクセラレータ用にコンパイルされたプログラムをシェルから起動する，といった使い方ができる．すなわち，イメージとしては，計算サーバに ssh でログインし，そこでプログラムをコンパイルして走らせるイメージに近い．この場合，計算対象となるデータが PCIe バス上を行き来するオーバヘッドがなくなるため，実行速度面での恩恵が大きい．

アクセラレータを使い始めるための手順を簡単に示すと，次のような流れになる．なお，当研究室で使用しているソフトウェア等の名称，およびバージョンは，以下のとおりである．

OS：CentOS 6.6
コンパイラ：インテル Parallel Studio XE 2015 Composer Edition for C++ Linux
MPSS（後述）：MPSS 3.4.2

以下では，この環境を前提に解説する．

1．アクセラレータを搭載した PC を購入する
アクセラレータには，メモリ容量とコア数によって，57コア＋6Gバイト，60コア＋8Gバイト，61コア＋16Gバイトの3種類の製品が提供されている．アクセラレータを動かすホスト PC には，Xeon プロセッサが必須である*3.

2．ホストコンピュータに OS をインストールする
アクセラレータを動作させるためのソフトウェア（デバイスドライバ等）は，Manycore Platform Software Stack（MPSS）と呼ばれる．MPSS でサポートされる OS（Linux，又は Windows）のバージョンには制限があるので，それに適合する OS をインストールする．

3．インテル製コンパイラをインストールする
アクセラレータ上で，Xeon プロセッサ用にコンパイルされたバイナリがそのまま動く訳ではなく，アクセラレータ用にコンパイルし直す必要がある．gcc を用いることもできるが，現状ではパフォーマンスに問題があるので，インテル製コンパイラを使用するのが良い．

4．MPSS をインストールする
MPSS に添付される README ファイルに従って作業を進める．RPM パッケージの形式でドライバ群が提供されるので，問題が発生することはほぼ無いと思って良い．

5．MPSS の初期設定
MPSS のインストールが完了したら，
・初期化をして，アクセラレータ用のまっさらな Linux 環境をホスト PC 上に構築
・ユーザを追加
・ホスト PC とアクセラレータとのファイルの受渡しのための NFS（Network File System）を設定
などを，ホスト PC 側から `micctrl` コマンドを使用して行う．一般的な Linux PC の管理を行った経験があれば，15分程度で作業は終了する*4．なお，シェル上でホスト PC 側からアクセラレータにリモートログインする際に，いちいちパスワードを入力するのは面倒なので，公開かぎ認証を用いてパスワード無しでログインできるようにしておくと便利である．

6．アクセラレータの起動
「アクセラレータの起動」とは，アクセラレータ用の OS をアクセラレータ内で起動することを意味する．ホスト OS にサービスとして MPSS が登録されているので，OS 環境に応じて*5,

```
# service mpss start
```

または，

＊3　Core プロセッサでは動作が保証されていない．
＊4　NFS の設定の際にホスト PC 側のファイアウォールの設定変更を忘れないように！
＊5　"#"は管理者権限（root）のプロンプト，"$"は一般ユーザのプロンプトを表す．

```
# systemctl enable mpss
```

として，アクセラレータを起動する．概ね 1 分程度で起動する．

7．プログラムの準備

OpenMP（どのように並列化するかをコンパイラに指示する規約．次章参照）でコーディングされたソースファイル（今は C 言語を仮定する）を準備し，インテル製の C コンパイラを

```
$ icc -mmic simulation.c -o simulation.x
```

と起動し，アクセラレータ用のバイナリを生成する．コンパイル環境と実行環境が同一の場合，インテルコンパイラは-xHOST をオプションに指定することにより，当該システムのCPUのベクトル演算能力を最大限発揮できるオプションを自動設定する．アクセラレータの場合は，-xHOSTの代わりに，-mmicを指定し，アクセラレータ用のバイナリを生成する．生成されたバイナリは，アクセラレータとNFSで接続されたディレクトリにコピーする．また，アクセラレータ内に存在しない共有ライブラリもあるので，ldd コマンドを使用し，足りない共有ライブラリを調べ，それらのファイルもホスト PC からバイナリが置かれているディレクトリにコピーする．

8．実行

アクセラレータに ssh でリモートログインし，先ほどコピーしたファイルを実行する．Linux のシェル上で普通にプログラムを起動するのとまったく同じである．

```
$ ./simulation.x
```

以上が，アクセラレータの導入，およびシミュレーションの実行過程の概要である．OpenMP でコーディングされたコードを持っているのであれば，GPU よりも気軽に始められる点を，改めて強調したい．

1.3 シミュレーション例

ここでは，非中性プラズマに対応した点渦系シミュレーションの紹介を行う．点渦系は，渦度方程式(6)を粒子の集合体で離散化するモデルの一つであり，計算機シミュレーションでもよく使われる技法である．本節では，アクセラレータを使用すると，どの程度の N 体シミュレーションが可能になるのか，視覚的に示す．シミュレーション速度に関する定量的な評価は，次章に掲載する．非中性プラズマ実験に関する解説は[5]と重なる部分もあるが，読者の便宜のため，特に点渦系との対応に関する部分のみ，ここで解説する．

1.3.1 非中性プラズマと点渦シミュレーション

プラズマの電気的中性条件を破ったプラズマが非中性プラズマであり，最も極端な例の一つが電子のみから構成された純電子プラズマである[6]．電子プラズマを閉じ込める装置としては，円筒真空容器の軸方向に磁場，軸方向両端に負電位を印加する Penning-Malmberg トラップが使われる．

軸方向に印加された磁場，および電子と導体壁との間の誘導電場中に置かれた電子の運動方程式は，電子の質量を m_e，負号をつけた素電荷を $-e$ とすると，

$$m_e \frac{\mathrm{d}\boldsymbol{v}}{\mathrm{d}t} = -e(\boldsymbol{E}+\boldsymbol{v}\times\boldsymbol{B}) \tag{1}$$

と書ける．高周波のサイクロトロン運動を落とすため，左辺の時間微分を落とし $\boldsymbol{v}$ について整理すると，

$$\begin{aligned}\boldsymbol{v} &= \frac{\boldsymbol{E}\times\boldsymbol{B}}{|\boldsymbol{B}|^2} \\ &= \frac{\boldsymbol{E}\times\hat{\boldsymbol{z}}}{B_0}\end{aligned} \tag{2}$$

を得る．ここで，円筒容器の軸方向を z 軸にとり，磁場を z 方向の単位ベクトル $\hat{\boldsymbol{z}}$ を用いて $\boldsymbol{B} = B_0\hat{\boldsymbol{z}}$ とした．さらに，電場を静電ポテンシャルを用いて $\boldsymbol{E} = -\nabla\phi$ とすると，

$$\boldsymbol{v} = \frac{\hat{\boldsymbol{z}}\times\nabla\phi}{B_0} \tag{3}$$

が得られる．この式の形と 2 次元流に対する流れ関数の形との類推から，

- サイクロトロン運動を無視した電子の流れは，非圧縮（$\nabla\cdot\boldsymbol{v}=0$）である
- 静電ポテンシャル ϕ と流れ関数 ψ には，$\psi = \phi/B_0$ なる比例関係が成り立つ

ことがわかる．(3)式の rot をとると，渦度を ω_z として，

$$\begin{aligned}\omega_z\hat{\boldsymbol{z}} &= \nabla\times\boldsymbol{v} \\ &= \frac{\nabla^2\phi}{B_0}\hat{\boldsymbol{z}} \\ &= \frac{en}{\varepsilon_0 B_0}\hat{\boldsymbol{z}}\end{aligned} \tag{4}$$

となる．ここで，最後の式変形で，Poisson 方程式を使った．n は電子の数密度である．この式から，

- 渦度 ω_z は，電子の数密度 n に比例する

ことが分かる．これにより，電子の輝度分布を測定することは，すなわち，渦度を測定することと同等となる．さらに，電子の数密度は空間中での生成消滅がないので，連続の式

$$\frac{\partial n}{\partial t} + \boldsymbol{v}\cdot\nabla n = 0 \tag{5}$$

を満たす．この式の両辺に $e/(\varepsilon_0 B_0)$ をかけると，

$$\frac{\partial \omega_z}{\partial t} + \boldsymbol{v}\cdot\nabla\omega_z = 0 \tag{6}$$

なる渦度方程式が得られ，電子の数密度 n の時間発展方程式は，2 次元非圧縮非粘性の Euler 方程式と同等であることが示される．

以上より，サイクロトロン運動を無視した電子の運動を数値的に追跡するためには，(6)式を解けば良いことがわかる．このために，点渦法を用いる．点渦法は，渦度をDirac のデルタ関数$\delta(\boldsymbol{r})$で離散化する．$\boldsymbol{r}_i$，Ω_i は，i 番目の点渦の位置ベクトルと循環（強度）である．

$$\omega_z(\boldsymbol{r})=\sum_i \Omega_i \delta(\boldsymbol{r}-\boldsymbol{r}_i) \qquad (7)$$

$\boldsymbol{r}_i$ の時間発展方程式は，以下のBiot-Savart積分によって与えられる．これは，(7)式を(6)式に代入すると得られる．

$$\frac{\mathrm{d}\boldsymbol{r}_i}{\mathrm{d}t}=-\frac{1}{2\pi}\sum_{j\neq i}\Omega_j\frac{(\boldsymbol{r}_i-\boldsymbol{r}_j)\times\hat{\boldsymbol{z}}}{|\boldsymbol{r}_i-\boldsymbol{r}_j|^2}+\frac{1}{2\pi}\sum_j\Omega_j\frac{(\boldsymbol{r}_i-\overline{\boldsymbol{r}}_j)\times\hat{\boldsymbol{z}}}{|\boldsymbol{r}_i-\overline{\boldsymbol{r}}_j|^2} \qquad (8)$$

なお，ここで半径 R の円筒境界の効果を $\overline{\boldsymbol{r}}_j=R^2\boldsymbol{r}_j/|\boldsymbol{r}_j|^2$ に置いた鏡像渦で取り入れた．(8)式の和は，全点渦に関してとる．つまり，すべての点渦に対する $\mathrm{d}\boldsymbol{r}_i/\mathrm{d}t$ を求めるためには点渦数の2乗に比例する時間計算量を必要とするため，1000粒子を超えるシミュレーションとなると，速い計算機を使いたくなってくる．我々は，このBiot-Savart積分を高速に計算できる計算機として，Intel® Xeon Phi™ コプロセッサ（アクセラレータ）を選択した．

1.3.2　結果

ここでは，アクセラレータを使用して，どの程度の規模の点渦シミュレーションが可能か，視覚的に提示する．具体的な計算速度等は，次章で解説する．

非中性純電子プラズマで見られる有名な不安定性に，diocotron 不安定性がある．Kelvin-Helmholtz 不安定性と言った方が通りが良いかもしれない．磁場に垂直な断面内で電子の初期分布がドーナツ形状*6をしていると，シア流（せん断流）により，ドーナツは幾つかの塊に分裂する．ドーナツが分裂する塊の個数（線型不安定なモード）などは線型安定性解析により細かく調べられており，シミュレーションが定性的／定量的に正しい計算を行っているかチェックする対象として最適なので，ここでは diocotron 不安定性に関するシミュレーション結果を掲載する．

点渦シミュレーションのパラメータを**表1**に示す．ドーナツの外側半径と1点渦あたりの循環，点渦の数密度を一定にして，ドーナツの内側半径をパラメータとした．時間発展は，(8)式で追跡する．長さの単位は，壁半径R_0で規格化している．特徴的時間スケールについて，24.7という値は，半径 $0.5R_0$ の円内にドーナツと同じ数密度で点渦を一様に配置した渦塊の自転周期から求めた．時間発展は，この自転周期と同じ時間刻みで，$T=200$ まで行う．すなわち，全シミュレーション時間中に渦塊は8回程度自転運動する．

表1　シミュレーション設定．

ドーナツの外側半径 R_1	$0.5R_0$
ドーナツの内側半径 R_2	$0.3R_0$，$0.4R_0$
鏡像点渦を除いた実点渦数	6141，3382
特徴的時間スケール T_0	24.7

時間発展結果を**図2**（内側半径$=0.3R_0$），**図3**（内側半径$=0.4R_0$）に示す．**図2**では，$T=20$ 頃にドーナツが崩れ始め，モード3が励起されたのち，$T=60$ には熱平衡分布を目指し，中央にピークを持つ一山分布へと遷移し始める．その後，少しずつ，軸対称な分布を目指して緩和を続ける．一方，**図3**では，**図2**と同様に，$T=20$ 頃にドーナツが崩れ始め，モード6が励起される．これを渦結晶配位と呼ぶ．その後，すぐに熱平衡分布をめざすのではなく，しばらく六つの渦塊が対称的な配位を保ったまま公転を続ける．その後，$T=80$ での右下の渦塊群のように，ある二つの渦塊のすそ野同士の接触事故がおこると，渦塊同士の融合が起こり，6→5→4→…と渦塊数が減少し続け，一山分布の熱平衡解を目指す．しかし，$T=160$ 以降，一旦，渦結晶配位に比べて軸対称性が高い配位に落ち着くと，緩和速度が遅くなり，十分緩和が進む前にシミュレーションは終了する．

線型不安定性解析では，壁半径，ドーナツの外側，及び内側半径から線型不安定なモードを決定することができ

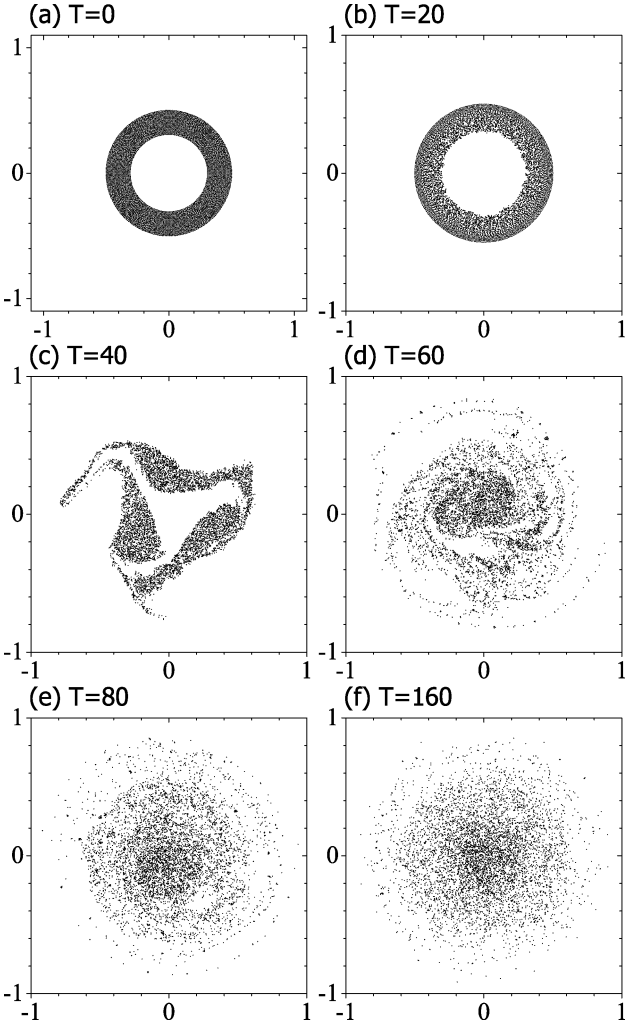

図2　時間発展結果．ドーナツの内側半径=$0.3R_0$ の場合．

＊6　3次元的にはちくわ状である．ちくわを包丁で切った断面がドーナツ形状になる．

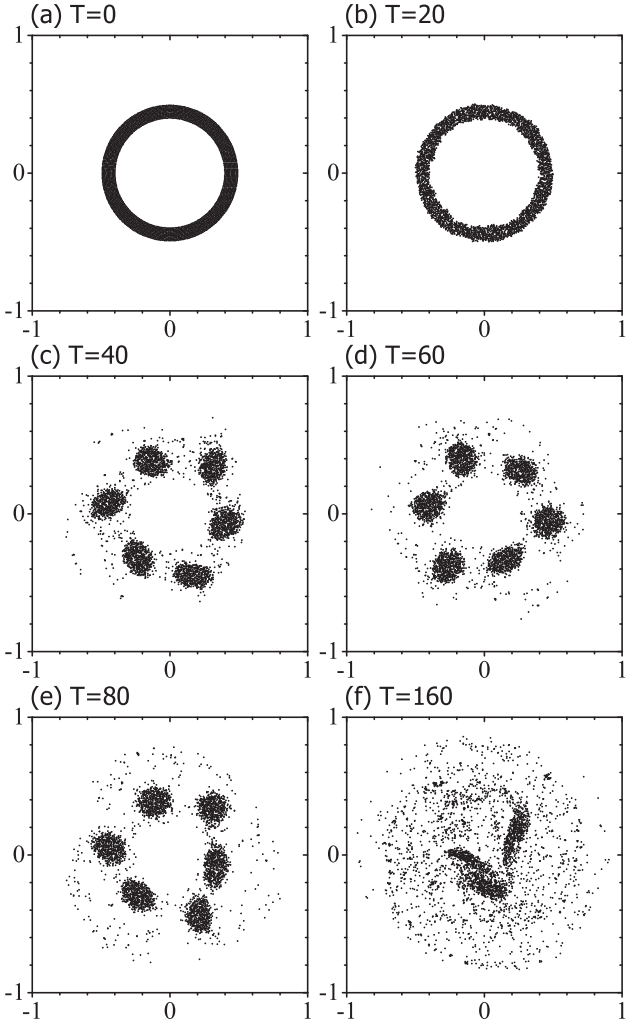

図 3　時間発展結果．ドーナツの内側半径=0.4R_0 の場合．

る*7．シミュレーション設定に対応した線型不安定なモードを不安定な順に列挙した結果を，**表 2** に示す．**表 2** から，**図 2** での最も線形不安定なモードは 3 であり，この結果を見る限り，定量的に正しい計算を行っている証拠の一つとなる．しかし，**図 3** での最も線形不安定なモードは 5 になるはずだが，シミュレーションではモード 6 が現れている．このような現象は，初期の点渦配置で，周方向の点渦数が，たまたま 6 の倍数になっていたりすると現れることがある[7]．乱数を用いて特定のモードにエネルギーを注入しないように注意をしないと，このような問題が発生することを，教育的な例として指摘したい．

図 2，**3** に掲載したシミュレーションの実行時間は，それぞれ518分，183分である．本シミュレーションでは，境界条件を鏡像渦により表現しているため，Biot-Savart 積分で加算を行う粒子数は，それぞれ 2 倍の12282粒子，6764粒子である．長年，専用計算機MDGRAPEシリーズで点渦シミュレーションを行ってきた経験からすると，N 体シミュレーション研究に十分耐えるだけの速度が実現されていると感じる．

表 2　線型不安定なモード．

内側半径 R_2	線型不安定なモード
$0.3R_0$	3，2
$0.4R_0$	5，4，6，3，2

さらに，物理的に正しい結果を得ていることの確認のため，上記時間発展結果に対して，H 関数の時間発展を追跡した結果を**図 4** に示す．有限個の正方格子で点渦の配位空間を区切り，i 番目のセルに含まれる点渦数を n_i で表すと，H 関数の値は次式で与えられる．

$$H = \sum_i n_i \log n_i \qquad (9)$$

H 関数の一般的性質として，その値は時間と共に広義単調減少し，熱平衡に達した段階で $\mathrm{d}H/\mathrm{d}t = 0$ となる．また，準安定状態などに系がトラップされても，$\mathrm{d}H/\mathrm{d}t = 0$ となるため，系が現在どのような状態にあるのか，系の緩和に関する貴重な情報源となる．**図 2** のシミュレーションに対応する H 関数の時間発展は，$T = 40$ 前まで，ほぼ一定値を保っているが，その後単調減少する．これは $T = 60$ 以降の軸対称性が高い分布において，熱平衡状態へ向けた緩和が進んでいることを表している．一方，**図 3** のシミュレーションに対応する H 関数の時間発展は，$T = 30$ 程度で値が減少するが，すぐに一定値を保つようになる．これは，準安定な渦結晶配位に留まっていることを表す．その後，値が減少に転じているのは，渦結晶配位が崩れ，熱平衡状態への緩和が進んでいることを表す[8]．

1.4　まとめ

本章では，アクセラレータの特徴，および使い方について簡単に解説を行うと共に，どの程度の N 体シミュレーションが可能であるか，視覚的に示すことを目的として，非中性プラズマ実験に対応した点渦シミュレーションの結

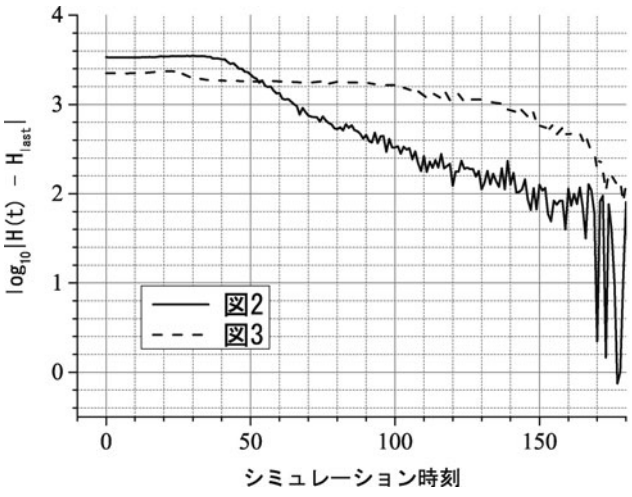

図 4　H 関数の時間発展．

＊7　Davidson の教科書 [6] に掲載されている線型不安定性解析では，電子分布の内側にも導体壁がある，すなわち，2 重連結領域内に電子群を閉じ込めるような設定を許容しているが，今回のシミュレーションでは内側には導体壁が存在しないので，上記三つのパラメータのみに依存する，と書いている．

果を例示した．これから，境界を表現する鏡像粒子込みで数万粒子程度の点渦シミュレーションであれば，アクセラレータにより十分実行可能であることがわかる．地方国立大学の1研究室レベルで保有可能なマッシブな計算環境としてGPU以外の選択肢が増えたことは良いことだろう．

参考文献

［1］SE編集部：僕らのパソコン30年史（翔泳社，東京，2010）．
［2］坪山博貴：CPUの謎（ソーテック，2005）．
［3］伊藤智義：スーパーコンピューターを20万円で創る（集英社（新書），2007）．
［4］杉本大一郎（編）：専用計算機によるシミュレーション（朝倉書店，1994）．
［5］際本泰士：プラズマ・核融合学会誌 **77**, 338 (2001) .
［6］R.C. Davidson, *Physics of Nonneutral Plasmas* (Addison-Wesley, Redwood city, CA, 1990).
［7］Y. Yatsuyanagi *et al.,* Phys. Plasmas **10**, 3188 (2003).
［8］Y. Yatsuyanagi *et al.,* J. Phys. Soc. Jpn. **84**, 014402 (2015).

八柳祐一（やつやなぎゆういち）

静岡大学教育学部准教授．主な研究分野：GPUやXeon Phiなどの小さくても速いコンピュータを駆使したシミュレーション研究．特に，長距離相関に支配されたN体系の自己組織化に興味があります．2011年7月号小特集の時のように我が子との写真を掲載したかったのですが，このアイデアは子どもたちによりゴミ箱へダンクシュート…．やむなく，ガ○プラと．

堀越将司（ほりこしまさし）

インテル株式会社，シニアアプリケーションエンジニア．大阪大学大学院工学研究科博士課程修了後，2006年から現職．これまでHPCの分野でスーパーコンピューター向けに　ISVや製造業，大学，研究所のアプリケーションの並列化，最適化およびベンチマークに従事してきた．

田邊恵李（たなべえり）

兵庫県立大学シミュレーション学研究科前期博士課程基盤研究室所属1年．学部の卒論では，Xeon Phiを用いて非中性プラズマの緩和過程に関するシミュレーションを点渦モデルにて行った．今年，遂に母が結婚した歳に自分が達し，就職活動と同レベルの焦りとプレッシャーを感じている．

J. Plasma Fusion Res. Vol.92, No.7 (2016) 567-572

講座 今日からはじめるメニーコアアクセラレータ

2．メニーコアアクセラレータ向けにプログラムをチューニングしよう

2. Let's Optimize a Program for Many-Core Accelerator

田邊恵李[1)]，八柳祐一[1)]，堀越将司[2)]
TANABE Eri[1)], YATSUYANAGI Yuichi[1)] and HORIKOSHI Masashi[2)]
[1)]静岡大学教育学部，[2)]インテル株式会社ソフトウェア＆ソリューション統括部

（原稿受付：2016年3月18日）

前章に引き続き，大学の研究室レベルで保有可能でありながら，マッシブな（大規模シミュレーションが可能な）計算環境を提供してくれるIntel® Xeon Phi™ コプロセッサ（アクセラレータ）の紹介を行う．本章では，一般的なシミュレーションコードのチューニング技法として，ベクトル化，並列化といった概念や，並列化に際して必要となるOpenMPの概要を紹介すると共に，アクセラレータでの性能がどのように改善されるのか，具体的な事例を引用しながら定量的な情報提供を行う．

Keywords:
OpenMP,optimization,parallelize,vectorize,N-body simulation

2.1 初めに

本章では，Intel® Xeon Phi™ コプロセッサで走るシミュレーションプログラムの開発の仕方を，前章で述べた点渦系を例に説明する[1]．以降，Intel® Xeon Phi™ コプロセッサをアクセラレータと呼ぶ．

2.2 高速化技法

ここでは，シミュレーションプログラムの高速化に使われる代表的技法であるベクトル化，並列化[2]，および，今やGPUやCPUなどを含めて一般的な高速化技法の一つとなったFMA（Fused Multiply and Add）の利用に分けて概説する．なお，並列化に関連する話題として，並列処理を行なうための宣言子であるOpenMPについても触れる[3]．

2.2.1 ベクトル化

ベクトル化とは，今まで，一つずつ順番に

$$z_1 = x_1 + y_1 \quad (1)$$
$$z_2 = x_2 + y_2 \quad (2)$$
$$z_3 = x_3 + y_3 \quad (3)$$
$$\cdots$$

という具合に処理していた演算を，

$$\boldsymbol{x} = (x_1, x_2, \cdots, x_n) \quad (4)$$
$$\boldsymbol{y} = (y_1, y_2, \cdots, y_n) \quad (5)$$
$$\boldsymbol{z} = (z_1, z_2, \cdots, z_n) \quad (6)$$

というベクトルに対する一つの演算

$$\boldsymbol{z} = \boldsymbol{x} + \boldsymbol{y} \quad (7)$$

に変換することを指す*1．（1）～（3）式での加算は“スカラ命令”と呼ばれるのに対して，（7）式での加算は“ベクトル命令”または“SIMD（Single Instruction Multiple Data stream）命令”と呼ばれる．まとめて処理できる要素数nをベクトル長と呼び，計算機（プロセッサ）によって異なる．アクセラレータでは，512ビット長のデータに対応したSIMD命令があるため，倍精度浮動小数点数（64ビット長）で8個の演算が1命令で処理できる．すなわち，（1）式のような演算を8回行うループ処理があったとすると，スカラ命令ならば8個の加算命令が必要なのに対して，ベクトル命令ならば1個の加算命令で済むため，大幅な高速化が望める．しかし，コンパイラは，ループ処理を自動でベクトル化してくれる訳ではない．なぜならば，ベクトルの要素間に依存性がないとコンパイラが判断できない時には，ベクトル化されないためである．ベクトルの要素間に依存性が「ある」とは，例えばベクトル$\boldsymbol{x} = (x_1, x_2, x_3, \cdots)$の要素$x_3$が$x_1$と$x_2$の関数となっているような場合，$x_1$と$x_2$の値が決定しない限り，$x_3$の値を決定できないことを表す．このような依存性を落とすためには，プログラムの構造や変数定義を根本的に見直す必要があるときもあるので，注意が必要である．

2.2.2 並列化

一般に，単一のオペレーティングシステムで管理される一つのシステム内での並列化と，別々のオペレーティングシステムで管理される多数のシステム間での並列化では，用いなくてはいけない技術が異なり，前者ではOpenMP，後者ではMPI（Message Passing Interface）が代表的な手法である．今回のアクセラレータは前者に該当するので，

＊1 （4）式のようなベクトルデータは，プログラミング言語上では通常，配列で表現されることに注意してほしい．

Faculty of Education, Shizuoka University, Shizuoka City, SHIZUOKA 422-8529, Japan
corresponding author's e-mail: tanabe@ikona.ed.shizuoka.ac.jp

ここでは並列化をコンパイラに指示するための規格OpenMPの概説をする．また，並列化を行う際に注意すべきこととして，スレッド間の負荷の均一化について触れる．

2.2.2.1 OpenMP

C言語やFortranには，並列処理を行うための規格が存在しない．そのため，並列処理機能を補う目的で1997年に業界標準規格としてOpenMPが発表された．OpenMPの詳細を解説するには紙面が足りないので，ここではその代表的な使い方を示すに留める．詳しくはインテル㈱が提供している「インテルC/C++コンパイラーOpenMP活用ガイド」がコンパクトながら良くまとまっているので，参照してほしい[4]．

OpenMPによるC言語でのプログラム例を示す．/*…*/は，プログラム中での注釈を表す．

```
/* (1) */
#pragma omp parallel private(i) /* (2) */
{
  #pragma omp for /* (3) */
  for (i=0; i<N; i++) {
    c[i] = a[i] + b[i];
  }
  #pragma omp for /* (4) */
  for (i=0; i<N; i++) {
    z[i] = x[i] + y[i];
  }
  #pragma omp for reduction (+:sum) /* (5) */
  for (i=0; i<N; i++) {
    sum += c[i] + z[i];
  }
} /* (6) */
```

OpenMPでは，並列処理をしない部分とする部分を行き来しながら処理を進める．まず初めに，プログラムは単一の処理の流れ（マスタースレッド）で処理が始まる（(1)の状態）．並列処理の開始を指示する宣言子に出会うと（(2)の行），マスタースレッドである本人に加えて，それらの「影武者」が生成されて，本人と影武者による並列処理が始まり，対応する閉じ括弧が現れる(6)まで続く．この影武者を，スレーブスレッドと呼ぶ．(2)の後半に指定されている“private(i)”は，並列処理中に含まれるループ処理でカウンタとして使用される変数iについて，影武者用にも独立した変数iを用意することを指示している．例えば，マスタースレッド一つとスレーブスレッド三つの計四つのスレッドで処理を行う場合，マスタースレッドのiに加えてスレーブスレッド用に3個のiの格納場所が用意される．次に，(3)ではi＝0，1，…，N－1について行われるループ処理を，影武者と協力して分散処理することを指示する．(3)の指示がないと，本人が

```
c[0] = a[0] + b[0];
c[1] = a[1] + b[1];
…
```

という処理をしている間に，影武者も全く同一の

```
c[0] = a[0] + b[0];
c[1] = a[1] + b[1];
…
```

という処理を行うことになり，並列化の意味がない．そこで，(3)の指示を書くことにより，本人はi＝0～24の部分，一人目の影武者はi＝25～50の部分，…，といった具合に負荷分散が行われる．この例からも，変数iにプライベート指定が必要な理由がよくわかるだろう．

(5)はリダクションと呼ばれる演算のための書き方で，和を求める変数sumにプライベート指定をしてしまうと，影武者たちがsumに求めた和の値を回収する機会を失う．そこで，リダクション指定を行ったループでは，当該ループ処理が終了したときに，“(+:sum)”の後ろに指定されたプライベート変数*2について“(+:sum)”の前半に指定された演算（ここでは，"+"の加算）が行われる．他にも，リダクションでは乗算などを指定することもできるが，可能な演算種には制限があるので，注意してほしい．

最後に，(6)に差し掛かると並列処理は終了し，影武者たちは消え去り，本人のみがその後の処理を継続する．以上が，OpenMPによる並列化の基本的な考え方である．並列化では，他にも，同期という重要な概念がある．これをきちんと管理しないと，走らせるたびに異なる結果が得られるといった問題が発生することになるので，文献などでチェックしてほしい[3]．

2.2.2.2 スレッド毎の負荷の均一化

例えば，以下のような4種類の処理があるとしよう．

(1) 1～10の和を計算
(2) 1～20の和を計算
(3) 1～30の和を計算
(4) 1～40の和を計算

このような処理を，2分割して二つのプロセッサで並列に計算する場合，1番目のプロセッサに(1)，(2)，2番目のプロセッサに(3)，(4)と処理を割り振ると，2番目のプロセッサの処理が終了する前に1番目のプロセッサの処理が終了し，1番目のプロセッサが遊休状態となる．すなわち，並列処理を行う場合には，プロセッサ毎の負荷を均一化しないといけないことがわかる．ベクトル化でもそうだが，計算のためのリソースをできるだけ多く活用するためにも，遊んでいるスレッドができるのは避けたい．これは，コンパイラでは判断できないため，プログラマが明示的にOpenMP宣言子scheduleを用いて指示する必要がある．通常は，“schedule（static)”が指定されたものとみなされ，各スレッドにはあらかじめわかっているループの処理回数を均等割した分の処理が割り当てられる．例えば，i＝0～99の処理を4スレッドで処理する場合には，1番目のスレッドにはi＝0～24，2番目のスレッドにはi＝25～49

＊2 リダクション演算では，reduction句の後ろの括弧内に指定された変数は，自動的にプライベートの扱いとなる．

という具合である．しかし，上記に示した例のように，各i毎に処理の重さが異なっているとiでの分割が平等でも実際の負荷は均等化されない．このような場合に“schedule(dynamic)”を指定する．この場合には，1番目のスレッドにi＝0，2番目のスレッドにi＝1，というように割り振りながら，処理の終了したスレッドから順次，次のiが指定され，処理を進める．staticの場合に比べて処理を配分する手間はかかるが，遊休状態となるスレッドが減るため，並列化効率は結果として向上する．

2.2.3 FMA

FMA（Fused Multiply and Add）は，$a \times x + b$ という1次式を高速に計算する最適化技法である．上記のような積と和の計算が1回ずつ現れる式では，本来であれば加算命令と乗算命令を1個づつ使用しなければいけないが，FMAでは1命令で加算と乗算を行うことができる．FMAの適用の可否はコンパイラが自動的に判断してくれるが，複雑なコーディングでは見逃されることがあるので，やはり100%活用するには，わかりやすいプログラムとする必要がある．

2.3 高速化の手順

本節では，点渦系のシミュレーションコードを例にとって，具体的な高速化の手順（チューニング法）を紹介する．チューニングは，3段階に分けて行う．第1段階は，研究者にとって負担の少ないコンパイルオプションのみによるチューニング，第2段階は，さらなる速度向上を狙ったプログラムの書き換えによるチューニング，そして第3段階は，第1，2段階を併用した最高速を狙うチューニングである．

ここで使用した機材／環境は，**表1**のとおりである．

2.3.1 第1段階

使用したコンパイラのオプションについて，概説する．-O0（オーゼロ）は，コンパイラによってデフォルトで行われる最適化を含めた全ての最適化を無効にするオプションである．普段使用することはないが，今回はコンパイラによってどれだけ最適化がされているかを確認するために使用した．-no-vecは，ベクトル化のみを無効化するオプションである．-no-vec以外にも個別に無効化できるオプションは多いが，アクセラレータの本質に関わりの高い最適化はベクトル化であるため，ベクトル化の有無の影響を調べた．-O3は，コンパイラによって最大限の最適化を行うことを指示する．コンパイラではデフォルトで-O2が付いているが，それよりも強い最適化が行われる．ベクトル化などに加えて，関数のインライン展開やループの展開（アンロール）など，様々な最適化がコンパイラによって自動的に行われる．-fp-modelは，浮動小数点数の演算精度を制限することで，計算速度を向上させるオプションである．今回使用した-fp-model fast＝2では，特に速度を優先した演算が行われる．計算の精度は若干落ちるが，点渦系のような多体系では大きな問題が発生する可能性が低いことを経験的に知っているため使用している．よって，計算モデルによっては使用を控えた方が良い場合もある．-no-prec-divは浮動小数点数の除算の精度を下げ，演算速度を上げることを指示する．ただし，このオプションを指定すると，IEEEで決められた浮動小数点演算規則と異なる除算が行われるため，IEEE準拠で除算を行った場合と計算結果が異なることがある．よって，-fp-modelと同様に計算結果に影響を及ぼす可能性があるので，計算モデルによっては使用を控えた方が良い場合もある．なお，-fp-modelと-no-prec-divは，-O3が付いていなければ最適化が有効にならないため注意してほしい．

表1　環境．

プロセッサ	Intel® Xeon™ E5 2670
プロセッサ数	2
プロセッサ当たりのコア数	8
主記憶容量	64GB
使用したアクセラレータ	Intel® Xeon Phi™ 7120A
OS	CentOS release 6.7
コンパイラ	Intel® Parallel Studio XE 2015 Composer Edition for C++ Linux

2.3.2 第2段階

ここでは，それぞれ

(1) パディング
(2) 配列のアライメント
(3) pow関数から2乗の掛け算への書き換え
(4) 2次元配列1本から1次元配列2本への書き換え
(5) ベクトルループ内の配列要素の変数への書き換え
(6) コア毎の計算負荷の均一化

に関する解説を行う．まず，キャッシュの利用率向上のために，(1)と(2)を行う．パディングは配列要素へのアクセスの際に発生するキャッシュの競合を緩和するために，本来必要とされる要素数に加えて，余分な要素を含めて配列領域を確保する操作のことである．余分に確保する要素数は，計算機や処理内容によって異なるので，いろいろと試す必要がある．今回，アクセラレータなしの場合は4要素，アクセラレータ有りの場合は8要素を余分な要素として追加した．配列のアライメントは，配列を宣言する際に，`All`を配列要素数を指定する定数として，

```
__declspec(align(ALGN))
          static double array[All];
```

のように行う．この指定子を使うと，配列がプロセッサからアクセスしやすい境界に揃うように要素が並べられ，効率的に主記憶からキャッシュへデータが流れるようになることが期待できる．アクセスしやすいメモリ境界は，プロセッサの設計特性に影響を受けるためマシンによって異なるが，アクセラレータの場合は64バイト境界で始まるアドレスにデータがあるとよい．

次に，(3)について．関数pow(a, b)は a^b を計算する関数である．かつては2乗の計算にpow関数を使用すると高速になることが知られていた．そこで，Biot-Savart積分に現れる2乗の計算にpow関数を使用していたが，アクセラ

レータの場合にはFMAとの関連で，普通に“a * a”と書いた方が速い場合が多いようである．よって，ここでは今まで，“pow(a, 2.0)”と書いていた部分を，“a * a”と変更する．

(4)では，配列の書き方を変更する．例えばDを次元数，Allを点渦数とすると，点渦の位置ベクトルをpos[D][All]と宣言していたものを，posx[All]，posy[All]と書き換えるように，2次元配列から二つの1次元配列に変換する．2次元配列と1次元配列を比べると1次元配列の方がより少ない間接参照で当該要素にアクセスできる．またそれ以外にもコンパイラに「配列自身に依存性がない」と判断をしてもらいやすいというメリットがある．

(5)は，ベクトル化が行われるループ内の配列要素を変数に書き換える操作である．例えば“a[i]＝b[i]*c[i]”と書いていたものを“a＝b[i]*c[i]”と書き換える．Intelコンパイラでは，ベクトル化が行われるループ内にある変数は，ループ毎に（別々の変数として）プライベートに扱われる．つまり，配列と同じように扱われるようになる．そのため本来なら配列を置かなければいけない部分を変数で置き換えることができる．変数と配列では，変数の方が間接参照の手間が一手間少ない*3．すなわち，計算の途中結果などは配列ではなく，普通の変数に格納することによりメモリアクセスを高速化できる．

(6)では，コア毎の計算の負荷を均一化するためにOpenMPの宣言子で，“schedule(dynamic)”を用いる最適化を行う．デフォルトの“schedule(static)”では，コア毎の負荷などを考慮した並列化はできないが，“schedule(dynamic)”を明示的に指定することにより，実行時に動的に仕事の配分が行われるようになるため，並列化効率が向上する．“static”と“dynamic”のどちらを使用するかは，動的に仕事の配分を行う手間と効率から判断すればよい．今回は，実行時に動的に並列化を行なうコスト以上に，スレッドへの負荷が均一化することによる速度向上の方が大きいため，dynamicを使用した．

2.3.3 第3段階

第3段階では，第2段階で書き換えたコードに対して，第1段階で効果のあったコンパイラオプション-fp-modelと-no-prec-divを付けるチューニングを行い，どの程度高速化が可能かチェックを行った．さらに，Biot-Savart積分の分母に表れる $\boldsymbol{r}_i$，$\boldsymbol{r}_j$ という二つの位置ベクトルの差 $|\boldsymbol{r}_i - \boldsymbol{r}_j|$ がゼロとなる状態を取り除くため，チューニング前は，変数iが固定された変数jに関する内側ループで，“j＝0からj＝i－1まで”と“j＝i＋1からj＝N－1”までの二つのループに分割していた．

```
for (i=0; i<N; i++) {
  for (j=0; j<i; j++)
    …
  for (j=i+1; j<N; j++)
    …
}
```

しかし，この形式だとコア毎の負荷の分散が行いにくいようので，次のように書き換えた．

```
for (i=0; i<N; i++) {
  for (j=0; j<i; j++)
    …
}
for (i=0; i<N; i++) {
  for (j=i+1; j<N; j++)
    …
}
```

これによって，コア毎の負荷分散がさらに促進され，結果としてスループットが向上する．

2.4 ベンチマーク結果

ここでは，前節で説明したチューニングを行った結果，どの程度高速化が達成できたのか定量的に示すため，ベンチマーク結果を報告する．ベンチマーク共通の条件は，粒子数が10370，ループの回数が60回である．最終的に，Xeon E5 2670 8コア×2での16並列に対して約10倍の高速化に成功した．これは，アクセラレータの理論ピーク性能の5割程度に相当し，十分満足がいく結果といえる．

2.4.1 アクセラレータを使用しない場合

ここでは，そもそも正しい挙動をするプログラムなのかを，アクセラレータを使用しないで確認する．初めに，プログラムにおいて，並列度をn倍にしたときに，処理時間が1/n になっているかを確認する．結果を，**図1**に示す．メモリのアクセス競合等が発生するため，計算時間がきっちり理論通りになることはない．それでも，スレッド数＝1の場合と比べて，スレッド数＝16の場合の処理速度は14.5倍となっており，今回作成したプログラムは，論理的に正しく並列化されていることがわかる．次に，スレッド数を16に固定し，粒子数を変化させた場合の速度変化を確認する．結果を，**図2**に示す．縦軸が時間［秒］，横軸が点渦数［個］である．今回のホットスポット*4であるBiot-

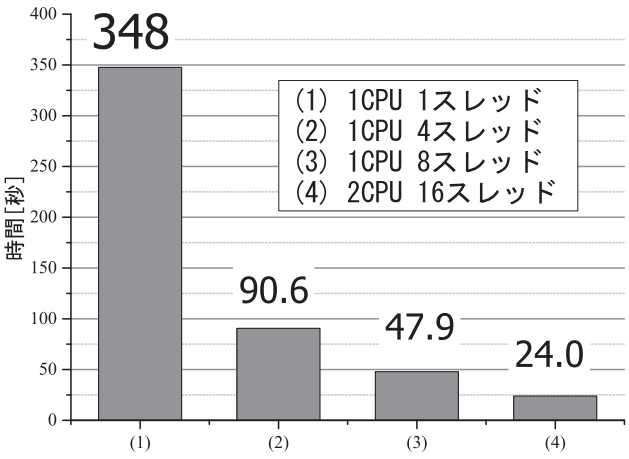

図1 スレッド数をパラメータにした速度変化．

＊3 配列は（レジスタではなく）主記憶に置かれるので，一般的に遅くなる．

＊4 プログラムで特に時間を要する重たい処理を，このように呼ぶ．

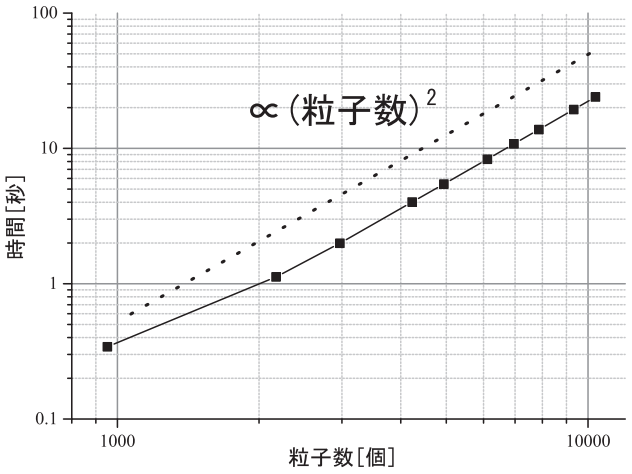

図2　粒子数をパラメータにした速度変化．点線は，N^2 に比例するシミュレーション時間の目安(傾き)を表す．

Savart 積分は，点渦数を N とすると，計算時間は N^2 に比例する．点線が，N^2 に比例する時間計算量を表しており，粒子数が多くなるにつれて N^2 に漸近していることから，Biot-Savart 積分でのループの反復回数が間違っていないことを確認できる．なお，点渦数が少ない側で N^2 のスケールからずれるのは，Biot-Savart 積分以外の部分の処理時間に律速されていることを表している．

2.4.2　アクセラレータ＋コンパイラオプション

通常のプロセッサ上で，期待通りの動作をしていることが確認できたので，ここから，アクセラレータを使用したチューニングに移る．まず，研究者にとって負担の少ないオプションのみを付加するチューニングを行う．今回使用したオプションの組み合せは，

(1)-O0
(2)-O2　-no-vec
(3)-O3
(4)-O3　-fp-model　fast＝2
(5)-O3　-no-prec-div
(6)-O3　-fp-model　fast＝2　-no-prec-div

の六つである．結果を**図3**に示す．まず，コンパイラの恩恵を確認するために，(1)-O0 と (3)-O3 の比較を行う．両者では，約260倍の速度差があるのがわかる．すなわち，260倍に加速するだけの仕事を，研究者の代わりにコンパイラがしてくれたことになる．ただ，コンパイラも万能ではないため，研究者が最適化しやすいだろうと気を利かせてコードを書くと，コンパイラがこちらの意図を正確に判断できず，最適化をしてくれないことがあるため工夫をする時には注意が必要となる．

次に，(2)-no-vec と (3)-O3 を比較する．ベクトル化の有無で速度が 6 倍変化する．アクセラレータのベクトルレジスタは512ビットなので，完全なベクトル化を達成すると両者の差は 8 倍になるはずだが，残念ながら，そこまでの速度向上は望めなかった．

さらに，-O3 に加えて付加した(4)-fp-model fast=2 と (5)-no-prec-div の結果を見ていく．-O3 と fp-model では 1.4 倍，-no-prec-div では 1.3 倍速くなる．

オプションはコンパイル時に指定するだけで良いので，研究者への負担が非常に少ないながら，オプション指定のみでアクセラレータなしでの16並列と比べて 4.1 倍の速度を実現できた．

2.4.3　アクセラレータ＋プログラム書き換え

前節で説明したプログラムの書き換えを行った場合の結果を**図4**に示す．まず，(1)～(5)の書き換えにより，書き換えを行う前で最も速い場合より 1.6 倍速くなっているのがわかる．さらに(6)の負荷の均一化を加えると，同様に 1.65 倍速くなる．速度向上効果は小さいように見えるが，粒子数が多くなるに従ってこの差は大きくなるので，書き換えを行う価値はあると考える．最終的に，(1)～(6)の書き換えにコンパイラオプション - fp-model fast＝2 と - no-prec-div を加えた場合には，(1)～(5)の書き換えのみを行った場合と比較しても 1.5 倍速くなっている．

2.4.4　最終結果

最後に，各段階でのベンチマーク結果の比較を行う．結果を**図5**に示す．プログラムを書き換え，オプションを付けてチューニングを行った(4)は，アクセラレータなしの(1)に比べて 9.6 倍の速度に達する．プログラムの書き換えなしでオプションのみのチューニングを行った場合と比べても 2.3 倍となっている．手間を掛けず，オプションのみのチューニングでも，アクセラレータの有無で速度に 4.1 倍

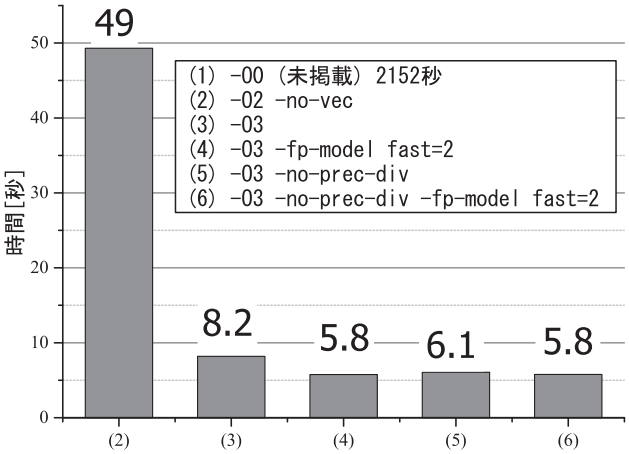

図3　コンパイルオプションをパラメータにした速度変化．

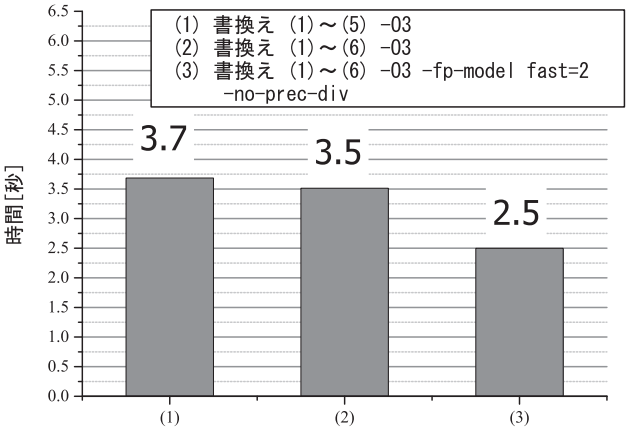

図4　プログラムの書き換えをパラメータにした速度変化．

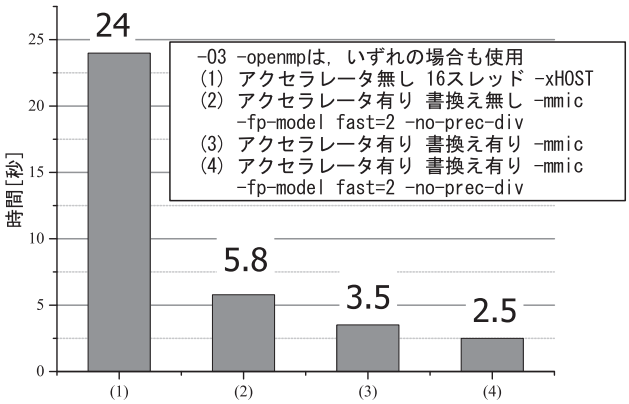

図5 最終結果.

の開きが出る．そのため，オプションのみのチューニングか，プログラム自体の書き換えを行うかは，負担を考慮した上で選択すれば良いだろう．プログラムの書き換えは研究者にとって負担ではあるが，今回プログラムに施したチューニングは，ハードウェアやプログラミングへの深い知識を必要とせず，煩雑な手間が掛かるものではない．そのため，速度差が2.3倍になることを考え合わせると，プログラム自体を書き換えるチューニングは施す価値があると考えている．

2.5 結論

アクセラレータの高速化技法には，ベクトル化，並列化，FMA 等がある．アクセラレータをより高速化させるためには，これらを最大限運用できるようにコーディングを工夫する必要がある．もし，コーディングの負担を小さくしたいのならば，コンパイラオプションだけを使うチューニング方法もある．本章では，高速化に有効なコンパイラオプションとして-no-prec-div と-fp-model fast＝2 の二つを紹介した．ただし，計算精度が落ちてしまうためモデルによっては導入を避けた方がよいかもしれない．コンパイラオプションだけでも速度が向上するが，さらなる高速化を期待するならば，プログラムを書き換える必要がある．まず配列の境界を64バイトの倍数に揃えることで，メインメモリへのアクセス効率向上を狙う．次に，各種演算式を，できる限り，乗算と加算の組合せに書き換えて FMA が最大限利用できるようにする．また，1本の多次元配列を複数の1次元配列に書き換えると速くなることがある．ベクトルループ内の変数は，ループ毎にプライベートに扱われるということを利用して配列を変数に書き換えてもメモリアクセスの効率を良くすることができる．最後に OpenMP 宣言子で schedule（dynamic）を用いてコア毎の負担の均一化を行う．コーディングによるチューニングとコンパイラオプションによるチューニングは併用可能であるため，この両者を用いると更なる速度向上が期待できる．

参 考 文 献

［1］J. Jeffers, J. Reinders,（訳）すがわらきよふみ，エクセルソフト株式会社：インテル Xeon Phi コプロセッサーハイパフォーマンス・プログラミング（カットシステム，東京，2014）.
［2］D.A. Patterson,（訳）成田光彰：コンピュータの構成と設計第5版（上下）（日経 BP 社，東京，2014）.
［3］北山洋幸：OpenMP 入門（秀和システム，東京，2009）.
［4］インテル㈱：http://jp.xlsoft.com/documents/intel/compiler/526J-001.pdf.

J. Plasma Fusion Res. Vol.92, No.8 (2016) 612-622

講座 今日からはじめるメニーコアアクセラレータ

3．IFERC-CSC におけるメニーコアアクセラレータとその利用例

3. The Many-Core Accelerator and Examples of Its Use at IFERC–CSC

中島徳嘉，宮戸直亮[1)]，内藤裕志[2)]，佐竹信一[3)]，保坂和樹[3)]，功刀資彰[4)]
NAKAJIMA Noriyoshi, MIYATO Naoaki[1)], NAITOU Hiroshi[2)], SATAKE Shin-ichi[3)], HOSAKA Kazuki[3)] and KUNUGI Tomoaki[4)]
核融合科学研究所，[1)]量子科学技術研究開発機構，[2)]山口大学大学院創成科学研究科，
[3)]東京理科大学基礎工学部，[4)]京都大学大学院工学研究科

（原稿受付：2016年5月20日）

最近の大型並列計算機では，性能向上および省電力化のため，一つのチップに多数の演算コアを搭載する傾向にある．青森県六ヶ所村にある国際核融合エネルギー研究センター（International Fusion Energy Research Centre; IFERC）の3副事業の一つである計算機シミュレーションセンター（Computational Simulation Centre; CSC）の大型並列計算機 Helios では，CPU のみの本体システムに加えて，多数の演算コアを持つコプロセッサを搭載する増強システムを運用している．本章では，この増強システムの利用環境，増強システムを実際に利用した2次元 PIC シミュレーションおよび MHD 乱流の直接数値計算について紹介する．

Keywords:
BA agreement, IFERC, CSC, simulation project, coarse-grained parallelism, fine-grained parallelism, streaming algorithm, direct numerical simulation, MPI+OpenMP, MIC

3.1 IFERC-CSC の MIC システムの環境紹介

国際核融合エネルギー研究センター（International Fusion Energy Research Centre; IFERC）の計算機シミュレーションセンター（Computational Simulation Centre; CSC）は日欧間のより広範な取組を通じた活動（Broader Approach; BA）協定に基づいて実施されている事業で，青森県六ヶ所村に大型並列計算機 Helios（愛称：六ちゃん）が設置され運用されている．この Helios 計算機は本体システム（4500ノード）と増強システム（180ノード）から成り，増強システムに Intel 社のメニーコア（Many Integrated Core; MIC）アーキテクチャをベースにした Xeon Phi^{TM} コプロセッサが搭載されている．このため増強システムは MIC システムとか MIC ノードと呼ばれている．

Helios 計算機の導入の経緯などは2015年11月号掲載の解説記事[1]に詳しいのでそちらを参照されたい．ここでは MIC システムの環境について簡単に説明する．MIC システムの1ノードは，2つの Intel 社製 CPU（Xeon，Sandy-Bridge EP，8コア，2.1 GHz）および2つの Xeon Phi コプロセッサ（Xeon Phi 5110P，60コア，1.05 GHz）と，合計で64GB のメモリを搭載している（**図1**）．MIC システム全体としての理論ピーク性能は412TFLOPS で，linpack 性能として 225.1 TFLOPS が得られているが，これは2015年11月の TOP500 リスト［2］で456位に位置する．この MIC システムは，本体システムとファイルシステムを共有し，高速・並列なデータの入出力が可能となっている．

MIC システムではプログラミングツールとして，OpenMP（Open Multi-Processing）4.0 を含む Intel コンパイラ（C/C++，FORTRAN），Allinea DDT，Intel VTune といったデバッガが利用可能である．また，Intel MKL，FFTW，PETSc，HDF5，MAGMA MIC，NAG，NetCDF という数値計算でよく使われるライブラリ類も MIC システム用のものが用意されている．Message Passing Interface（MPI）ライブラリとしては本体システムと同様に，IntelMPI と BullxMPI の2つが利用可能となっている．

ユーザーは，**図2**に模式的に示されているように，(1)

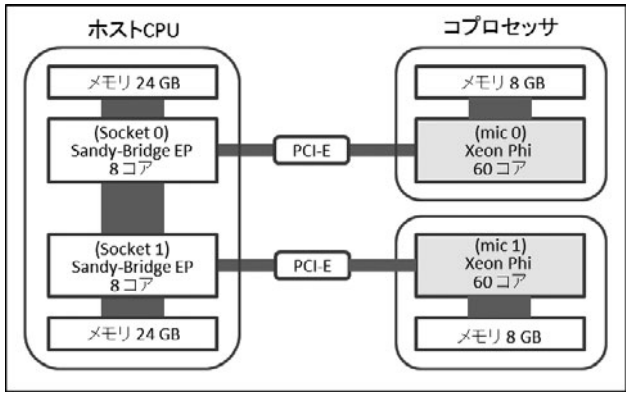

図1　1つの MIC ノードの構成．

corresponding author's e-mail: nakajima @nifs.ac.jp

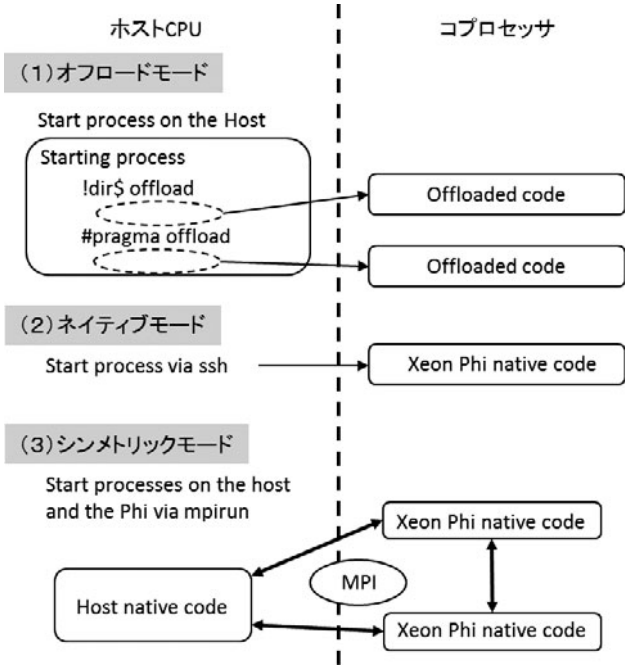

図2　コプロセッサを利用する3つのモード.

オフロードモード，(2)ネイティブモード，(3)シンメトリックモードの3つの形態でMICシステムのコプロセッサを計算に用いることができる.

(1) オフロードモード
MICノードの中のホストCPUでプロセスを実行し，コードの中で指示行により明示した部分だけ，コプロセッサを利用するモード.

(2) ネイティブモード
コンパイラでコプロセッサ用のバイナリを作成し，コプロセッサのみで計算を行うモード.

(3) シンメトリックモード
MPIによってホストCPU，コプロセッサで複数のプロセスを実行し，計算を行うモード．このモードでは異なるノード上のMIC（Xeon Phi）同士でもホストを介さずにMPIを使った直接通信が可能.

このようにMICシステムの利用法には多様性があり，それに応じて，プログラム作成の難易度やコードの実行速度も変化することになる．ユーザーの方々が今後MICシステムを活用する際の具体的参考例として，3.2節では主としてネイティブモードを用いた2次元PICシミュレーションを内藤が，3.3節ではシンメトリックモードを用いたMHD乱流の直接数値計算を佐竹，保坂，功刀が紹介する．これらの具体例を参考として，MICシステムの効率的な利用が広まることを期待している.

3.2　2次元PICシミュレーション

本節では，2次元PIC（Particle-In-Cell）コードの並列化を例にXeon Phi対応プログラム作成の指針と応用例について解説する．説明は，IFERC-CSCのMICシステムで使用されているXeon Phi 5110Pを具体的な対象としているが，その他のXeon Phiについても本質は同じである．Xeon Phiのプログラミング方法についての一般的な解説書としては文献[3]があり，本章でも参考にした.

3.2.1　Xeon Phiでのスレッド並列化の指針

科学技術計算の高速化には，並列化が必須である．並列化には，プロセス並列とスレッド並列がある．プロセス並列は分散メモリモデルに対応しており，各プロセスは独立なメモリを持ち，独立に動くが，必要に応じて，異なるプロセスの持つメモリ情報を交換（通信）したり，同期を取ったりする．通常はMPIを用いてプログラムする．スレッド並列は共有メモリモデルに対応しており，各スレッドから共通のメモリ空間への読み書きができる．通常マスタースレッドが全体を統括し，必要に応じて，自分を含む複数の管理下のスレッドに実行命令を送る．並列化には，自動並列化かOpenMPを用いる．複数のコンピュータ環境を利用する場合は，可搬性に優れたOpenMPの利用が推奨される.

単一のパーソナルコンピュータ（PC）のユーザーは，マルチコアの性能を引き出すために，プロセス並列かスレッド並列かのどちらか一つを採用する．プロセス並列を利用する場合，共有メモリは仮想的に分散メモリとして利用される．PCクラスタやスーパーコンピュータのユーザーは，純粋なプロセス並列モデルを用いるか，プロセス並列とスレッド並列のハイブリッドモデルを用いる．スレッド並列部分で計算速度の十分な高速化を得るためには，スレッド間の独立性の極めて高い（場合によっては新しい）アルゴリズムを採用することが必須である.

アクセラレータとしてGraphics Processing Unit（GPU）やXeon Phiを用いる場合は，スレッド並列が唯一の選択肢である．プログラム開発をできるだけ効率良くするためには，マルチコアシステムで並列性の高い（コア数に比例して高速化される）アルゴリズムを採用したプログラムをOpenMPで書き，デバッグを済ませておくことが重要である．こうしたプログラムを標準プログラムにして，GPUの場合は，例えばCUDA FORTRANを用いて書き換え，Xeon Phiの場合はそのまま利用（ネイティブモードの場合）するか，指示行を追加することにより書き換える（オフロードモードの場合）．オフロードモードの場合は，60コアの内1コアはホストとの通信に使用されるため，使用できるコアは59コアになる．1コアは4ハードウェア・スレッドに対応するため，1個のXeon Phiで，ネイティブモードの場合は240スレッドまで，オフロードモードの場合は236スレッドまでの計算が可能になる.

ハイブリッド並列の場合は，MPIを併用する．ネイティブモードの場合は，シンメトリックモードを用いる．この場合60コアの内，1コアはホストとの通信に使用されるため，スレッド並列で使用できるのは59コアである．オフロードモードの場合は，そのままMPIを併用することができる.

GPUをアクセラレータとして使用するのと比較して，Xeon Phiを利用する最大の利点はマルチプロセッサ対応のプログラムがあり，それが十分な並列化性能を持っていれば，ほとんど無修正でXeon Phiで実行できることである.

ただし，プログラムの一部に並列化が不可能もしくは並列性に著しく劣る部分があって，全体を並列実行した場合にその部分の実行時間が無視できない場合は，オフロードモードを用いる．この場合，ホストプロセッサと Xeon Phi コプロセッサでそれぞれ得意な分野の計算を分担することになる．

3.2.2 Xeon Phi での PIC コードの並列化

ここでは，2 次元 PIC コードを例にネイティブモードでの Xeon Phi の利用について述べる．PIC コードは，荷電粒子の運動を，荷電粒子自身が作る電磁場（もちろん外部磁場や外部電場を含めて良い）による電磁力のもとで追跡することにより，プラズマのシミュレーションを行う．第一原理シミュレーションであるため，コンピュータに対する負荷は巨大であるが，最新のコンピュータ環境を利用することにより，核融合プラズマや実験室プラズマの物理現象を解明したり予測したりする重要な武器になっている．PIC コードによるシミュレーションの教科書としては文献[4]がある．また，本誌にも講座[5]がある．

PIC コードでは，粒子と場の量を取り扱う．粒子とはプラズマを構成する電子やイオンの荷電粒子を示し，シミュレーション空間を自由に動きまわる．場の量は電場と磁場を示し，空間メッシュの格子点でのみ計算される．ここでは，直角座標で，各方向で等間隔にメッシュが構成されているとする．空間メッシュを構成する最小単位を**セル**と呼ぶ．

任意の時間で任意の荷電粒子を取り出すと，空間メッシュ上のどこかのセル内に存在する．この荷電粒子が感じる場の量は，近傍の格子点の場の量からの内挿で計算される．線形補間を用いる場合はセルの頂点の場の量のみが使用される．以下では線形補間の場合に限定して解説する．粒子の位置と速度は，ニュートン・ローレンツの運動方程式を差分化したものを用いて 1 時間ステップ進める．この過程を **PUSH** と呼ぶことにする．PUSH では，粒子番号の順に処理を進めるが，各粒子が引用する場の量へのアクセスはメモリ空間でランダムになる．このことは，キャッシュの有効利用という点で障害になり，コードの高速化性能の劣化の原因になる．また，この部分を並列化した場合，メモリへの同時アクセスが生じやすく，速度低下の原因になる．

場の量を計算するためには，格子点上の電荷密度と電流密度を計算する必要がある．各粒子の電荷と電流を，セルの頂点に対応する格子点に分配する．分配公式としては，逆線形補間を用いる．この過程を **SOURCE** と呼ぶことにする．SOURCE では，場の量にランダムにアクセスし，データを書き換える（加算する）必要があり，高速化の妨げになっている．また，並列化した場合は，同時に同一のデータを書き換える場合の結果は保証されていない．ハードウェア的またはソフトウェア的に並列化することは可能であるが，速度低下の原因になっている．

電荷密度と電流密度から場の量を計算する過程を **FIELD** と呼ぶことにする．PIC コードは，SOURCE-FIELD-PUSH を順に計算することにより 1 時間ステップ進めることができる．

従来，PIC コードの並列化には，粗粒度の並列性（coarse-grained parallelism）が用いられてきた．これはプロセス並列に対応している．場の量に複数のコピーを使用する粒子分割や，場の量を複数の領域に分割する領域分割を用い，主に MPI を用いてプログラムされる．詳しくは本誌の講座[6]を参照されたい．先に示したように，従来の PIC コードの手法ではスレッド並列の部分の並列化性能を引き出すのは困難である．筆者の経験では，すべてプロセス並列でするよりは，2 スレッドか 4 スレッドのスレッド並列を併用したハイブリッド並列の方がわずかに高速であった．

マルチスレッドやアクセラレータに対応して，PIC コードの性能を引き出す手法の一つに，細粒度の並列性（fine-grained parallelism）を利用する方法がある．ここで解説する手法は，Decyk と Singh による論文[7,8]に詳しく記述されている．これらの論文では，GPU での高速化がメインテーマであるが，マルチコアやメニーコアへの拡張性についても議論されている．

PIC コードでは，任意の粒子は，その粒子の存在するセルのみと相互作用する．このため，粒子全体を一つの配列で表すのではなく，セルごとの配列を用意する．これは粒子がセルごとにソートされていることに対応している．また，全体の場の量を表すメッシュ配列に加え，セルごとの配列を用意する．SOURCE と PUSH の計算では，従来の PIC コードが粒子番号に対応する DO ループで表されたのと比較して，新しいアルゴリズムではセル番号に DO ループが対応する．また内側の DO ループとしてセル内の粒子に対するループを追加する．各セル内の粒子の総数を与える配列も用意する必要がある．計算はセルごとに独立であるため並列性は極めて高い．GPU の場合は，1 スレッドが 1 セルに対応している．Xeon Phi の場合は，複数のセルを 1 スレッドで対応することになるが，セル単位で独立性があるため，スレッドごとの独立性も保証されている．また，メモリーアクセスの場合の局所性も保証されるのでキャッシュの性能も引き出しやすい．

新しいアルゴリズムを文献[7,8]ではストリーミング・アルゴリズムと呼んでいる．PUSH の処理を終了すると，セルの境界を超える粒子を，新しく粒子の所属するセルへ移動させる過程（**SORT**）が追加される．1 時間ステップに，複数のセルを横切る粒子はないと仮定して問題ない．

以上の並列化をセル並列と呼ぶ．セル並列の場合，SOURCE と PUSH の並列化性能は優れているが，追加された SORT の計算時間が無視できなくなる．この問題を解決するために複数のセルをまとめて**タイル**としたタイル並列を利用する．

2 次元でのセルとタイルの例を**図 3** に示す．タイルの例では x 方向 4（$mx = 4$），y 方向 4（$my = 4$）の場合を示している．タイルを利用するとセルの場合と比較して，全体として，領域境界を横切る粒子の数が減るため計算時間が短縮される（タイルを使用すると，タイル境界を除くタイル内のメッシュを横切る粒子に対する SORT の処理が必要なくなると考えれば理解しやすい）．OpenMP で書かれた

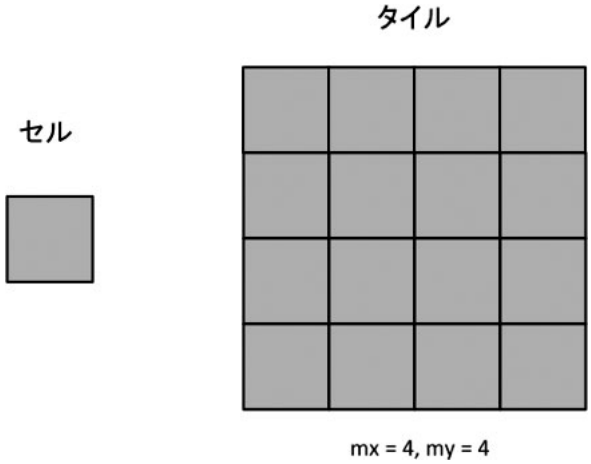

図 3　2 次元でのセルとタイルの例.

SORT のプログラム例が文献[8]に記載されている．タイルを利用した場合，タイルサイズが大きくなるにつれて，SOURCE と PUSH の計算時間は増大する傾向にある．このため，タイルサイズには最適値がある．

3. 2. 3　Xeon Phi での 2 次元 PIC コードの実行例

ここでは，2 次元静電近似コードの Xeon Phi での実行例について記す．プログラムは FORTRAN で書かれ，OpenMP 4.0 により並列化されている．場の量は，256×256 のメッシュで表される．各セル当たりの粒子数は平均して 100個，全粒子数は6,553,600である．x と y の両方向には周期境界条件を用いる．外部磁場はない．電子のみを考え，イオンは電荷を中和するための一様なバックグラウンドとする．粒子に関する量は，位置 x，y と速度 v_x，v_y がある．場の量としては，電場の x，y 成分，電荷密度と静電ポテンシャルがある．粒子の位置と速度を 1 ステップ進めるためにリープ・フロッグ法が用いられている．電子の空間分布は一様で，速度分布はマクスウェルである．空間はメッシュ幅で，時間はプラズマ振動数の逆数で規格化されている．規格化されたデバイ長（規格化された電子の熱速度と同じ）は 1 であり，規格化された時間ステップ幅は 0.1 である．1000ステップの計算時間を比較する．計算はすべて倍精度で実行した．結果の一部は，本学会誌に掲載されたプロジェクトレビュー[9]の189-190ページに示されたものとほぼ同様であるが，本小特集の執筆に際して高速フーリエ変換（Fast Fourier Transform; FFT）の並列化を加え，再計算した結果を示す．またホストプロセッサでハイパースレッディングを利用した計算を追加した．

図 4 と**図 5** は，IFERC の Helios 増強システムの単一のホストプロセッサ（Xeon，8 コア）での 2 次元 PIC コードの並列化性能を示している．タイルサイズは，Xeon Phi の場合に最適になる $mx = 4$，$my = 4$ を用いた．**図 4** は，使用したスレッド数に対しての SOURCE，PUSH，SORT，FIELD の部分の計算時間を積み上げ棒グラフで示している．8 スレッドまでは物理コアに対する計算でありスレッド数の増加に対応して高速化されている．16スレッドの場合は 1 物理コアを 2 論理コアとみなすハイパースレッディングを用いているが，8 コアの場合と比較して計算時間の

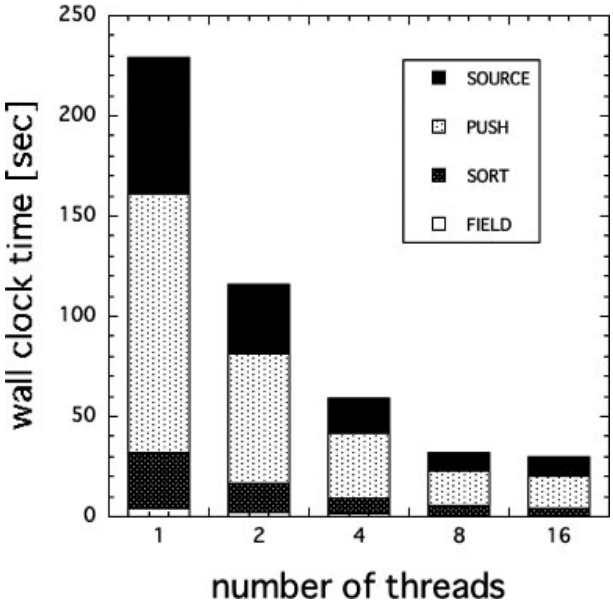

図 4　マルチコアに対する 2 次元 PIC コードの並列化性能.

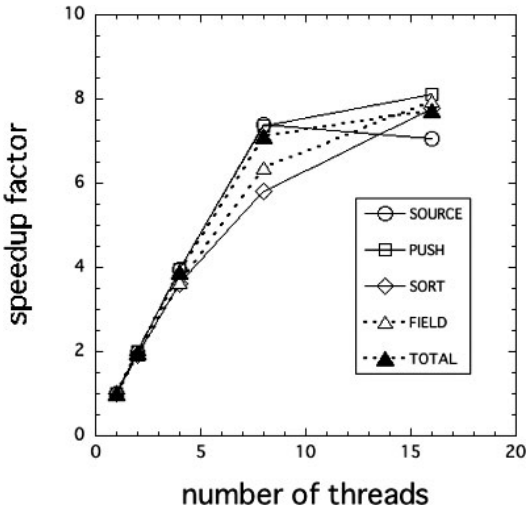

図 5　マルチコアに対する 2 次元 PIC コードの高速化因子.

短縮はわずか（約 8 パーセント）である．使用した各スレッドで，PUSH の計算時間の割合が最大で，SOURCE，SORT が続く．FIELD の計算時間の割合は無視できる程度である．**図 5** は，SOURCE，PUSH，SORT，FIELD に対して，それぞれスレッド数 1 の場合と比較した高速化因子を示している．8 コア（16コア）で比較すると SOURCE で 7.4(7.1) 倍，PUSH で 7.4(8.1) 倍になっているので並列化性能が高いことが実証されている．SORT と FIELD についても 5.8 (7.8) 倍，6.4(8.0) 倍になっている．全体としては 7.1(7.7) 倍の高速化が達成されている．SOURCE の場合のみハイパースレッディングを適用して少し遅くなっているのは気になるところである．

マルチコアで高い並列性能が証明されたので安心して Xeon Phi に適用できる．**図 6** と**図 7** は，Xeon Phi での並列化性能を示している．**図 6** は，使用したスレッド数に対する計算時間の積み上げ棒グラフを示している．スレッド数の増加に従って，順調に高速化されていることがわかる．

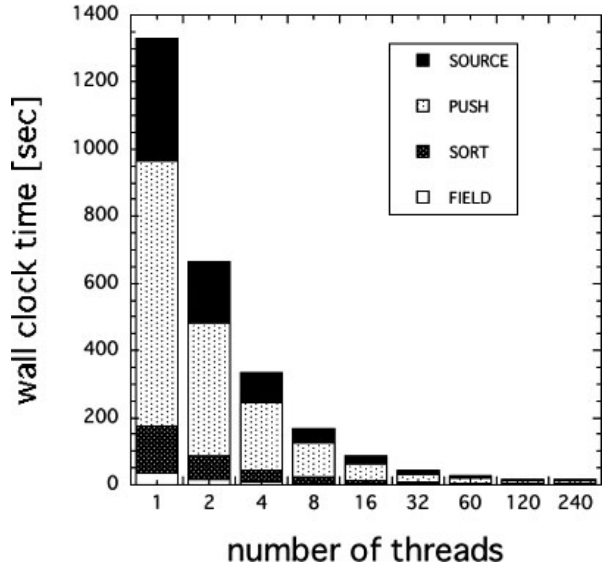

図6　メニーコアに対する2次元PICコードの並列化性能.

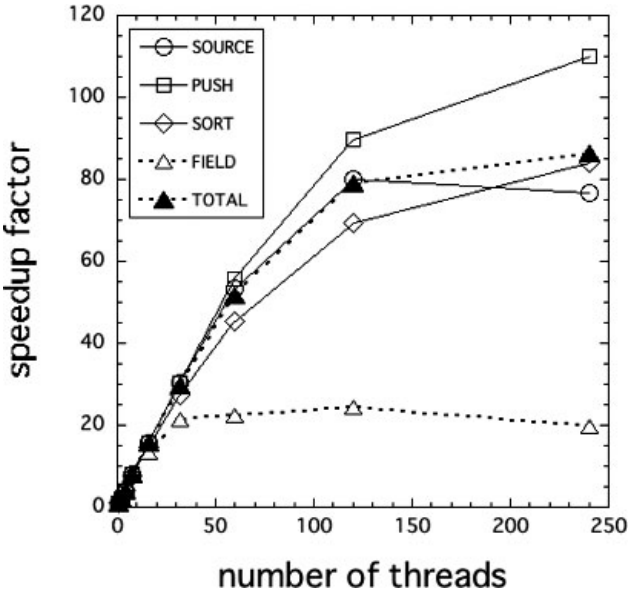

図7　メニーコアに対する2次元PICコードの高速化因子.

図7はスレッド数の増加に対する高速化因子を示している．60スレッドで比較すると，SOURCEで53.3倍，PUSHで55.8倍と高い並列化性能を示している．SORTでは45.3倍である．FIELDは22.4倍と並列化性能は少し劣るが，全体の計算時間に対する割合が小さいため問題ではない．また，FIELDではFFTの計算が支配的であり，32スレッド以上で高速化は飽和している．プログラム全体では51.8倍に高速化されている．

120スレッドと240スレッドの計算は，ハードウェア・スレッドが用いられているためか，マルチスレッドの場合と比較して顕著な高速化が測定されている．240スレッドで比較するとSOURCEが76.8倍，PUSHが110.1倍，SORTが84.0倍と高速化されている．FIELDは19.6倍となっているが，全体の計算時間に対する割合が小さいため問題ではない．全体では86.2倍の高速化を示している．

図8は，16スレッドのホストプロセッサと比較した

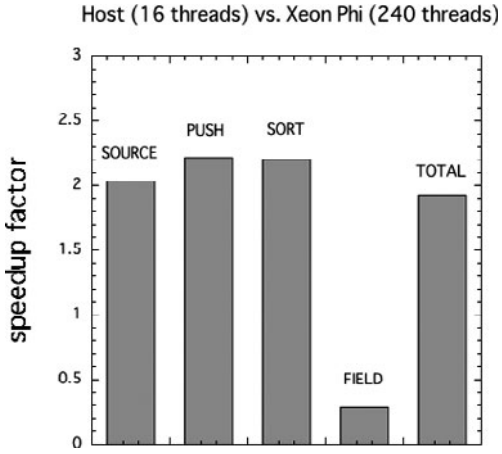

図8　メニーコア（240スレッド）とマルチコア（16スレッド）の2次元PICコードの実行時間の比較.

Xeon Phiの計算時間の高速化因子を示している．SOURCEで2.0倍，PUSHで2.2倍，SORTで2.2倍に高速化されている．FIELDは0.28倍と遅くなっているが全計算に対する比率が小さいため問題ではない．全体で1.9倍の高速化が達成されている．これは，1スレッドで比較するとXeon Phiの方が5.8倍遅いが，スレッド数の差とハードウェア・スレッドによって逆転して高い高速化が達成されていることになる．

3.2.4　まとめ

2次元静電近似PICコードを例に，Xeon Phiでの高速化手法について解説した．実行時間を比較すると，Xeon Phiを用いると，ホストプロセッサに対して約2倍の高速化が達成されることを実証した．筆者は同じ計算をIFERC-CSCの新しいGPU並列システムの構成要素であるGK210（TESLA K80に2個搭載されている）と核融合科学研究所の新しいプラズマシミュレータの構成要素であるSPARC64^{TM}XIfxで実行した．GK210は2496コアを持ち最大理論性能は0.935 TFLOPSである．SPARC64 XIfxはマルチコアであり，32コアに加えて2個の補助コアを持ち，0.908 TFLOPSである．Xeon Phi 5110Pの理論性能は1.011 TFLOPSである．Xeon Phi 5110PとGK210はアクセラレータであり，SPARC64 XIfxはホストプロセッサである違いがあるが，理論性能はほとんど同じである．またSPARC64 XIfxはOpenMPで書かれたプログラムがそのまま利用できる．結果として，SOURCE，PUSH，SORT，FIELDの計算時間の割合はそれぞれ異なった傾向を示したが，全体の実行時間はほとんど同一であった．メニーコアに対してマルチコアのコア数も増加傾向にあるため，次世代で差がどのように変化するか興味があるところである．またGPUについてはプログラムの難しさはあるが，すでにプログラムを開発済みのプログラマにとっては問題ない．それぞれどのような進化をするのか楽しみである．

3.3 マルチ MIC＋マルチ CPU で構成されたヘテロジニアス計算機を使った MHD 乱流の直接数値計算

3.3.1 はじめに

近年のスーパーコンピュータは GPU や MIC のようなアクセラレータが使われているものが多数を占めている．これらは MPI を用いた従来の並列技術に加え，各コアでスレッド並列を行っている．オフロードモードではGPUを用いた並列計算をアクセラレータ（コプロセッサ）に処理をさせることが必要となる．しかし，MIC と呼ばれる Intel Xeon Phiコプロセッサは GPUとは異なり，コードを書き換えることなく実行できる．このモードでは OS が CPU 側のホストプロセスで実行されるため，コプロセッサ側で処理する必要がない．したがって，MPI プロセスを直接コプロセッサで実行することができるようになる．その場合，コプロセッサに MPI のランク[i]が設定でき，ピア・ツー・ピア通信を行うため，独立した計算ノードのように働く．したがって，プロセッサとコプロセッサで構成されるヘテロジニアスシステムが可能となる．つまり，MPI+OpenMP で記述されたプログラムコードを変更することなく，コプロセッサの有無にかかわらず MPI アプリケーションを使用が可能である．本節で紹介する乱流直接数値シミュレーションのプログラミングコードはMPI＋OpenMPによる並列化をしている．その際の効率的なヘテロジニアスシステムのためのコード設計，例えばサブミッションシェルにおけるノード数，CPU 数，およびMIC 数の組み合わせと計算速度について報告する．

3.3.2 DNS コード

すでに我々は，CPU+MIC を用いて円管内乱流 DNS コードの並列化に成功している[10-12]．本節では，Satake *et al.*[13]により開発されたチャネル内乱流 DNS コードを改良したものを用いて，Helios 計算機を使った速度向上についてベンチマークを行った．まず用いたチャネル内乱流 DNS コードの計算手法について説明する．3 次元乱流の運動を規定する支配方程式は，直交座標系での Navier-Stokes 方程式，熱伝導方程式，連続の式である．壁面には non-slip 境界条件を，流れ方向とスパン方向には周期境界条件を課した．空間の離散化には，2 方向にスペクトル法を用いて壁垂直方向に差分法を用いるいわゆるハイブリッド法である．計算格子は，壁垂直方向速度のみにスタガード格子を用いる．粘性項の時間積分には，陰解法である Crank-Nicolson 法を用い，他の項には陽解法である三次精度 modified Runge-Kutta 法を用いる．

3.3.2.1 並列化の実装

粘性項のHelmholtzの式と圧力Poisson方程式は，フーリエスペースの三重対角行列解法を用いて解いている．**図 9** で示すように並列アルゴリズムとして *x-y* 面内でFFTをかけ，壁垂直方向に三重対角行列解法による解法を必要とする．Poisson 方程式は**図 9** のような転置を必要とし，メモリパーティションが異なる 2 つのメモリ空間を用いる．毎ステップで対流項計算時（パディング法）に転置が必要となる．等間隔計算格子と周期的境界条件は主流方向で使われるので，フーリエ変換を用いることによって 3 次元 Poisson 方程式を 1 次元の式として扱うことができる．行列の計算において，三重対角行列解法は，波数ごとに使用している．

3.3.2.2 プログラミングの方針

MPI+OpenMPを適用したFORTRANで書かれたプログラムコードをネイティブモード（MIC）とヘテロジニアス計算（CPU+MIC）で使う．本プログラムコードは FORTRAN90で書かれており，MPI ライブラリを参照している． 3 次元配列の計算に対し，外側の DO ループを MPI

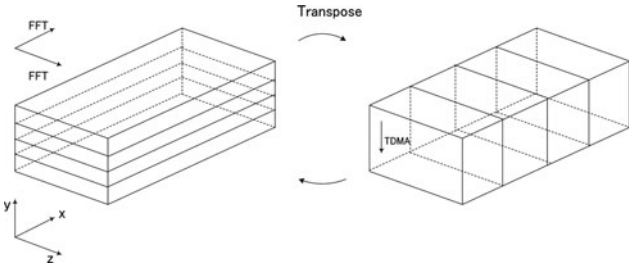

図 9　計算領域に対するメモリーの分割方法と転送．

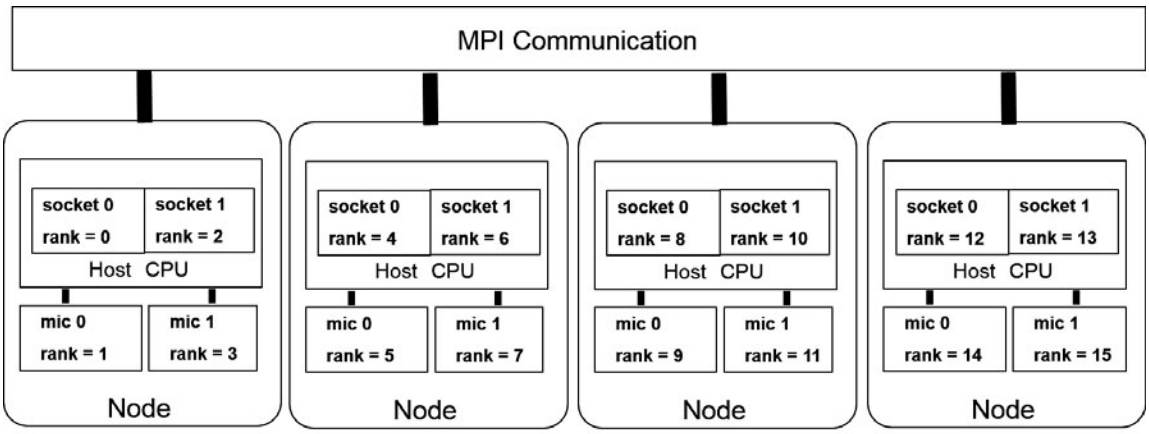

図10　シンメトリックモードにおけるランクの構成．

i　ランクとは，MPI でのプログラムの実行単位であるプロセスの識別番号で，0 から順に連続した整数が割り当てられる．

による並列化を行い，その内側の2つのループに関してOpenMPによる並列化を行った．このルールはFFTや軸交換ルーチン等，全てに適用している．

3.3.2.3 Helios計算機のヘテロジニアスクラスタリングシステム

シンメトリックモードは，**図10**のようなヘテロジニアス計算である．

このモードではアプリケーションプロセスがホストCPUとXeon Phiコプロセッサの両方で実行される．それらは通常，MPIを介して通信する．この実行環境はヘテロジニアスクラスタ環境において同一クラスタ内でも別ノードとして扱うことができる．スレッド定義はKMP_AFFINITYによって指定する．スレッドは0から239の範囲でKMP_AFFINITYによって書くことができる．しかし，0，237，238，および239のスレッドはOSで使用しているため，通常は使用することができない．よってMICの最大スレッド数は236スレッドとなる．

3.3.2.4 ヘテロジニアス計算を行うための実行シェル

ホストCPUとMICを同時に使うヘテロジニアス計算を行うには使用するノードとCPU及びMICのプロセッサをシェルで指定する必要がある．1ノード内で2CPU，2MICを稼働させ，1つのプロセッサ内で2つランクを立てた時の例を**図11**に示す．また，ここでは主要な記述以外は省略している．

まずNODELISTで使用するノードを決定する．この記述では16個のノードを使用するため，-I0-15と記述している．次にforループで使用するノードとCPU及びMICをMachinefileに書き込む．CPUを使う場合はノードの番号のみ書き込み，MICを使用する場合はノード番号のあとに-mic0もしくは-mic1と書く．今回のMachinefileへの書き込

```
NODELIST=$(nodeset -e -I0-15 -S " " $SLURM_JOB_NODELIST)
for h in $NODELIST; do
  echo "$h" >> $MACHINEFILE
done
for h in $NODELIST; do
  echo "$h-mic0" >> $MACHINEFILE
done
for h in $NODELIST; do
  echo "$h" >> $MACHINEFILE
done
for h in $NODELIST; do
  echo "$h-mic1" >> $MACHINEFILE
done
for h in $NODELIST; do
  echo "$h" >> $MACHINEFILE
done
for h in $NODELIST; do
  echo "$h-mic0" >> $MACHINEFILE
done
for h in $NODELIST; do
  echo "$h" >> $MACHINEFILE
done
for h in $NODELIST; do
  echo "$h-mic1" >> $MACHINEFILE
done

clush -w $SLURM_JOB_NODELIST mkdir /mic_tmp/pipe.${SLURM_JOBID}
clush -w $SLURM_JOB_NODELIST ls -ld /mic_tmp/pipe.${SLURM_JOBID} | dshbak -c
clush -w $SLURM_JOB_NODELIST --copy --dest /mic_tmp $DIR/pipe.mic $DIR/pipe.out $DIR/pipe.hyb2.sh
clush -w $SLURM_JOB_NODELIST ls -l /mic_tmp/pipe.mic | dshbak -c
cd /mic_tmp/pipe.${SLURM_JOBID}

time mpiexec.hydra -binding none -np 128 -machinefile $MACHINEFILE /mic_tmp/pipe.hyb2.sh
```

図11 実行シェル（hybrid2.sh）．

みは for ループで16回書くことを 4 パターン，2 回繰り返し行っている．つまり使用するノードが16個で，1 つのノード内で CPU を 4 個，MIC を 4 個使うことになる．よって，ランクが128個設定される．なお，ランクは Machinefile に書かれた順番通りに振られていく．次に CPU 用の実行ファイル pipe.out と MIC 用の実行ファイル pipe.mic，実行用のシェルの pipe.hyb2.sh を MIC 用のディレクトリである mic_tmp にコピーする必要がある．そのため，clush というコマンドを使いコピー先のノードを指定し，各ノードに mic_tmp を作り，その中に上記のファイルをコピーする．ジョブ実行コマンドは mpiexec.hydra である．実行ファイルは直接実行するのではなく，別のシェルで実行する必要がある．この時のシェルの記述内容を**図12**に示す．また，ここでは主要な記述以外は省略している．

このシェルは Machinefile の記述によってどのプロセッサでどのファイルを実行するか判定している．Machinefile に mic と書いているかの判定はしておらず，ランクの番号のみを if 文で判定している．ランクが16より小さい場合は，KMP_AFFINITY でスレッド 0 から 3 を指定し，mic_tmp 内の pipe.out をホスト CPU で実行する．ランクが32より小さい場合は，スレッドを KMP_AFFINITY で指定し，mic_tmp 内の pipe.mic を MIC で実行する．ランクが48より小さい場合は，KMP_AFFINITY でスレッド 4 から 7 を指定し，mic_tmp 内の pipe.out をホスト CPU で実行する．ランクが64より小さい場合は，スレッドを KMP_AFFINITY で指定し，mic_tmp 内の pipe.mic を MIC で実行する．同様にして，128ランクまで指定する．なお，1 つのプロセッサで 2 つのランクを立てるときには使用するスレッドが重複しないように注意しなければいけない．さらに MIC の場合は OS で使用するスレッドも使ってはいけない．今回の場合は mic0，mic1 共に 0，119，120，239が OS で使用されている．

3. 3. 3　結果と考察

表 1 にある 2 種類のレイノルズ数を使い，各レイノルズ

```
if [ $PMI_RANK -lt 16 ]; then
  export OMP_NUM_THREADS=4
  export KMP_AFFINITY="explicit,granularity=fine,proclist=[0-3]"
  /mic_tmp/pipe.out
elif [ $PMI_RANK -lt 32 ]; then
  export OMP_NUM_THREADS=60
  export
KMP_AFFINITY="explicit,granularity=fine,proclist=[1,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,4
6,48,50,52,54,56,58,60,62,64,66,68,70,72,74,76,78,80,82,84,86,88,90,92,94,96,98,100,102,104,106,108,110,112,114,116,1
18]"
  /mic_tmp/pipe.mic
elif [ $PMI_RANK -lt 48 ]; then
  export OMP_NUM_THREADS=4
  export KMP_AFFINITY="explicit,granularity=fine,proclist=[4-7]"
  /mic_tmp/pipe.out
elif [ $PMI_RANK -lt 64 ]; then
  export OMP_NUM_THREADS=60
  export
KMP_AFFINITY="explicit,granularity=fine,proclist=[1,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,4
6,48,50,52,54,56,58,60,62,64,66,68,70,72,74,76,78,80,82,84,86,88,90,92,94,96,98,100,102,104,106,108,110,112,114,116,1
18]"
  /mic_tmp/pipe.mic
elif [ $PMI_RANK -lt 80 ]; then
  export OMP_NUM_THREADS=4
  export KMP_AFFINITY="explicit,granularity=fine,proclist=[8-11]"
  /mic_tmp/pipe.out
elif [ $PMI_RANK -lt 96 ]; then
  export OMP_NUM_THREADS=60
  export
KMP_AFFINITY="explicit,granularity=fine,proclist=[121,122,124,126,128,130,132,134,136,138,140,142,144,146,148,1
50,152,154,156,158,160,162,164,166,168,170,172,174,176,178,180,182,184,186,188,190,192,194,196,198,200,202,204,2
06,208,210,212,214,216,218,220,222,224,226,228,230,232,234,236,238]"
  /mic_tmp/pipe.mic
elif [ $PMI_RANK -lt 112 ]; then
  export OMP_NUM_THREADS=4
  export KMP_AFFINITY="explicit,granularity=fine,proclist=[12-15]"
  /mic_tmp/pipe.out
else
  export OMP_NUM_THREADS=60
  export
KMP_AFFINITY="explicit,granularity=fine,proclist=[121,122,124,126,128,130,132,134,136,138,140,142,144,146,148,1
50,152,154,156,158,160,162,164,166,168,170,172,174,176,178,180,182,184,186,188,190,192,194,196,198,200,202,204,2
06,208,210,212,214,216,218,220,222,224,226,228,230,232,234,236,238]"
  /mic_tmp/pipe.mic
fi
```

図12　実行シェル 2 （pipe.hyb2.sh）．

数に応じたメッシュ解像度でそれぞれのベンチマークを行った．

これらのデータは Satake *et al.*[13]に基づいている．プログラムのベンチマークは，経過時間[sec]を倍精度実数で返す関数である mpi_wtime( )を用いて計測した．

3.3.3.1 ヘテロジニアス計算（ホスト CPU とマルチの MIC）の評価

比較を行うためにノード数を同じにした CPU と MIC の結果も示した．**図13**は，128×128×128での CPU+MIC のノード数に対する速度向上率を示している．**図14**は，1024×1024×768での CPU+MIC のノード数に対する速度向上率を示している．CPU のみで行った計算が最速で，次は CPU+MIC，最も遅い結果が MIC のみとなった．

図15にランクの数を固定した時，128×128×128においてホスト CPU，MIC，CPU+MIC を様々な組み合わせで稼働させた時の速度比較を示す．**表 2** は**図15**における 1 つの CPU，1 つの MIC あたりのランク数の組み合わせを示している．

ランクの数の合計はどのパターンにおいても32となる．全てのケースにおいて MIC 内のスレッド数は120に固定した．4 ノードでは CPU+MIC がホスト CPU の速度を超えた．**図16**に1024×1024×768において合計ランク数を128に固定した時，ホスト CPU，MIC，CPU+MIC を様々な組み合わせで稼働させた時の速度比較を示す．**表 3** は**図16**における 1 つの CPU，1 つの MIC あたりのランク数の組み合わせを示している．

ランクの数の合計はどのパターンにおいても128となる．全てのケースにおいて MIC 内のスレッド数は120に固定した．32ノードでは CPU+MIC がホスト CPU の速度を超え

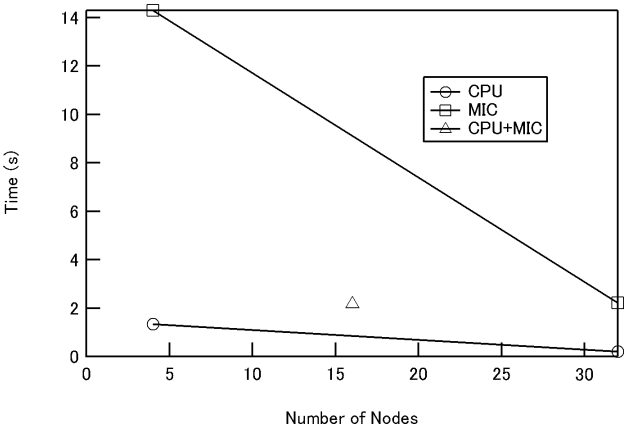

図13 ノード数変化による計算時間（128×128×128）．

表 1 レイノルズ数とメッシュ数．

Reynolds number	number of mesh points (x y z)
150	128x128x128
1100	1024x1024x768

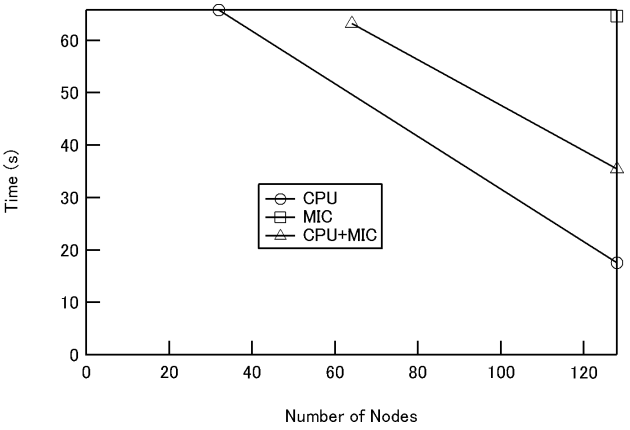

図14 ノード数変化による計算時間（1024×1024×768）．

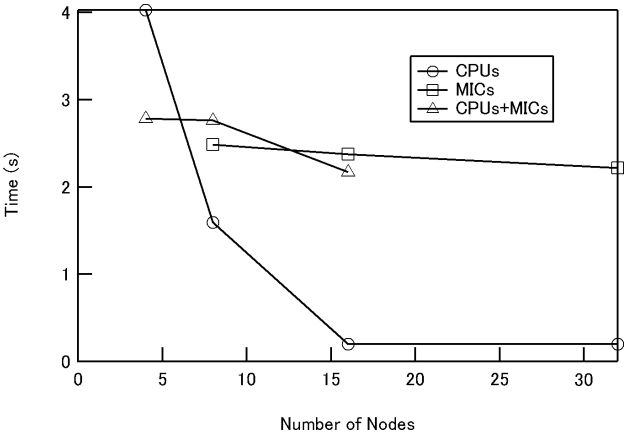

図15 32ランクの場合のノード数変化による計算時間（128×128×128）．

表 2 128×128×128 における CPU と MIC の組み合わせ．

	CPUs				MICs			CPUs+MICs		
Nodes	32	16	8	4	32	16	8	16	8	4
data/CPUs	1	2	4	8	0	0	0	1	2	4
data/MIC0	0	0	0	0	1	1	2	1	1	2
data/MIC1	0	0	0	0	0	1	2	0	1	2

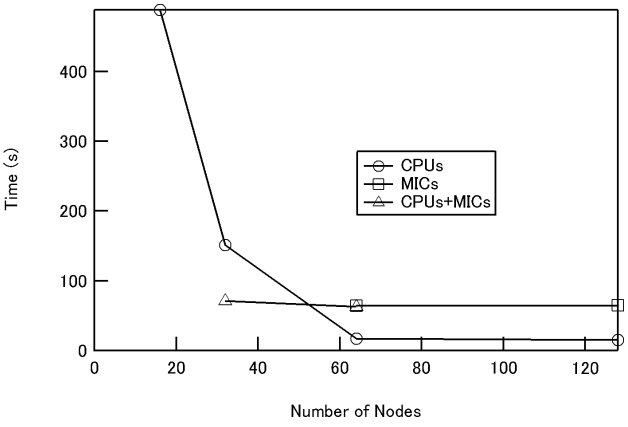

図16 128ランクの場合のノード数変化による計算時間（1024×1024×768）．

表 3 1024×1024×768（128 ranks）における CPU と MIC の組み合わせ．

	CPUs				MICs		CPUs+MICs	
Nodes	128	64	32	16	128	64	64	32
data/CPUs	1	2	4	8	0	0	1	2
data/MIC0	0	0	0	0	1	1	1	1
data/MIC1	0	0	0	0	0	1	0	1

た．合計128ランクの1024×1024×768において32ノードではMICのみで実行することができなかった．また，16ノードではCPU+MICで実行することができなかった．これらの原因はMICのメモリ容量不足であり，MIC内に2ランクたてることができなかった．そのため，分割数を128から256に変えた．

図17に1024×1024×768において合計ランク数を256に固定した時，ホストCPU，MIC，CPU+MICを様々な組み合わせで稼働させた時の速度比較を示す．**表4**は**図17**における1つのCPU，1つのMICあたりのランク数の組み合わせを示している．

ランクの数の合計はどのパターンにおいても256となる．全てのケースにおいてMIC内のスレッド数は120に固定した．32ノードと64ノードではCPU+MICがホストCPUの速度を超えた．しかし，128ノードを使った時はホストCPUが最速となった．

3.3.4 結論

速度比較の評価はホストCPUを使ったものや，MICのネイティブモード，シンメトリックモード（ヘテロジニアス計算）といった複数の計算方法で行った．Helios計算機を使用した場合の計算時間は以下の様な結論が得られた．

1．コードがMPI+OpenMPで書かれているとき，ホストCPUで並列計算のスレッド数を増加させることで，メッシュの数に比例する速度向上を得ることができる．

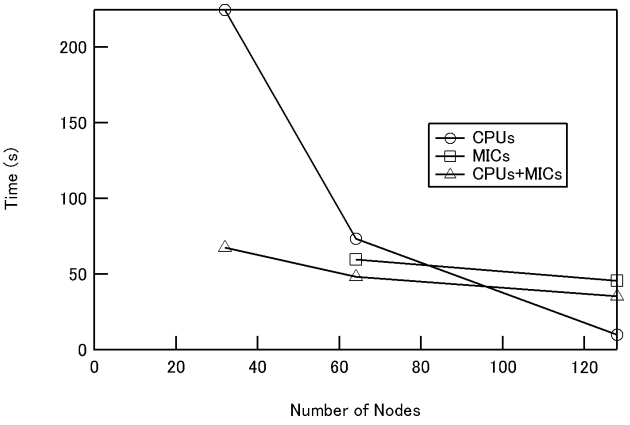

図17 256ランクの場合のノード数変化による計算時間（1024×1024×768）．

表4 1024×1024×768（256 ranks）におけるCPUとMICの組み合わせ．

	CPUs			MICs		CPUs+MICs		
Nodes	128	64	32	128	64	128	64	32
data/CPUs	2	4	8	0	0	1	2	4
data/MIC 0	0	0	0	1	2	1	1	2
data/MIC 1	0	0	0	1	2	0	1	2

2．マルチMICによるネイティブモードの計算では120スレッドを使った計算が最速となった．

3．MICはCPU+MICの組み合わせであるヘテロジニアス計算においてMPIでランクを作ることができるので，演算コアとして使用することができる．つまり，コア内でランクを振ることができるため，全体の並列化効率を向上させることができる．

4．ホストCPU，ネイティブモード，およびこれらのハイブリッドの速度を比較すると，ホストCPUが最速となった．そして次は，シンメトリックモード，MICネイティブモードであった．1024×1024×768においてノードの数が32，64の時シンメトリックモードとホストCPUが同じ数のランクを使用する場合はシンメトリックモードのほうがホストCPUより速くなる．

ヘテロジニアススーパーコンピュータでシンメトリックモードが使える環境であるなら，上記のように少ないノード数で高性能な計算を行うことができるということが明らかになった．

参考文献

［1］中島徳嘉 他：プラズマ・核融合学会誌 **91**, 711 (2015).
［2］http://www.top500.org/lists/2015/11/.
［3］ジム・ジェファース，ジェームス・レインダース（訳：すがわらきよふみ・エクセルソフト株式会社），インテル®Xeon Phi™ コプロセッサ ハイパフォーマンス・プログラミング（株式会社カットシステム，日本，2014）．
［4］C.K. Birdsall and A.B. Langdon, *Plasma Physics via Computer Simulation* (Institule of Physics Publishing, Bristol and Philadelphia, 1991).
［5］内藤裕志：プラズマ・核融合学会誌 **74**, 470 (1998).
［6］内藤裕志，佐竹真介：プラズマ・核融合学会誌 **89**, 245 (2013).
［7］V.K. Decyk and T.V. Singh, Comput. Phys. Commun. **182**, 641 (2011).
［8］V.K. Decyk and T.V. Singh, Comput. Phys. Commun. **185**, 708 (2014).
［9］日本原子力研究開発機構 IFERC-CSC 利用委員会他編集：プラズマ・核融合学会誌 **92**, 157 (2016).
［10］佐竹信一 他：”マルチMIC＋マルチCPUを使ったヘテロジニアス計算を用いたMPI+OpenMPによる円管内乱流の直接数値計算”，第29回数値流体力学シンポジウム講演論文集 (2015)12.15-17，九州大学，B08-1.
［11］S. Satake *et al.,* Fusion Sci. Tech. **68**, 640 (2015). dx.doi.org/10.13182/FST14-956
［12］K. Hosaka *et al.,* "DNS of a turbulent flow by MPI+OpenMP for heterogeneous computing using multi-CPU and multi-MIC", Proc. 28th International Conference on Parallel Computational Fluid Dynamics, Parallel CFD 2016 (2016).
［13］S. Satake *et al.,* Phys. Fluids **18**, 125106 (2006).

中島徳嘉（なかじまのりよし）

核融合科学研究所 ヘリカル研究部 六ヶ所研究センター 教授．今年5月から六ヶ所赴任．主としてMHD・新古典理論の研究に従事してきたが，IFERC事業長兼任もほぼ6年となり，六ヶ所生活共々，得難い経験と人間関係を得た．事業の2019年末までの延長も決まり，新たな展開に期待している．

内藤裕志（ないとうひろし）

山口大学大学院創成科学研究科教授．学部担当は知能情報工学科．研究分野はプラズマのコンピュータシミュレーション．趣味はハングルの勉強．本年度末で無事定年退職の予定．定年後は明るい下流老人として最高の人生を送りたいと思っています．

保坂和樹（ほさかかずき）

東京理科大学大学院基礎工学研究科電子応用工学専攻．専門分野／関心分野：シミュレーション工学／特にヘテロジニアス数値計算．

宮戸直亮（みやとなおあき）

量子科学技術研究開発機構・六ヶ所核融合研究所 上席研究員・BA計画調整グループサブリーダー．日本原子力研究所，日本原子力研究開発機構を経て2016年4月から現職．と書くと研究機関を渡り歩いているように見えるが，実質の所属は変わらず，所属機関の名前が十数年で2度変わっている．1回ならともかく，2回も変わるとは……こうなると3回目もあるかも．専門は磁場閉じ込めプラズマの理論・シミュレーションで，最近はジャイロ運動論モデルの拡張などプラズマの基礎モデルの研究が中心．

佐竹信一（さたけしんいち）

東京理科大学基礎工学部電子応用工学科 教授．専門分野／関心分野：数値熱流体力学／特に核融合炉冷却に関するMHD乱流熱伝達．

功刀資彰（くぬぎともあき）

京都大学 物理工学系（大学院工学研究科）教授．専門分野／関心分野：伝熱工学，特に沸騰現象の実験と解析／混相流工学／数値熱流体力学／核融合炉工学（ブランケット，ダイバータ，安全性など）．