2010年度
卒業論文

スカラー並列型スーパーコンピュータ向け
磁気流体コードの最適化

神戸大学工学部情報知能工学科

山浦優気

指導教員　陰山聡

2011年2月23日

スカラー並列型スーパーコンピュータ向け磁気流体コードの最適化

山浦優気

要旨

ベクトル並列型スーパーコンピュータ向けに開発された磁気流体力学シミュレーションコードを，スカラー並列型スーパーコンピュータ向けに最適化するための様々な手法とその効果について研究した．特にレジスタ溢れを防ぐレジスタ最適化，ストライドを正常にする配列添字の入れ替え，キャッシュヒット率を上げるループタイリングについて調べた．シミュレーションコードの中で計算負荷が最も集中する場所を特定した上で，問題サイズを L1 キャッシュに乗るサイズ，L2 キャッシュに乗るサイズ，L2 キャッシュに乗らないサイズの 3 つに設定し，性能向上の検証を行った．レジスタ最適化手法によって，すべての問題サイズで性能が向上した．配列の添字の入れ替え手法では，L2 キャッシュに乗らない問題サイズについて性能が向上した．ループタイリングでは，どの問題サイズに対しても性能を向上させることができなかった．

目 次

1　序論

現在，国家プロジェクトとしての「最先端・高性能汎用スーパーコンピュータの開発利用」[1] が文部科学省を中心に進められている．この中の「世界最先端・最高性能の汎用京速計算機システムの開発・整備」により，次世代スーパーコンピュータ「京」の開発が 2012 年の運用開始を目指して進められている.「京」は計算性能 10 ペタフロップスの達成を目標としており．これは 2010 年 11 月時点 [3] で，世界一のスーパーコンピュータである「Tianhe-1A」[2] の 4.5 ペタフロップスと比べても 2 倍以上高速である．

海洋研究開発機構の「地球シミュレータ」に次ぐ国家プロジェクトである「京」は，スカラー並列型スーパーコンピュータであり，ベクトル並列型の「地球シミュレータ」と全く異なるアーキテクチャである．スカラー並列型とベクトル並列型の違いについては 2.1 章で詳しく説明する．

ベクトル並列型スーパーコンピュータからスカラー並列型スーパーコンピュータへの移行は世界的潮流である．スーパーコンピュータのハードウェアアーキテクチャの変化にともない当然それに応じたアプリケーションへの変更が求められる．スーパーコンピュータのアプリケーションは大規模な計算機シミュレーションが多い．その対象は科学技術のあらゆる分野に広がっている．

これらのシミュレーションプログラムの開発には長い年月を費やすのが普通である．スーパーコンピュータの持つ演算能力を十分に活用するために，その実装方法や場合によっては基本アルゴリズムまでもハードウェアアーキテクチャに応じて変更する必要があるからである．

シミュレーションの研究者にとって地球シミュレータのようなベクトル並列型スーパーコンピュータから,「京」のようなスカラー並列型スーパーコンピュータへの移行は大きな挑戦である．この移行のための技術的な知見の蓄積が未だ不十分であるためである．

本研究の目的は，地球シミュレータで実績のあるシミュレーションコードを題材にとり，これをスカラー並列型スーパーコンピュータ向けに移植し，複数の最適化手法を施すことで，その効果を調べることである．

2　スーパーコンピュータと数値計算

2.1　ベクトル計算機とスカラー計算機

ベクトル計算機は高性能なベクトルプロセッサ [4] と高いメモリバンド幅を使用した計算機である．代表的なベクトル計算機は地球シミュレータである．ベクトルプロセッサは，各レジスタが多数の要素を持つアレイレジスタを用いる．そして，アレイレジスタをオペランドとして，1 つの命令でアレイレジスタの全要素に対して演算を行うプロセッサである．ベクトルプロセッサと高速なメモリを持つベクトル計算機は，メモリを逐一参照するプログラムを高速に処理できる特徴を持つ．このためベクトル計算機向けの最適化としては主として最内側ループ長を長くするのが有効である [8].

スカラー計算機はベクトルプロセッサに比べ安価で低消費電力なスカラープロセッサを使用した計算機である．次世代スーパコンピュータ京や HA8000 クラスタシステム (T2K 東大) はスカラー計算機である．スカラープロセッサはベクトルプロセッサのようにアレイレジスタを搭載していない極めて単純なプロセッサである．近年のプロセッサの目覚ましい進歩によって，メモリのデータ転送速度に対して，プロセッサの処理能力は劇的に速くなっている．そのため，多くのスカラープロセッサはキャッシュ(3.3 章) を導入している．主記憶とレジスタの間にあるキャッシュは，主記憶に比べ高速でレジスタよりも多くのデータを格納することができる．

このためスカラー計算機向けの最適化としては，キャッシュに格納されているデータを何度も参照することが有効である [9].

2.2　HA8000 クラスタシステム

本研究で用いたスーパーコンピュータはスカラー並列型スカラー計算機 HA8000 クラスタシステム (T2K 東大) である．Table 2.1 に HA8000 クラスタシステムの仕様 [5][6] をまとめる．1 コアのクロック周波数は 2.3GHz であるが，SIMD(Single Instruction Multiple Data)[7] により 1 クロックで 4 回の浮動小数点演算を実現している．つまり，1 コアの理論演算性能は 9.2GFlops である:

$$4 \times 2.3[GHz] = 9.2[GFlops] \tag{2.1}$$

1 コアの最大性能である 9.2GFlops に対して，L2 キャッシュメモリの転送速度は 2300MHz と遅い．従って，メモリに頻繁にアクセスする必要のあるプログラムの場合，実行性能が低下する．

Table 2.1: Specifications of the HA8000 Cluster System(T2K)

Processor	AMD Opteron 8356
CPU Speed	2.3GHz
Number of cores	4core / processor
L1 cache size	64KB / core
L2 cache size	512KB / core
L3 cache size	2MB / processor
L1 cache speed	2300MHz
L2 cache speed	2300MHz
Number of processor in a node	4 / node
Peak performance of a core	9.2GFlops / core

2.3 磁気流体コード

本研究で最適化を施す磁気流体コードについて簡単にまとめる．

このシミュレーションの最大の目的は，地球内部のコア領域の流体鉄の流れと，それによる磁場の成長 と 維持の物理機構を解明する事である [10]．流体鉄は地球内部の外核と呼ばれる層にある．この外核の液体鉄が対流運動するために，その運動エネルギーが MHD (Magnetohydrodynamics[11]) ダイナモ作用によって磁場のエネルギーに変換されている．シミュレーションモデルとして，地球の外核を想定し，二つの同心球面に挟まれた球殻状の領域を考える．その中に電気伝導性流体（MHD 流体）が入っている．内側の球面（半径 $r = r_i$）は高温，外側の球面（半径 $r = r_o$）は低温に保たれている．球の中心方向に重力がはたらき，二つの球殻は同じ角速度 Ω で回転する．温度差が十分に大きければ（レイリー数 Ra が十分高ければ）内部の流体は熱対流運動し，MHD ダイナモ機構によって，対流の運動エネルギーが磁場のエネルギーに変換され，磁場が生成される [12]．

流体鉄の流れと磁場の変化は以下の MHD 方程式にしたがう．この MHD 方程式をスーパーコンピュータを用いて可能な限り高速・高解像度で解く事で地球磁場の起源に迫る．

$$\frac{\partial \rho}{\partial t} = -\nabla \cdot f, \tag{2.2}$$

$$\frac{\partial f}{\partial t} = -\nabla \cdot (vf) - \nabla p + \mathbf{j} \times \mathbf{B} + \rho \mathbf{g} + 2\rho \mathbf{v} \times \Omega + \mu(\nabla^2 \mathbf{v} + \frac{1}{3}\nabla(\nabla \cdot \mathbf{v})), \tag{2.3}$$

$$\frac{\partial p}{\partial t} = -\mathbf{v} \cdot \nabla p \ \gamma p \nabla \cdot \mathbf{v} + (\gamma - 1) K \nabla^2 T + (\gamma - 1)\eta \mathbf{j}^2 + (\gamma - 1)\Phi, \tag{2.4}$$

$$\frac{\partial \mathbf{A}}{\partial t} = -\mathbf{E}, \tag{2.5}$$

ここで

$$\begin{aligned} p &= \rho T, \\ \mathbf{B} &= \nabla \times \mathbf{A}, \\ \mathbf{j} &= \nabla \times \mathbf{B}, \\ \mathbf{E} &= -\mathbf{v} \times \mathbf{B} + \eta \mathbf{j}, \\ \mathbf{g} &= -\frac{g_0}{r^2 \hat{r}}, \\ \Phi &= 2\mu(e_{ij}e_{ij} - \frac{1}{3}(\nabla \cdot \mathbf{v})^2), \\ e_{ij} &= \frac{1}{2}(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i}). \end{aligned} \tag{2.6}$$

ここで質量密度 ρ，圧力 p，質量フラックス密度$\boldsymbol{f}$，磁場のベクトルポテンシャル$\boldsymbol{A}$が基本変数である．磁場$\boldsymbol{B}$，電束密度$\boldsymbol{j}$と電場$\boldsymbol{E}$は補助的な場として扱われる．また比熱比 γ，粘性率 μ，熱伝導率 K と電気抵抗 η は定数とする．ベクトル$\boldsymbol{g}$は重力加速度，$\hat{r}$ は動径方向の単位ベクトルである．

本研究で最適化を行う磁気流体コードでは，この MHD 方程式を空間方向には 2 次精度の有限差分法で離散化し，時間方向には 4 次精度の Runge-Kutta 法で積分する．空間の離散格子には Yin-Yang 格子を用いる [13]．並列化のために Yin-Yang 格子の水平面方向で 2 次元領域分割し，MPI (Message Passing Interface) を用いて通信を行う．

このコードは地球シミュレータの全ノードを用いた大規模並列計算を行う目的で陰山教授によって開発・最適化されたものである [14]．地球シミュレータの 4096 個のベクトルプロセッサを用いた大規模並列計算により 15.2TFlops(理論ピーク性能の 46%) の演算速度を出し，これによって 2004 年のゴードン・ベル賞を受賞した実績のあるコードである [15]．

Table 3.1: Memory and Size

Technology	I/O Speed	Size	Price per bit
SRAM	High	Low	High
DRAM	Medium	Medium	Medium
Disk	Low	Large	Low

3　メモリアクセスと実行性能

3.1　メモリ

本章では，スカラー並列型スーパーコンピュータ向けに磁気流体コードを最適化するのに必要とするメモリ関係のハードウェアについて簡単に紹介する．計算機で用いられるメモリは大きく分けて 3 種類 [16] ある．

- SRAM
- DRAM
- 磁気ディスクなどの外部記憶装置

CPU に近いキャッシュ部分では SRAM が用いられ，主記憶には DRAM が使用される．メモリと実際に実装される容量の関係は Table 3.1 に示す．実装上重要となる，記憶装置のビットあたりの単価も示した．

近年のプロセッサ性能の飛躍的向上によって，プロセッサの演算速度はメモリの参照速度に比べて劇的に速くなっている．そのため，プログラムの実行においてメモリを参照している間，手続きが先に進まない状態が頻繁に現れるようになってきた．手続きが先に進まない状態をメモリ待ちと呼び，この時プロセッサは遊んでいる状態となる．これを避けるため，上記の 3 つの記憶装置を複数の段階にわけて，参照される可能性が高いデータをより高速なメモリに保存する．プログラムをより高速に動作させるには，高速なメモリに保存されているデータを再利用することによって，メモリ待ち時間をいかに少なくするかが重要となる．

3.2　レジスタ

レジスタとは，値を保存しておく一種のメモリである．保存可能な容量は少なく，多くの 64bit のシステムの場合，1 つのレジスタの容量は 64bit や 128bit である．プロセッサコアの中には，レジスタがいくつか集まっている．またレジスタの

中にはスタック領域に使用されるスタックレジスタやプログラムカウンタを保存するプログラムレジスタなど特定の目的に使用されるものもある．そのため，実際のプログラム中で使用する変数を保存できる汎用レジスタは多くない．

多くのアーキテクチャでは，演算ユニットである ALU から読み書き可能なメモリはレジスタだけであるため，主記憶の値を変更するときは，逐一読み出す必要がある．例えば，Program 1 の場合アセンブリ言語に変更すると Program 2 のようになる．

Program 1:

```
1 A = A + B
```

Program 2:

```
1 LD GR0, addr(A)
2 ADD GR0, addr(B)
3 ST GR0, addr(A)
```

この時の LD をロード命令，ST をストア命令といい，主記憶からの読み出しやメモリへの書き込みが行われるため，実行に数クロックかかる．また ADD 命令時にも，値 B が格納されている主記憶から MDR(メモリデータレジスタ) へ読み込みが行われるため，ここでメモリ待ちが発生する．

主記憶の参照回数が多いと，プログラムの実行性能は落ちる．そのため，レジスタに保存されたデータを複数回再利用するコードにすることがプログラムの高速化につながる [17].

3.3 キャッシュ

HA8000 クラスタシステムなどのスカラー並列型スーパーコンピュータの多くで採用されている Intel や AMD などのプロセッサは，キャッシュ [18] を導入している．キャッシュはレジスタより参照速度は遅いが，主記憶よりも参照速度が速い．このため，レジスタに比べ参照速度が遅い主記憶へのアクセスを少しでも高速化するために導入されている．

特に最近のスーパーコンピュータアーキテクチャでは複数 (3 段など) のキャッシュを用意しているものが多い．キャッシュのアーキテクチャは，実データを保存するキャシュテーブルとキャッシュインデックスから成る．

主記憶への参照が行われたとき，まずキャッシュインデックスを参照し，該当アドレスの値がキャッシュに保存されているか調べる．保存されている場合はキャッシュから値が参照される．そうでない場合は，主記憶から値をフェチし，キャッ

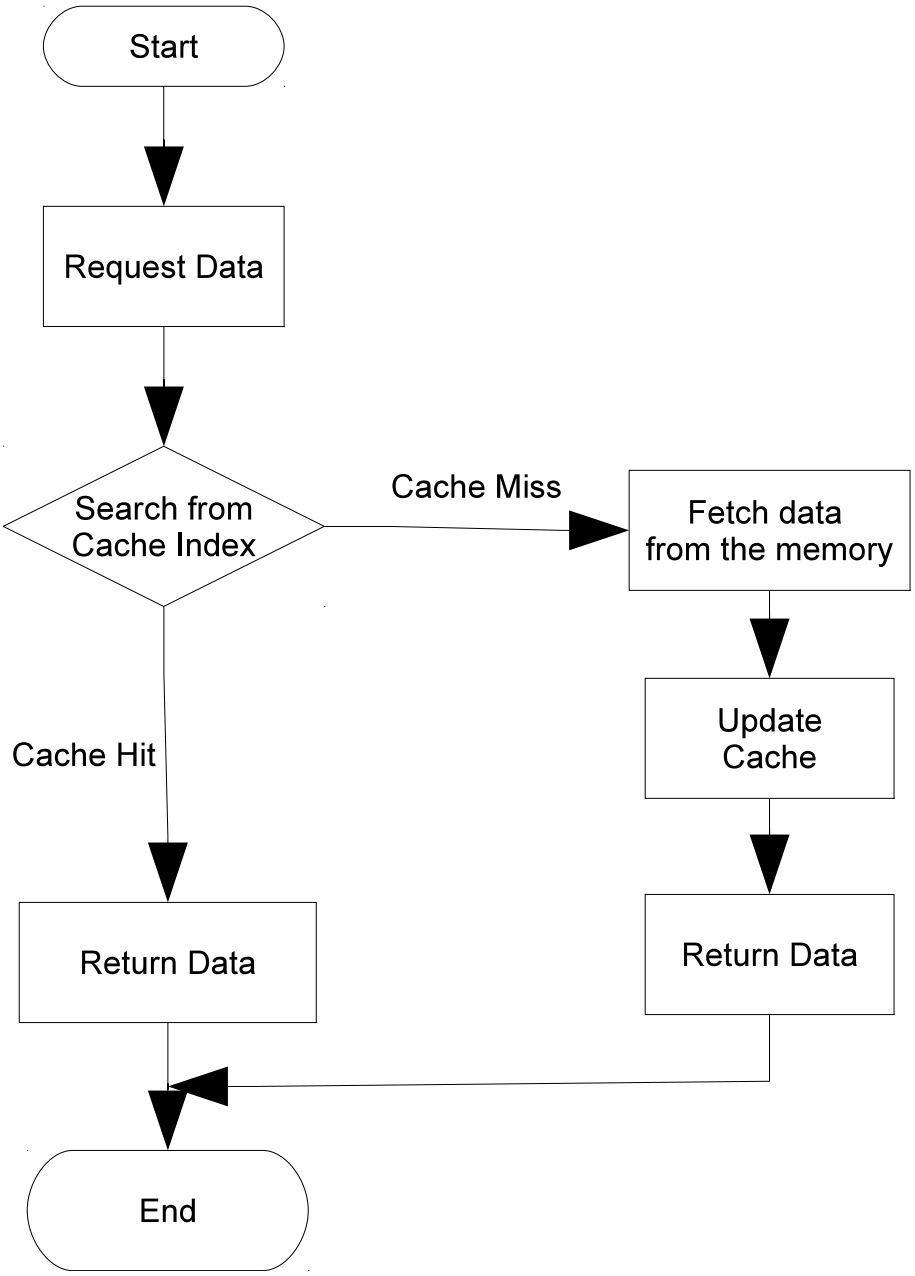

Fig. 3.1: An example of cache operation

シュインデックスを更新する [19]．また，キャッシュテーブルはいくつかのまとまりに分割されていて，これらをキャッシュラインと呼ぶ．キャッシュは空間的局所性に基づいてアクセスのあった付近の値もまとめてキャッシュラインに保存する．Fig.3.1 に 1 段のキャッシュを持つ場合のメモリへの参照のフローチャートを示す．キャッシュに値が見当たらず，主記憶からの読み出しを行うことをキャッシュミスといい，キャッシュミスが多発するとメモリ待ち時間が長くなり実行速度の増大につながる．キャッシュミスを可能な限り少なくすることがスカラー並列型スーパーコンピュータ向け最適化の 1 つの重要なポイントである．

4　スカラー並列型スーパーコンピュータ向け最適化手法

この章では，本研究で行った最適化手法を紹介する．

4.1　プロファイリング

コードの最適化，高速化を行うために，必要不可欠な準備作業がプロファイリングである．プロファイリングの目的はコードのホットスポットを特定でき，最適化の効果を評価することである．

プロファイリングの手法としては，プログラム自体に時間を計測するコードを挿入する方法と専用のプロファイリングツールを使用する方法がある．プロファイリングツールには，プログラムをフックしたり，コンパイル時にプログラムコードにプロファイリングコードを挿入することによってプロファイリングを行う Event-based profiler や Instrumenting profiler と呼ばれるプロファイリングツールと，実行時に定期的にプログラムを監視することによってプロファイリングを行う Statistical profiler と呼ばれるプロファイリングツールとがある．

本研究で使用する磁気流体コードには，プロファイリングのために，プログラムに実行時間を計測するコードが挿入されているが，より詳細なプロファイリングデータをとるために Statistical profiler である gprof [21] を用いる．gprof は Linux システムで動作可能であり，HA8000 クラスタシステムにも標準でインストールされている．gprof では，ある関数毎の実行にかかった時間を細かく計測することができる．またその関数に含まれる外部関数の実行時間と差引することによって，その関数自身を実行するのにかかった時間を計測することができる．

本研究で使用する，Intel Fortran Compiler 11 では，gprof は使用可能である．しかし，その比較対象とする日立製作所製最適化 Fortran コンパイラでは使用することができない．そこで，日立製作所製最適化 Fortran コンパイラでのプロファイリングには，pmpr [20] を用いる．pmpr は gprof と同様，Statistical profiler であり，プログラムの実行にかかった flop 数 (浮動小数点数演算の総実行回数) を計測できるツールである．

4.2　コンパイラオプションによる最適化

最も簡単な最適化手法は，コンパイラオプションを変えること，および他のコンパイラを使ってみることである．現在のコンパイラは，プログラムソースの構文解析を行う事でいくつかの最適化を自動的に適用することができる．コンパイラやオプションによって最適化のレベルが異なるので，様々なパターンを試す必

要がある．ただし過度のコンパイラオプションは，意図しない演算順序の変更を行い，実行結果を壊してしまう可能性があるので注意が必要である．

4.3 レジスタ最適化

レジスタ最適化 [17] とは，プログラムコード中に使用される変数を可能な限りレジスタに乗るように変更することで，メモリへの参照回数を減らし，コードの高速化を図るチューニング手法である．これは，プログラム中の一時変数の共有化や statement の分割によって実現される．

プログラム中で使用される変数が多い場合，汎用レジスタの数が足りなくなる (レジスタ溢れ)．レジスタ溢れが発生すると，レジスタに格納されている変数はストア命令によってメモリへの退避が行われ，後に必要なときにロード命令によって主記憶から参照される．ロード命令とストア命令はメモリのデータ転送速度に依存するため，HA8000 のようにメモリのデータ転送速度が遅いアーキテクチャではこれらの命令の数を可能な限り少なくすべきである．

本研究では，対象コードのロード命令とストア命令を減らすために，ループ内の statement の分割や順序の入れ替え等を行った．また，使用するレジスタ数を減らすために配列の結合操作も行った．これは Program 3 の場合，A1，A2，A3，A4 を一つの配列として，一次元増やした新たな配列 AA にまとめる．その結果は Program 4 となる．レジスタ溢れの有無は，日立最適化 Fortarn コンパイラの出力結果から判断した．詳細については 5.3 章で述べる．

Program 3:

```
1 do j = 1 , NJ
2   do i = 1 , NI
3     A1(i,j) = A1(i,j) * B(i,j)
4     A2(i,j) = A2(i,j) + B(i,j)
5     A3(i,j) = A3(i,j) ** 2
6     A4(i,j) = A4(i,j) + 2 * B(i,j)
7   end do
8 end do
```

Program 4:

```
1 do j = 1 , NJ
2   do i = 1 , NI
3     A(1,i,j) = A(1,i,j) * B(i,j)
4     A(2,i,j) = A(2,i,j) + B(i,j)
```

```
5       A(3,i,j) = A(3,i,j) ** 2
6       A(4,i,j) = A(4,i,j) + 2 * B(i,j)
7     end do
8   end do
```

4.4 配列添字の入れ替え

配列添字の入れ替え [22] は，多重配列への参照の局所性を上げる方法である．たとえば下に示す Program 5 の場合，Fortran 言語の多次元配列では左の添字が優先されメモリに納められる (column major)．そのため，3 行目の配列$\boldsymbol{B}$のように左の添字を j にすると，配列への参照にストライドが発生しキャッシュミスが頻発する．

Program 5:

```
1  do j = 1 , NJ
2    do i = 1 , NI
3      A(i,j) = A(i,j) * B(j,i)
4    end do
5  end do
```

Program 6: Array swap optimization

```
1  do j = 1 , NJ
2    do i = 1 , NI
3      BIJ(i,j) = B(j,i)
4    end do
5  end do
6
7  do j = 1 , NJ
8    do i = 1 , NI
9      A(i,j) = A(i,j) * BIJ(i,<F2>j)
10   end do
11 end do
```

配列$\boldsymbol{B}$の左の添字を i へ Program 6 のように変更すると，キャッシュミスが減る．一見すると無駄な演算が増え，処理が遅くなるように見える．しかし配列$\boldsymbol{B}$への参照が多い場合には，複雑な構造になりがちな後半部分に比べて前半部分は

単純な構造になるので，コンパイラにより後述するループタイリングなどの最適化が施され最終的に高速に動作する場合がある．もちろん配列$\boldsymbol{B}$の添字が(j,i)になるように配列を確保する必要がない場合は，最初から添字が(i,j)になるように配列を確保するべきである．

4.5 ループタイリング

ループタイリングまたは，ブロック化 [23] とは，4.4 章で述べた配列添字の入れ替えができない場合や，Program 7 のように，配列の 1 次元方向以外の，隣の配列に格納されている値を参照したい場合に有効な最適化手法である．Program 7 の場合，i ループをいくつかに分解し Program 8 のように変更する．

Program 7:

```
1 do j = 2 , NJ-1
2   do i = 1 , NI
3     b(i, j) = a(i, j+1) + a(i, j) + a(i, j-1)
4   end do
5 end do
```

Program 8:

```
1 do ii = 1, NI, NSIZE
2   do j = 2 , NJ-1
3     do i = ii , ii + NSIZE - 1
4       b(i, j) = a(i, j+1) + a(i, j) + a(i, j-1)
5     end do
6   end do
7 end do
```

Program 8 における NSIZE をブロックサイズという．ブロックサイズはループ内で参照される配列サイズ，キャッシュのラインサイズ，ライン数やから決まる．

キャッシュラインサイズが 3 でキャッシュライン数が 3 の場合 b(1,2) の値を計算する時に配列 a の a(1,1), a(1,2) a(1,3) の 3 つの値を使う．この時，それぞれの値が納められているメモリ周辺の値がキャッシュに格納される．説明のため例えば [a(1,1), a(2,1) a(3,1)]，[a(1,2), a(2,2) a(3,2)]，[a(1,3), a(2,3) a(3,3)] の 9 つの値がキャッシュにコピーされたとする．次にループカウンタの i が 1 から 2 にインクリメントとして b(2,2) の値を計算する時，必要な配列 a の値は a(2,1), a(2,2), a(2,3) の 3 つである．これらはすべてキャッシュに乗っているので高速に参照すること

ができる．さらに次のステップ (i=3) の場合も同様であり，b(3,2) 計算に必要な配列 a の値 a(3,1), a(3,2), a(3,3) の値は全てキャッシュに乗っている．

ブロック化の有無による効果は次のステップで現れる．ブロック化を行わないプログラム (Program 7) の場合，次のステップは i=4 である．b(4,2) を計算するのに必要となる配列 a の値は a(4,1), a(4,2), a(4,3) の 3 つであるが，この値はどれもキャッシュに乗っていない．従って主記憶から再び 9 つの値，[a(4,1), a(5,1) a(6,1)]，[a(4,2), a(5,2) a(6,2)]，[a(4,3), a(5,3) a(6,3)] をコピーする必要がある．

一方，ブロック化を施したプログラム (Program 8) では，次のステップでは j をインクリメントして i=1,j=3 の値，即ち b(1,3) の計算を行う．b(1,3) の計算に必要な配列 a の値は a(1,2), a(1,3), a(1,4) の 3 つであり．このうち a(1,2) と a(1,3) の 2 つはキャッシュに乗っているので高速にアクセスできる．a(1,4) の値はメモリからとってくる必要があるので，この時 [a(1,4), a(2,4), a(3,4)] がメモリからキャッシュにコピーされる．キャッシュ上にあった [a(1,1), a(2,1), a(3,1)] の値は破棄される．以上の動作を Fig. 4.1 にまとめた．

このようにブロック化を行うことでキャッシュ上のデータを有効に再利用することができる．

memory	step 1 (*)need for eval b(1,2)=	step 1 cache before eval	step 1 cache after eval	step 2 (*)need for eval b(2,2)=	step 2 cache before eval	step 2 cache after eval	step 3	step 4 (*)need for eval b(4,2)=	step 4 cache before eval	step 4 cache after eval
a(1 , 1)	*	cache	a(1 , 1)		a(1 , 1)	a(1 , 1)			a(1 , 1)	
a(2 , 1)		miss →	a(2 , 1)	*	a(2 , 1)	a(2 , 1)			a(2 , 1)	
a(3 , 1)			a(3 , 1)		a(3 , 1)	a(3 , 1)			a(3 , 1)	
a(4 , 1)								*	cache	a(4 , 1)
a(5 , 1)									miss →	a(5 , 1)
a(6 , 1)										a(6 , 1)
a(1 , 2)	*	cache	a(1 , 2)		a(1 , 2)	a(1 , 2)			a(1 , 2)	
a(2 , 2)		miss →	a(2 , 2)	*	a(2 , 2)	a(2 , 2)			a(2 , 2)	
a(3 , 2)			a(3 , 2)		a(3 , 2)	a(3 , 2)			a(3 , 2)	
a(4 , 2)								*	cache	a(4 , 2)
a(5 , 2)									miss →	a(5 , 2)
a(6 , 2)										a(6 , 2)
a(1 , 3)	*	cache	a(1 , 3)		a(1 , 3)	a(1 , 3)			a(1 , 3)	
a(2 , 3)		miss →	a(2 , 3)	*	a(2 , 3)	a(2 , 3)			a(2 , 3)	
a(3 , 3)			a(3 , 3)		a(3 , 3)	a(3 , 3)			a(3 , 3)	
a(4 , 3)								*	cache	a(4 , 3)
a(5 , 3)									miss →	a(5 , 3)
a(6 , 3)										a(6 , 3)
.										
.										
.										
a(6 , 6)										

without blocking (Program 7)

memory	step 1 (*)need for eval b(1,2)=	step 1 cache before eval	step 1 cache after eval	step 2 (*)need for eval b(2,2)=	step 2 cache before eval	step 2 cache after eval	step 3	step 4 (*)need for eval b(1,3)=	step 4 cache before eval	step 4 cache after eval
a(1 , 1)	*		a(1 , 1)		a(1 , 1)	a(1 , 1)			a(1 , 1)	
a(2 , 1)		cache	a(2 , 1)	*	a(2 , 1)	a(2 , 1)			a(2 , 1)	
a(3 , 1)		miss →	a(3 , 1)		a(3 , 1)	a(3 , 1)			a(3 , 1)	
a(4 , 1)										
a(5 , 1)										
a(6 , 1)										
a(1 , 2)	*		a(1 , 2)		a(1 , 2)	a(1 , 2)		*	a(1 , 2)	a(1 , 2)
a(2 , 2)		cache	a(2 , 2)	*	a(2 , 2)	a(2 , 2)			a(2 , 2)	a(2 , 2)
a(3 , 2)		miss →	a(3 , 2)		a(3 , 2)	a(3 , 2)			a(3 , 2)	a(3 , 2)
a(4 , 2)										
a(5 , 2)										
a(6 , 2)										
a(1 , 3)	*		a(1 , 3)		a(1 , 3)	a(1 , 3)		*	a(1 , 3)	a(1 , 3)
a(2 , 3)		cache	a(2 , 3)	*	a(2 , 3)	a(2 , 3)			a(2 , 3)	a(2 , 3)
a(3 , 3)		miss →	a(3 , 3)		a(3 , 3)	a(3 , 3)			a(3 , 3)	a(3 , 3)
a(4 , 3)										
a(5 , 3)										
a(6 , 3)										
.										
.										
a(1 , 4)								*	cache	a(1 , 4)
a(2 , 4)										a(2 , 4)
a(3 , 4)									miss →	a(3 , 4)
.										
.										
a(6 , 6)										

with blocking (Program 8)

Fig. 4.1: Effects of cache optimization by blocking

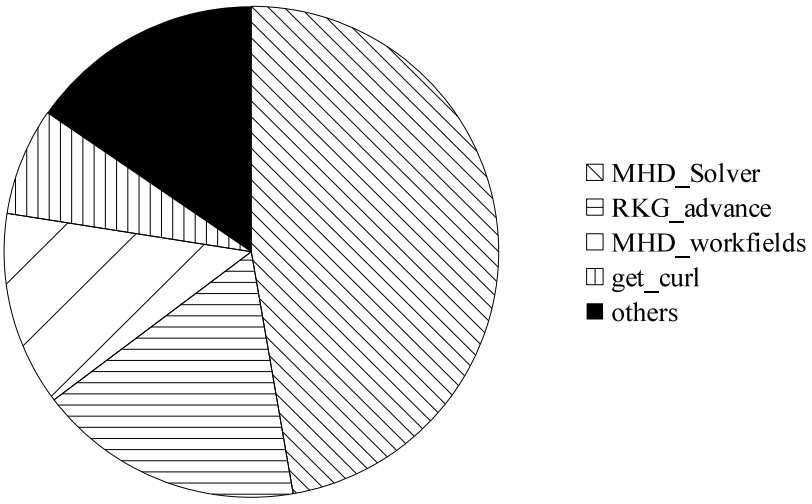

Fig. 5.1: Result of profiling

5　最適化とその効果

本章では，前章で説明した最適化手法を 2.3 章で示した地球ダイナモシミュレーション用 MHD コードに適用した結果をまとめる．

5.1　プロファイリング結果

gprof でのプロファイリング結果を Fig.5.1 に示す．これは Intel Fortran Compiler 11 での実行結果である．問題サイズは L2 キャッシュよりも大きいサイズで，総格子点数は $255 \times 258 \times 770 \times 2$(最後の係数 2 は Yin-Yang の Yin と Yang である) とした．並列化はノード数を 4，コア数を 16 として実行した．

Fig.5.1 の MHD_Solver は差分化された MHD 方程式を解くサブルーチンである．RKG_advance は Runge-Kutta 法による積分ルーチン，MHD work_fields は MHD の基本場以外の補助的な 3 次元場を基本場から計算するサブルーチン，get_curl はベクトル場の curl を計算するルーチンである．others 以外の部分はそれぞれのルーチン内での処理にかかった時間であり，そのルーチンから参照している外部ルーチンは含まれていない．初期化処理，MPI 通信や MPI_Barrier 等の処理はすべて others に含まれる．

Fig.5.1 から，初期化や通信処理には大きな時間がさかれておらず，MHD_Solver がこのコードのホットスポットになっていることがわかる．MHD_Solver は非常に大きな 3 重ループであり，多数の浮動小数点数演算を含んでいる．従って，本研究では，MHD_Solver を中心に最適化を施した．

Table 5.1: Comparison of execution speeds various for compilers and their options.

Compiler Vendor	Compile Option	GFlops
(1) Hitachi	-O3 -noparallel	13.01
	-Oss -noparallel -autoinline=2	
	-nolimit -noscope	17.04
	-O4	15.15
(2) Intel Fortran Compiler 11.0	-O3 -xSSE3 -msse3	20.66
	-O3 -xSSE3 -msse3 -ipo	20.63
	-O3 -xSSE3 -msse3 -ipo9	20.69k
	-O3 -xSSE3 -msse3 -ipo20	20.62k
	-O4	20.7
(3) PGI Fortran Compiler	-fast -O3 -tp=baracelona-64	
(4) GNU Fortran Compiler	-O3 -m64 -march=opteron	

5.2 コンパイラオプションによる最適化

各コンパイラによる性能評価結果を Table 5.1 に示す．使用したコンパイラは HA8000 クラスタシステム (T2K 東大) にインストールされている，(1) 日立製作所製最適化 Fortran コンパイラ，(2) Intel Fortran Compiler 11，(3) PGI Fortran コンパイラ，(4) GNU Fortran Compiler の 4 種である．総格子点数を $255\times258\times770\times2$ とし，ノード数は 4 で 1 ノードあたり 16 コアの 64 コアとし，ステップ数が 100 以上になるように 5 分間ジョブを流した．問題サイズは L2 キャッシュに乗らない大きさの問題を想定した．プログラムはどのコンパイラでもコンパイルでき，正常に実行する事ができた．日立最適化 Fortarn コンパイラと Intel Fortran Compiler 11 では正常に性能評価ができた．PGI Fortran コンパイラと GNU Fortran Compiler については，実行速度が遅くステップ数が 100 に達しなかったため性能評価ができなかった．

Table 5.1 から，何も最適化を施さない場合については，日立製コンパイラに比べ Intel 製コンパイラの方が 20%ほど速い事が分かる．以下では日立製コンパイラと Intel 製コンパイラに焦点を当てて，最適化の結果を示す．

5.3 レジスタ最適化

レジスタ最適化による効果を Fig.5.2 に示す，縦軸は 1 秒間に行われた浮動小数点演算数 (単位は MFlops) であり，横軸は領域分割した MPI プロセス 1 つあたり

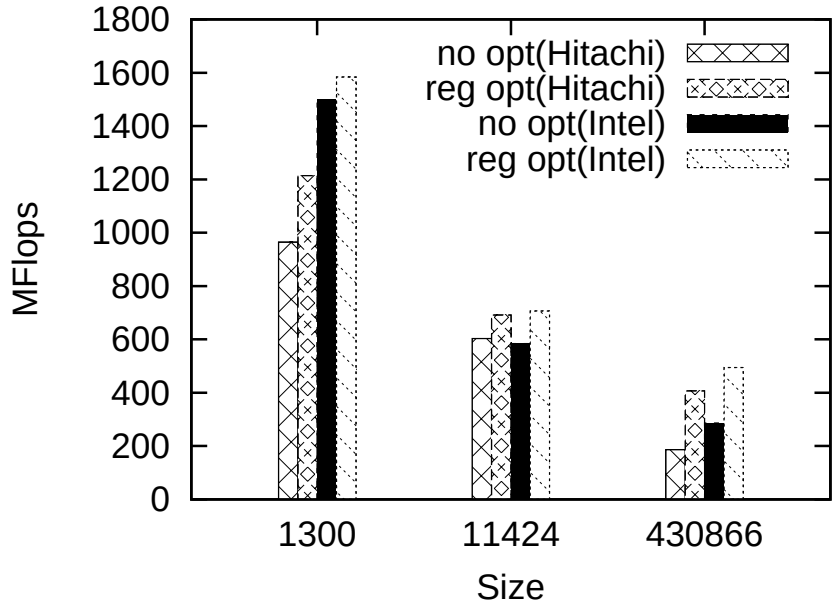

Fig. 5.2: Effects of Register Optimization(MHD_Solver)

の格子点数である．これらの値は以下の特徴を持つ:

- 問題サイズ 1300($10 \times 16 \times 44$) は L1 キャッシュに乗るサイズ．
- 問題サイズ 11424($28 \times 30 \times 86$) は L2 キャッシュに乗るサイズ．
- 問題サイズ 430866($101 \times 54 \times 79$) は L2 キャッシュに乗らないサイズである．

Fig.5.2 のグラフの中で no opt が最適化を適用する前，reg opt がレジスタ最適化を適用した結果である．Hitachi が日立製作所製最適化 Fortran コンパイラでの結果，Intel が Intel Fortarn Compiler 11 での結果である．ホットスポットである MHD のソルバー部分に最適化を施し計測を行った．浮動小数点演算数については，HA8000 に搭乗されているプロファイリングツール pmpr による結果を用いた．日立製作所製最適化 Fortran コンパイラでの実行時間の計測には同ツールの pmpr を使用した．また Intel Fortran Compiler 11 での実行時間の計測には gprof を使用した．

Fig.5.2 から，どの問題サイズ，コンパイラにおいてもレジスタ最適化による性能向上がみられることがわかる．異なる問題サイズで比較した場合，同じコンパイラであっても，問題サイズが大きい方がレジスタ最適化の効果が大きい．最も性能が向上したケースは，問題サイズ 430866 上で日立製コンパイラでの結果である．この時，最適化適用前と比べると 100%以上の性能向上が得られた．

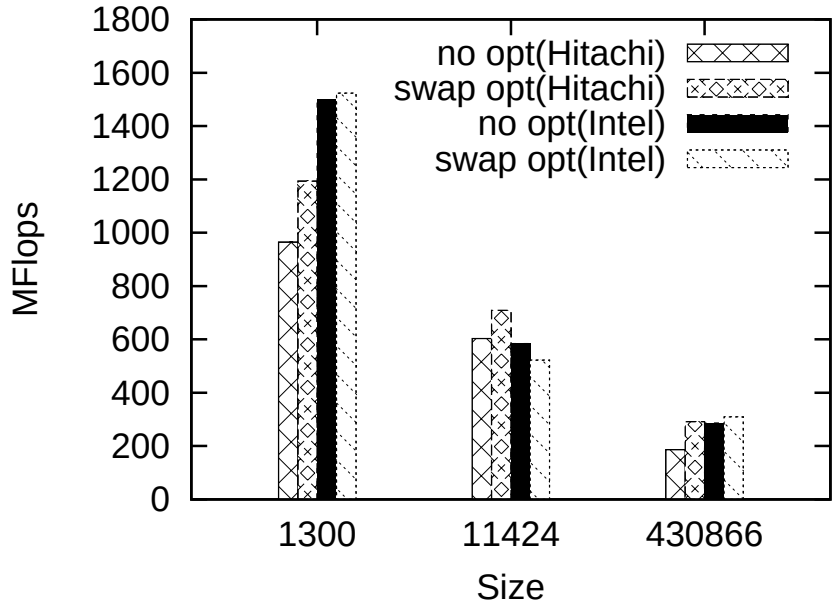

Fig. 5.3: Effects of array dimension swapping (MHD_Solver)

5.4 配列添字の入れ替え

配列添字の入れ替えによる効果を Fig.5.3 に示す，Fig.5.2 と同様に縦軸は MFlops，横軸は MPI プロセス 1 つあたりの格子点数である．問題サイズ 1300 は L1 キャッシュに乗るサイズ．問題サイズ 11424 は L2 キャッシュに乗るサイズ．問題サイズ 430866 は L2 キャッシュに乗らないサイズである．計測は pmpr と gprof を使用した．最適化を施す前が no opt で最適化を施した後が swap opt である．

この最適化を行うにあたっては 5.3 章と異なり対象ルーチンを MHD_Solver 以外に広げて最適化を行った．その結果は Fig.5.4 である．

続いて，5.3 章のレジスタ最適化を適応したコードに，配列添字の入れ替えを適応した結果を Fig.5.5，Fig. 5.6 に示す．レジスタ最適化を行った結果が reg opt，レジスタ最適化にさらに配列添字の入れ替えを行った結果が swap opt である．

Fig.5.5 から，キャッシュが有効活用される大きな問題サイズでわずかながら性能向上がみられることが分かる．問題サイズの小さい 2 つについては，性能向上するよりも性能劣化してしまった．これは，最適化を適応しても，そもそも配列がキャッシュに乗ってしまっているため，キャッシュミスが発生せずに余分なオーバーヘッドにより性能劣化してしまったためであろう．

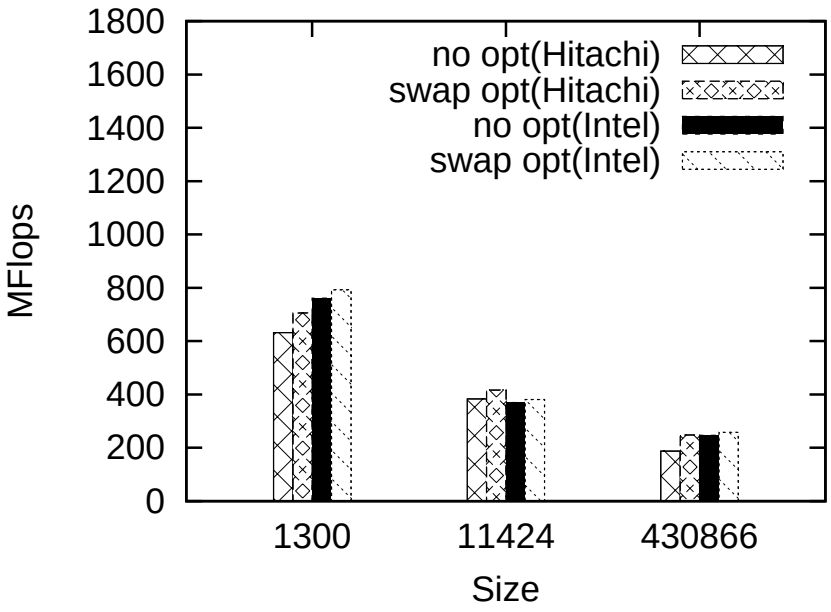

Fig. 5.4: Effects of array dimension swapping (whole)

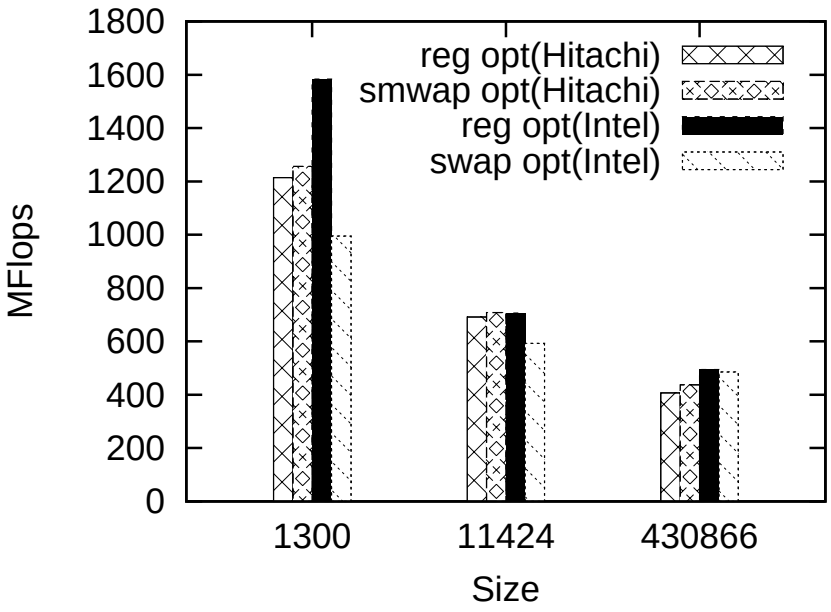

Fig. 5.5: Effects of array dimension swapping (MHD_Solver)

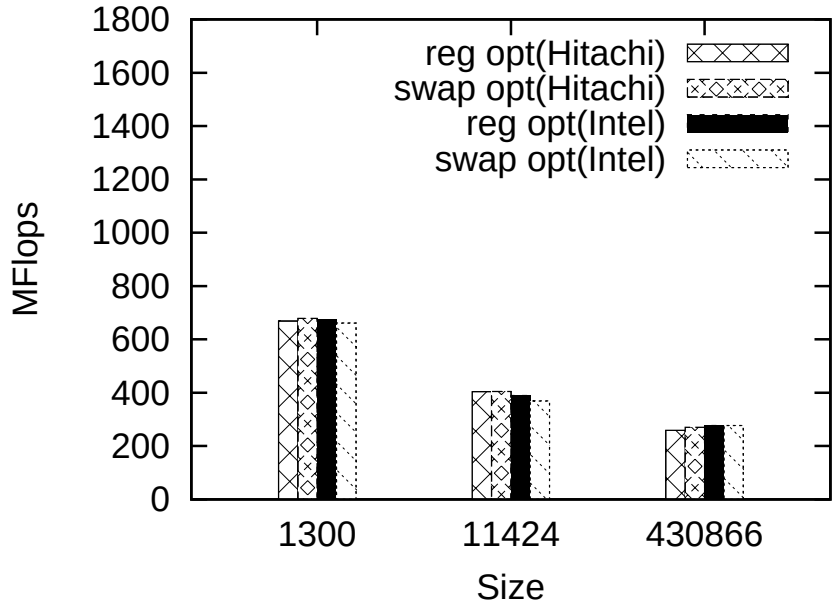

Fig. 5.6: Effects of array dimension swapping (whole)

5.5 ループタイリング

ループタイリングの有無による計算性能の違いをまとめたのが Fig.5.7 である．Fig.5.2 と同様に縦軸は Flops，横軸は MPI プロセス 1 つあたりの格子点数である．swap opt が 5.4 章の配列添字の入れ替えを適用した結果．blocked がさらにループタイリングを適用した結果である．

Fig.5.7 より，どの問題サイズでも性能劣化していることがわかる．問題サイズが小さい 2 つについては，配列が L2 キャッシュに乗ってしまっているので，余分な処理によってオーバーヘッドが増えたことが原因であろう．ループタイリングによる最適化は，タイルのサイズ (ループの分割数) や問題サイズなどに大きく依存する．そのため，問題サイズ 430886 について，適切なタイルサイズと問題サイズが設定されていないため性能劣化したと考えられる．また，レジスタ最適化によるループ分割によってループオーバーヘッドが上昇したのも原因と考えられる．一般にループタイリングを行わなければ Fig.5.2 のように問題サイズが大きいと，キャッシュミスが頻発するため，ループタイリングは必須である．

本研究では，適切なタイルサイズの評価を行うための十分な研究を行うことができなかった．この点は今後の課題である．

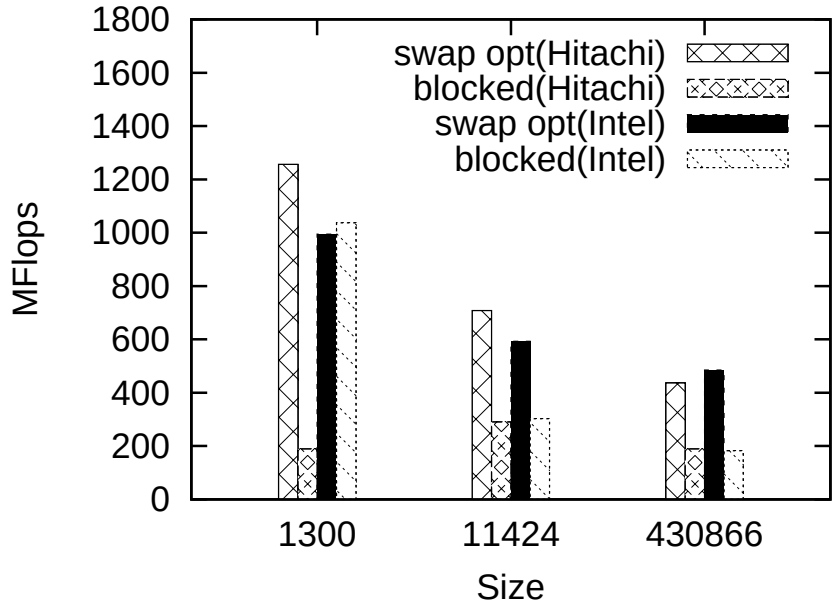

Fig. 5.7: Effects of Looptiling(MHD_Solver)

6 まとめ

地球シミュレータ (ベクトル並列型スーパーコンピュータ) 向けに最適化された磁気流体コードのスカラー並列型スーパーコンピュータ向け最適化を行った．レジスタ最適化では，日立製作所製最適化 Fortran コンパイラ，Intel Fortran Compiler 11 ともに高速化され，どの問題サイズでも高速化を確認できた．行列の添字入れ替えによる最適化では，ループタイリングでは，どのコンパイラ，問題サイズにおいても高速化は確認できなかった．もっとも最適化の効果があったのはレジスタ最適化であり，日立コンパイラを使用した場合，大きな問題サイズでは 100%以上の性能向上が見られた．

今後の課題はループタイリングを適切に行い，大きな問題サイズでも性能低下を引き起こさない最適化をコードに施す事である．

参考文献

[1] http://www.mext.go.jp/b_menu/shingi/gijyutu/gijyutu2/007/shiryo/05092001/003.pdf

[2] http://www.nscc-tj.gov.cn/en/

[3] http://www.top500.org/

[4] Kevin Dowd, “High Performance Computing,” 第 1 版, オーム社 (1994)

[5] “HA8000 クラスタシステム利用の手引き,”
http://www.cc.u-tokyo.ac.jp/service/ha8000/ha8000-tebiki/

[6] http://products.amd.com/pages/OpteronCPUDetail.aspx?id=420

[7] ジョン・L. ヘネシー デイビッド・A. パターソン, “コンピュータの構成と設計 - ハードウエアとソフトウエアのインタフェース〈上〉,” 第 3 版, 日経 *BP* 社 (2006)

[8] “A Report on Computer Processing Capability for the Magnetohydrodynamic Simulation Model,”
http://center.stelab.nagoya-u.ac.jp/web1/simulation/hpfja/comput04.html

[9] 深沢 圭一郎, 梅田 隆行, 荻野 瀧樹, “電磁流体コートによる大規模惑星磁気圏シミュレーション,” スーパーコンピューティングニュース Vol.12 特集号 1 (2010)

[10] 陰山 聡, “コンパスはなぜ北を指すのか?,” 岩波「科学」vol.77, pp.532–538 (2007)

[11] P.A. Davison, “An Introduction to Magnetohydrodynamics,” *Cambridge University Press* (2001)

[12] 山浦 優気, 陰山 聡, “地球ダイナモの新しいシミュレーションコード開発とその応用,” スーパーコンピューティングニュース (印刷中)

[13] Akira Kageyama and Tetsuya Sato, “”Yin-Yang Grid”: An Overset Grid in Spherical Geometry,”Geochem. Geophys. Geosyst., Q09005, doi:10.1029/2004GC000734; arXiv:physics/0403123 (2004)

[14] 陰山 聡, “陰陽格子の開発,” 第 1 7 回数値流体力学シンポジウム (2003)

[15] A. Kageyama et al., “A 15.2 TFlops Simulation of Geodynamo on the Earth Simulator,” *Proc. ACM/IEEE Conference SC2004 (Super Computing 2004)*, pp.35-43 (2004)

[16] David A. Patterson and John L. Hennessy, “Computer Organization and Design, Fourth Edition: The Hardware/Software Interface,” *Morgan Kaufmann* (2008)

[17] Charles Severance and Kevin Dowd, “High Performance Computing,” Second Edition, *O’Reilly Media* (1998)

[18] 大久保 英嗣, “オペレーティングシステムの基礎,” サイエンス社 (1997)

[19] Greg Fry, “Implementing AMD cache-optimal coding techniques,” http://developer.amd.com/documentation/articles/pages/implementingamdcache-optimalcodingtechniques.aspx (2008)

[20] （株）日立製作所, “HA800 クラスタシステム　性能モニタ機能の利用法,” スーパーコンピューティングニュース Vol.12 No.6, pp. 26-34 (2010)

[21] http://www.gnu.org/software/binutils/

[22] 青山 幸也, “チューニング技法入門,” 理化学研究所, 情報基盤センター, チューニング講習会資料 (2004)

[23] 寒川 光, “RISC 超並列化プログラミング技法,” 共立出版 (1995)

謝辞

本研究の実施にあたって，HA8000 クラスタシステム (T2K 東大) を使用させていただきました．最適化の実験に用いた MHD コードは指導教官である陰山教授が開発した yycore コードです．陰山教授には終始指導をしていただいたことを感謝いたします．また，東京大学の片桐准教授には最適化手法について，御助言をいただきました．この研究と論文のとりまとめにあたり様々な御助言を頂いた政田助教に感謝します．最後に，論文の確認をしていただいた目野氏と吉崎氏に感謝します．

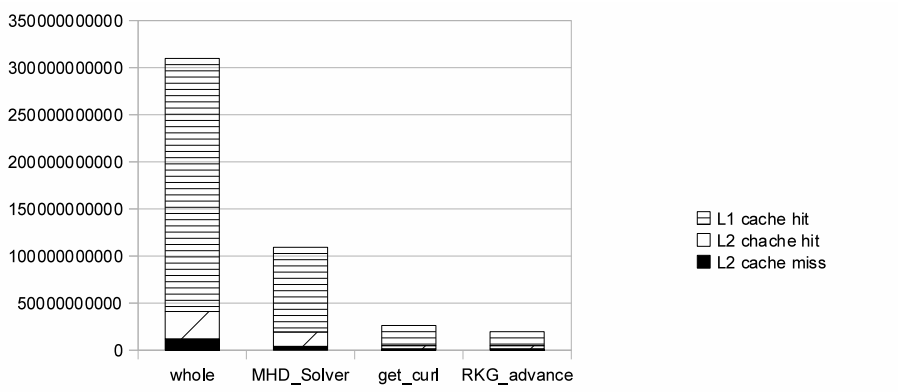

Fig. A.1: Results of Cachegrind

付録

HA8000 クラスタシステム (T2K 東大) での最適化について，スーパーコンピューティングニュース [12] で報告した．本付録はその内容をまとめたものである．

A　キャッシュヒット率の計測とその他の最適化

A.1　キッシュヒット率の計測

ダイナモコードの最大のホットスポットである MHD のソルバー部分を解析した．このルーチンは，3 次元空間に対応した 3 重 do-loop である．このループのキャッシュヒット率をキャッシュシミュレータで計測した．キャッシュヒット率を計測するツールとしては，oprofile や cachegrind（Valgrind）などがあるが，HA8000 クラスタシステムでは，oprofile を使用できなかったため，cachegrind を使用した．cachegrind はメモリ関連のプロファイリングツールである Valgrind に付属するプロファイラーであるが，I1， L1， L2（最新版ではもっとも大きなキャッシュである LL キャッシュ）の 3 つのキャッシュをシミュレートすることによって，キャッシュヒット率を擬似的ではあるが，計測することができる．ただし，あくまでシミュレータであることや，HA8000 クラスタシステムに搭載されている CPU である Opteron には L3 キャッシュ（しかもサイズ 2MByte と大きい）が搭載されているため，必ずしも正確なキャッシュヒット率とはならないことに留意する必要がある．キャッシュシミュレータによる実行結果を Fig.A.1 に示す．

なお，HA8000 クラスタシステムに cachegrind は，はじめからインストールされていたが，バージョンが古く，MPI 関連の最適化がなされていない事や複数プロセスのプロファイル結果を合成することができないため，新たにバージョン 3.5.0 をコンパイルして使用した．

A.2　ループ分割による最適化

ホットスポットを確認し，キャッシュヒット率を計測したところで，いくつか最適化を試みた．

メイン計算部分は 3 重ループで最内ループ内に複雑に多くの計算式がつらなっているが，これをいくつかのループに分割することを試みた．まず，ループ内で依存関係がない 2 つのブロック見つけ，単純にこれらを 2 つに分割してみた（Way1）．

MHD 方程式の右辺には，依存関係はあるものの，temporary な変数（例えば基本場の 2 次の非線形項）の設定部分と，それを利用する部分の二つに分けられるものが多い．そこで，そのような temporary な変数を 1 次配列に変更し，Program 9 を Program 10 のように 3 重ループの最内ループのみを分割する方法も試してみた（Way2）．

Program 9:

```
1 do k = 1 , NK
2   do j = 1 , NJ
3     do i = 1 , NI
4       tmp = a1(i,j,k)*a2(i,j,k)
5       b1 = tmp**2
6       b2 = tmp*a3(i,j,k)
7     end do
8   end do
9 end do
```

Program 10:

```
1 do k = 1 , NK
2   do j = 1 , NJ
3     do i = 1 , NI
4       tmp(i) = a1(i,j,k)*a2(i,j,k)
5     end do
6     do i = 1 , NI
7       b1 = tmp(i)**2
8       b2 = tmp(i)*a3(i,j,k)
9     end do
10  end do
11 end do
```

MHD 計算は 8 つの物理場をもつが，上記の temporary 変数を 1 次配列にしたことで，各物理量の計算を分割することが可能になったため，それらの計算を 8 つに分割してみた．上記では，b1，b2 の計算部分を分割する．(Way3)

キャッシュシミュレータによる結果を Fig.A.2，実行結果を Table A.1 に記す．いずれも 8 ノード 128 コアでの実行結果である．

Way2，Way3 では，Fig.A.2 のようにキャッシュ読み込み数が大幅に増え，L2 キャッシュミスヒット数も増えており，Table A.1 から分かるようにプログラム全体が遅くなる結果となった．いずれも，性能をあげることはできず，むしろ遅くなってしまう場合もあった．

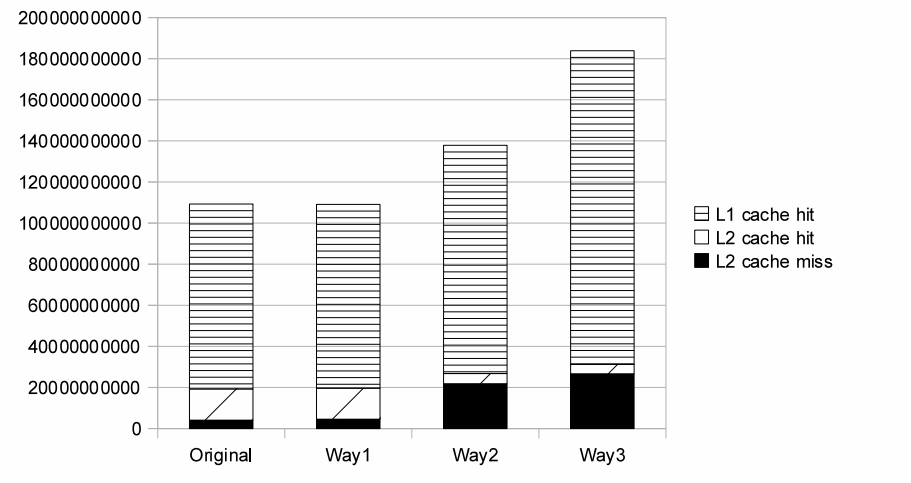

Fig. A.2: Effects of loop division: 1 step.

Table A.1: Effects of loop division: 200 steps with 8 nodes(128 cores). Way2 and Way3 is slower than Original.

	GFlops
Original	40.58
Way1	39.15
Way2	37.99
Way3	36.28

A.3　行列の入れ替え

次に行と列を入れ替える手法を試みた．これは以下の Program 12 のように，b のアクセスが不連続なため，キャッシュミスが多発するのを事前に b2 という転置行列を作成し防ぐものである．Program 11 を変更した結果は Program 12 となる．

Program 11:

```
1  do j = 1 , NJ
2    do i = 1 , NI
3      a(i,j) = a(i,j) + b(j,i)
4    end do
5  end do
```

Program 12:

```
1  do i = 1 , NI
2    do j = 1 , NJ
3      b2(i,j) = b(j,i)
4    end do
5  end do
6
7  do j = 1 , NJ
8    do i = 1 , NI
9      a(i,j) = a(i,j) + b2(i,j)
10   end do
11 end do
```

MHD ソルバー部分にこのような配列があったため，この方法を試みたのが（Way4）である．

また，いくつかあるループ内において，該当の配列へのアクセスが連続となっているのが，主にプログラムの初期化部分であったため，配列を元から行と列を入れ替えたのが（Way5）である．これらの結果を Fig.A.3，Table A.2 に示す．

なお，これらの図表における，「Original」のキャッシュヒット率や GFlops 値はこれまでに記述したチューニングを施した上での実行である．

Fig.A.3 は MHD ソルバー部分のみの結果である．Way4 と Way5 ではそれほど差が見られないものの，Way5 はプログラム全体に渡って変更したため，MHD ソルバー部分以外でのキャッシュミスヒット数が減り，その結果として Table A.2 のように全体で 7%ほどの高速化になった．Way4，Way5，とキャッシュミスヒット

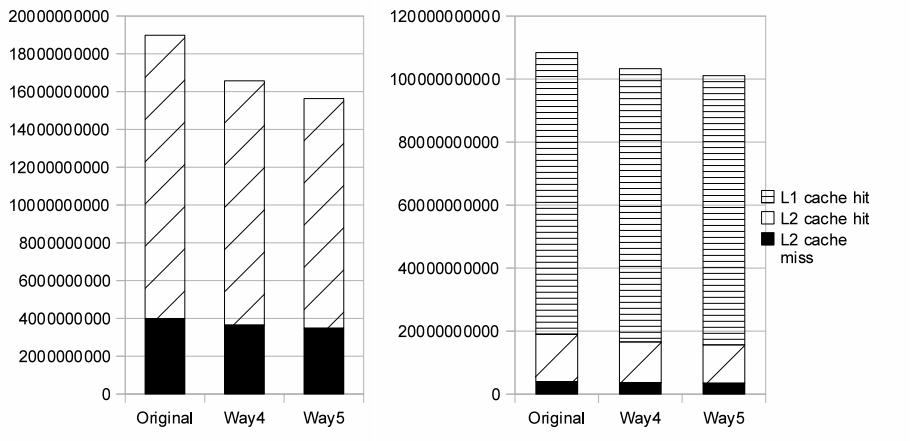

Fig. A.3: Effects of swapping: 1 step with 1 node(16 cores).

Table A.2: Effects of swapping: 200 steps with 8 nodes(128 cores).

	GFlops
Original	40.85
Way4	41.78
Way5	43.66

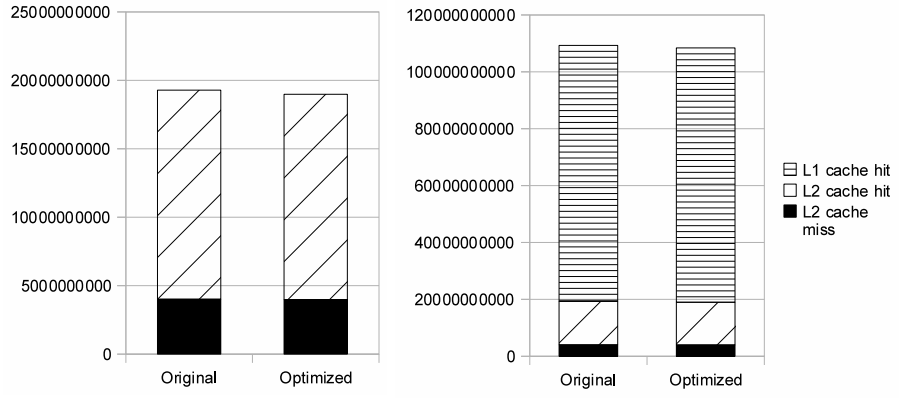

Fig. A.4: Effects of swap operation: 1 step with 1 node(16 cores).

Table A.3: Effects of swap operation

	GFlops
Original	40.58
Optimized	40.85

率は増えているものの，キャッシュにアクセスする回数，およびキャッシュミスヒット数が減ったことになる．

A.4 配列へのアクセス変更

非常に単純ではあるが，以下のプログラムにおいて，配列 a へのアクセスを明示的に変更する事によって若干ではあるがキャッシュミスが減少した．これは，コンパイラが自動で行ってくれていると判断していたが，3 重ループ構造が複雑であるためか，意外にもそうではなかったようである．

この結果は Fig.A.4，Table A.3 のようになり．1%ほどではあるが高速化された．例として Program 13 を変更前，Program 14 を変更後として示す．

Program 13:

```
1 do j = 1 , NJ
2   do i = 1 , NI
3     b(i,j) = a(i+1,j) + a(i-1,j)
4   end do
```

```
5 end do
```

Program 14:

```
1 do j = 1 , NJ
2   do i = 1 , NI
3     b(i,j) = a(i-1,j) + a(i+1,j)
4   end do
5 end do
```

A.5 まとめ

地磁気ダイナモ MHD シミュレーションコードのチューニングを目指し，準備的な研究を進めた．本付録では主に cachegrind というキャッシュシミュレータを使い，性能評価しつついくつかの最適化を試みた．演算性能の向上という点では大きな結果は得られなかったが，今後チューニングを進めるにあたり指針となる結果が得られた．ただし，cachegrind はあくまでシミュレータであることと，実行時のオーバーヘッドが大きく，数ループ分のデータしかとれなかったのが残念である．実行時オーバーヘッドの少ない oprofile もしくは他のサンプリング型キャッシュプロファイラーを使用できるようになれば大変ありがたいと感じた．