ISSN 1344—9567

スーパーコンピューティングニュース

Vol. 9 No. Special Issue 1 2008. 2

特集：コーディングしてみよう！
～ スパコンプログラミングを極めるⅠ ～

東京大学情報基盤センター
（スーパーコンピューティング部門）

特集：コーディングしてみよう！　～　スパコンプログラミングを極めるⅠ　～

特集号の発行にあたって

第１部　HITACHI SR11000/J2 の高速化技術

数値計算ライブラリーMATRIX/MPP、MATRIX/MPP/SSS、MSL2 のご紹介

（株）日立製作所

GeoFEM ベンチマークによる Hitachi SR11000/J2 の性能評価

中島研吾

実対称固有値問題に対する多分割の分割統治法の SR11000 への一実装

桑島豊　坪谷怜　田村純一　重原孝臣

スーパーコンピュータ SR11000 でのプログラム開発事例：
カップルドクラスター法による高分子の電子状態計算

片桐秀樹

第２部　T2K オープンスパコンの高速化技術

オープンスパコンの OS とアーキテクチャの基礎

松葉浩也

キャッシュ性能安定性について

今村俊幸

FFT におけるキャッシュ最適化方式

高橋大介

電磁場解析における数値ソルバのキャッシュ最適化チューニング

岩下武史

第３部　共通高速化技術

C 言語による OpenMP 入門

黒田久泰

C 言語による MPI プログラミング入門

片桐孝洋

スーパーコンピューティング

ニュース

特集 : コーディングしてみよう！　～　スパコンプログラミングを極めるⅠ　～

目　　次

特集：コーディングしてみよう！　～　スパコンプログラミングを極めるⅠ　～

特集号の発行にあたって・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・1

第１部　HITACHI SR11000/J2 の高速化技術

数値計算ライブラリーMATRIX/MPP、MATRIX/MPP/SSS、MSL2 のご紹介・・・・・・・・・・・5
（株）日立製作所

GeoFEM ベンチマークによる Hitachi SR11000/J2 の性能評価・・・・・・・・・・・・・・・ 21
中島研吾（東京大学大学院理学系研究科）

実対称固有値問題に対する多分割の分割統治法の SR11000 への一実装・・・・・・・・・・47
桑島豊　坪谷怜　田村純一　重原孝臣（埼玉大学理工学研究科）

スーパーコンピュータ SR11000 でのプログラム開発事例：
カップルドクラスター法による高分子の電子状態計算・・・・・・・・・71
片桐秀樹（産業技術総合研究所）

第２部　T2K オープンスパコンの高速化技術

オープンスパコンの OS とアーキテクチャの基礎・・・・・・・・・・・・・・・・・・・・・95
松葉浩也（東京大学情報基盤センター）

キャッシュ性能安定性について・・・・・・・・・・・・・・・・・・・・・・・・・・・ 111
今村俊幸（電気通信大学）

FFT におけるキャッシュ最適化方式・・・・・・・・・・・・・・・・・・・・・・・・・ 123
高橋大介（筑波大学大学院システム情報工学研究科）

電磁場解析における数値ソルバのキャッシュ最適化チューニング・・・・・・・・・・・ 135
岩下武史（京都大学学術情報メディアセンター）

第３部　共通高速化技術

C 言語による OpenMP 入門・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 149
黒田久泰（東京大学情報基盤センター）

C 言語による MPI プログラミング入門・・・・・・・・・・・・・・・・・・・・・・・・ 169
片桐孝洋（東京大学情報基盤センター）

特集号の発行にあたって

東京大学　情報基盤センター
特任准教授　片桐孝洋

東京大学情報基盤センタースーパーコンピューティング部門では、隔月で広報誌『スーパーコンピューティングニュース』を発行しています。昨年度、初めての試みとして、利用者の研究成果を異分野の利用者・学生等に紹介する特集号〈 スーパーコンピューターの拓く世界 ～ 異分野の研究に触れる ～ 〉を発行しました。第一線で活躍している研究者がスーパーコンピューターを活用し、多分野にわたる研究手法・成果をわかりやすく読者にお伝えすることにより、教員のみならず学生にもスーパーコンピューター利用とその成果を身近に知る手引きとして好評に迎えられることができました。

今年度はさらに焦点を絞り、スーパーコンピューター向けの具体的なプログラミング技法を紹介する特集号〈 コーディングしてみよう！ ～ スパコンプログラミングを極めるI ～ 〉を、昨年の特集号に引き続き出版する運びとなりました。

本特集号では、スーパーコンピューターを用いた高性能プログラミングの専門家であり、かつ研究開発の第一線でご活躍されている先生方に、スパコン向けのプログラミング技法をノウハウを含めてご紹介いただきます。これからスパコンを利用する入門期にある学生の皆様、コンピューターサイエンスが専門でないベテランの研究者にとっても有益な手引きとなるように、図解を含め容易な解説を心掛けて執筆いただくよう依頼をしました。皆様のお役に立つ手引きとなると信じております。

具体的には、本センターに導入されている HITACHI SR11000/J2 の高速化技術、次期システムである T2K オープンスパコンで必要となる高速化技術、および両スパコンで共通となる高速化技術、の３部に分けて記事を依頼し、編集いたしました。担当者と題目は、以下の通りです。

第１部　HITACHI SR11000/J2 の高速化技術

- （株）日立製作所：数値計算ライブラリーMATRIX/MPP、MATRIX/MPP/SSS、MSL2 のご紹介

- 東京大学 中島研吾：GeoFEM ベンチマークによる Hitachi SR11000/J2 の性能評価
- 埼玉大学 桑島豊、坪谷怜、田村純一、重原孝臣：実対称固有値問題に対する多分割の分割統治法の SR11000 への一実装
- 産業技術総合研究所 片桐秀樹：スーパーコンピュータ SR11000 でのプログラム開発事例：カップルドクラスター法による高分子の電子状態計算

第２部　T2K オープンスパコンの高速化技術

- 東京大学 松葉浩也：オープンスパコンの OS とアーキテクチャの基礎
- 電気通信大学 今村俊幸：キャッシュ性能安定性について
- 筑波大学 高橋大介：FFT におけるキャッシュ最適化方式
- 京都大学 岩下武史：電磁場解析における数値ソルバのキャッシュ最適化チューニング

第３部　共通高速化技術

- 東京大学 黒田久泰：C 言語による OpenMP 入門
- 東京大学 片桐孝洋：C 言語による MPI プログラミング入門

なお、本特集号で取り上げきれなかった話題や、次期導入機種である T2K オープンスパコンの特徴に応じて必要となる事項については、次年度も引き続き同様の特集号を発行することを予定しています。

本特集号の出版に際し、変革期の大学等において諸事多忙な日々を過ごされている先生方に、専門家向けの学術論文とは異なるスタイルでの執筆を快くお引き受け頂きました。末筆になりますが、労をとられた先生方に深くお礼を申し上げます。

2008 年 1 月　次期システムの導入を期待しつつ

第１部
HITACHI SR11000/J2の高速化技術

数値計算ライブラリー

MATRIX/MPP，MATRIX/MPP/SSS，MSL2 のご紹介

（株）日立製作所

１．はじめに

MATRIX/MPP，MATRIX/MPP/SSS，および MSL2 は，SR11000 上で利用可能な数値計算ライブラリーです。行列計算，関数計算，統計計算といった数値計算の分野でよく使われる計算をサポートしています。SR11000 向けにチューニングしており，ライブラリーを使用することで利用者プログラムの高速化ができます。

MATRIX/MPP，MATRIX/MPP/SSS，および MSL2 は，共有メモリー型並列に対応した機能を有しており，MATRIX/MPP および MATRIX/MPP/SSS は，分散メモリー型並列に対応した機能も有します。

ここでは，MATRIX/MPP，MATRIX/MPP/SSS，MSL2 の機能，およびインターフェイスの概要についてご紹介いたします。

２．機能概要

ライブラリー体系を図 2.1 に示します。

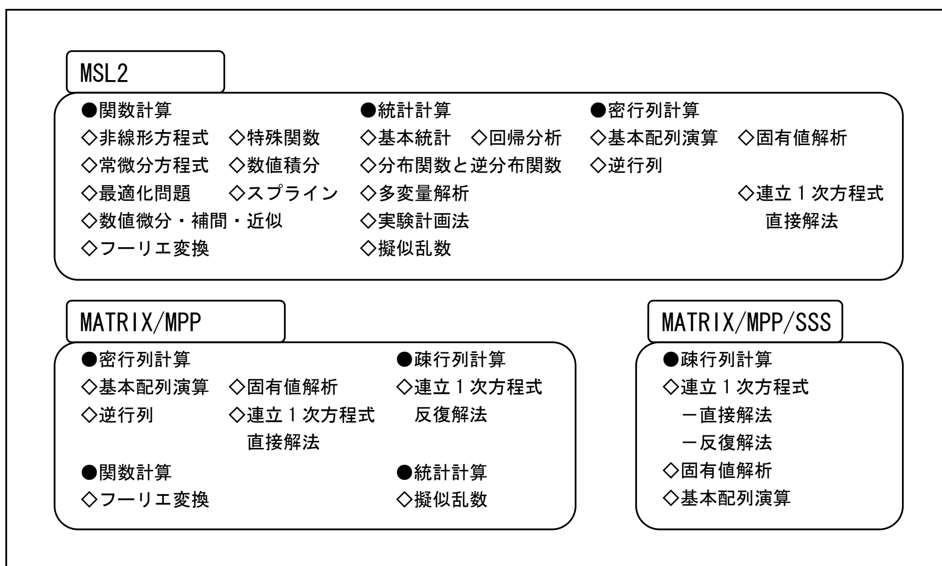


図 2.1 ライブラリー体系

MSL2 は，行列計算，関数計算，統計計算といった数値計算の分野でよく使われる計算をサポートした数値計算副プログラムライブラリーです。

MATRIX/MPP および MATRIX/MPP/SSS は，行列計算副プログラムライブラリーです。MATRIX/MPP は，MSL2 で扱う分野のうち，行列計算分野と，関数計算分野のフーリエ変換機能および関数計算分野の擬似乱数機能を，SR11000 向けにチューニングしたライブラリーです。MATRIX/MPP/SSS は，構造解析等の分野で扱う大次元疎行列に対するライブラリーです。

行列計算副プログラムライブラリーは継続して機能を拡張しています。SR11000 向けライブラリーでは，SR8000 向けライブラリーに対して主に次の機能を新規に追加しました。

(1) MATRIX/MPP
 (a) 基本配列演算
  ・複素数行列の行列乗算
  ・実転置行列の行列乗算
 (b) 固有値・固有ベクトル
  ・エルミート行列に対する一般固有値・固有ベクトル
 (c) 高速フーリエ変換
  ・2 次元および 3 次元の実フーリエ変換（Real ⇔ Complex）

(2) MATRIX/MPP/SSS
 (a) 連立 1 次方程式－直接解法
  ・エルミート疎行列に対するスパースダイレクトソルバー
 また，次の分野を新たに追加しました。
 (b) 連立 1 次方程式－反復解法
 (c) 固有値・固有ベクトル
 (d) 基本配列演算

３．特長

３．１　各ライブラリーの特長

以下に主な特長を示します。

(1) MATRIX/MPP，MATRIX/MPP/SSS
- SR11000 向けにチューニングしたライブラリー
- 分散メモリー型並列に対応した機能をサポート
- 7 基底まで対応した混合基底の高速フーリエ変換機能をサポート
- 非ゼロ要素の構造に着目し，ゼロ要素との演算を抑止することで，スカイライン法に比べ演算量を大幅に削減したスパースダイレクトソルバー機能をサポート
- スパースダイレクトソルバーの前処理において，ゼロ要素が計算過程で非ゼロ要素となる Fill-in 数を少なくするオーダリング機能をサポート

(2) MSL2
- 行列計算，関数計算，統計計算といった数値計算の分野でよく使われる計算をサポート

３．２　SR11000 向け最適化方式

SR11000 のハードウェア性能を引き出すために，ライブラリーでは次のような手法を採用しています。

・並列性の高いアルゴリズムの採用

・キャッシュブロッキング

・スーパースカラの有効利用（ループアンローリングなど）

以下に，並列性の高いアルゴリズムの採用例とキャッシュブロッキングの例を示します。

(1) 並列性の高いアルゴリズムの採用

アルゴリズムを改善し並列性を高めることで，処理を高速にしています。

ここでは一例として，1 次元フーリエ変換を 2 次元フーリエ変換に変形するアルゴリズム[1]について説明します。

フーリエ変換は一般に，

$$C_k = \sum_{j=0}^{N-1} f_j \exp(-\frac{2\pi kj}{N} i) \qquad (k = 0,1,\cdots,N-1) \qquad \cdots\cdots 式1$$

で表されます。

ここで，$N = Nx * Ny$ と因数分解できるとするとき，式 1 における j および k を，

$j = sNx + r$　（ただし，$0 \leqq s < Ny$，$0 \leqq r < Nx$）

$k = pNy + q$　（ただし，$0 \leqq p < Nx$，$0 \leqq q < Ny$）

と表すと式 1 は，次のように表すことができます。

$$C_{pNy+q} = \sum_{r=0}^{Nx-1} \sum_{s=0}^{Ny-1} f_{sNx+r} \exp(-\frac{2\pi (pNy+q)(sNx+r)}{N} i) \qquad \cdots\cdots 式2$$

ここで，三角関数の周期性から，

$$\exp(-\frac{2\pi i}{N}(pNy+q)(sNx+r)) = \exp(-2\pi i(\frac{pr}{Nx} + \frac{qs}{Ny} + \frac{qr}{N}))$$

が成り立つので式 2 は，次のように表すことができます。

$$C_{pNy+q} = \sum_{r=0}^{Nx-1} \sum_{s=0}^{Ny-1} f_{sNx+r} \exp(-\frac{2\pi pr}{Nx} i) \exp(-\frac{2\pi qs}{Ny} i) \exp(-\frac{2\pi qr}{N} i)$$

$$= \sum_{r=0}^{Nx-1} \left[\underbrace{\sum_{s=0}^{Ny-1} (f_{sNx+r} \exp(-\frac{2\pi qs}{Ny} i)) \exp(-\frac{2\pi qr}{N} i)}_{①} \right] \underbrace{\exp(-\frac{2\pi pr}{Nx} i)}_{②} \qquad \cdots 式3$$

式 3 で，N 点の 1 次元フーリエ変換が Nx 組 Ny 点の 2 次元フーリエ変換と Ny 組 Nx 点の 2 次元フーリエ変換に分離されます。分離されることで，①が r に依存しないため Nx に対して並列化可能になり，②が s に依存しないため Ny に対して並列化可能になります。

次に一例として，スカイライン法での手法を図 3.1 に示します。

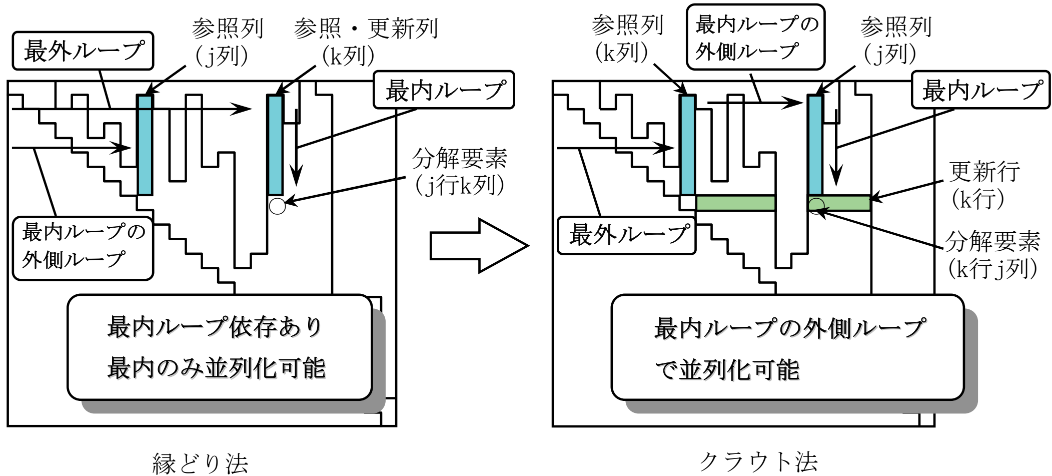


図 3.1 ライブラリーでの並列化手法

図 3.1 に示す縁どり法[2]では，1～(k-1)列を参照し，参照・更新列(k 列)を更新します。図は，参照列(j 列)を参照し，分解要素(j 行 k 列の要素)の更新を表しています。参照列が次の参照列(j+1 列)となるとき，前の処理で更新した分解要素(j 行 k 列の要素)を参照するため，1～(k-1)列の最内ループの外側ループにまたがり更新と参照の依存関係があります。そのため最内ループのみが並列化可能です。

上記の縁どり法での最内ループの外側ループ繰り返し依存の関係を解消するために，ライブラリーではクラウト法[2]を採用しています。図 3.1 に示すクラウト法では，k+1～N 列を参照し，更新行(k 行)を更新します。参照列(j 列)と更新行(k 行)には依存がないため，最内ループの外側ループで並列化可能となります。

(2) キャッシュブロッキング

キャッシュメモリーは実メモリーに比べアクセス速度が高速のため，キャッシュメモリーに入ったデーターを繰り返し利用できれば，実メモリーのデーターを用いるより計算処理が高速になります。キャッシュメモリーは容量が少ないため，計算に必要な全てのデーターを持つことはできません。そのため行列データーを，キャッシュメモリーに入るよう一定の大きさにブロッキングします。

ライブラリーでは，SR11000 に適したキャッシュブロッキングを採用することで，高速な演算を行っています。行列乗算 C=A×B の計算処理におけるブロッキング例を図 3.2 に示します。

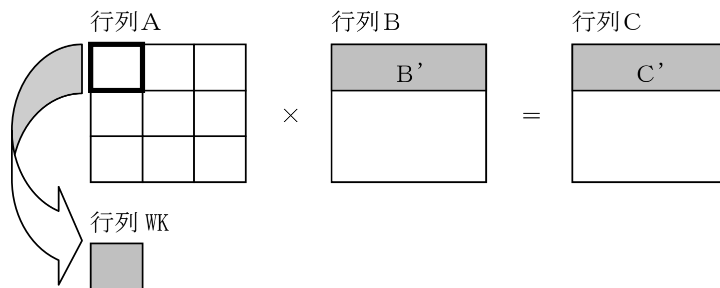


図 3.2 行列のブロッキング例

行列 WK は，行列 A の 1 ブロックを転置コピーしたものです。行列乗算処理では，行列 WK と行列 B の B'部分を参照して行列乗算を行い，行列 C の C'部分に格納します。このとき行列 WK はキャッシュメモリーに入った状態で繰り返し利用されます。

一般固有値の標準化変換における前進消去演算でのブロッキング例を図 3.3 に示します。

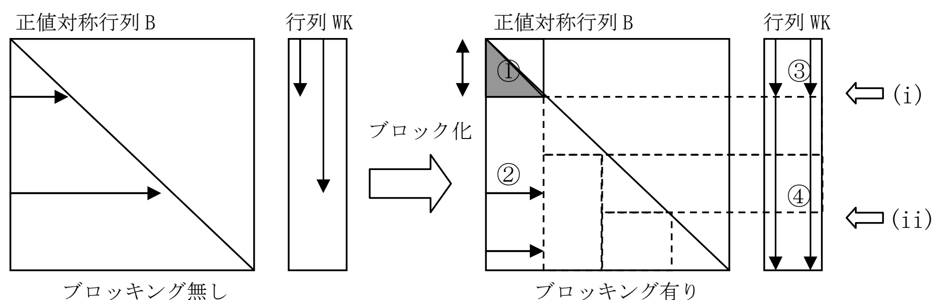


図 3.3 一般固有値でのブロッキング例

図 3.3 の行列 WK は，圧縮 1 次元配列で格納された実対称行列 A を，連続にアクセスするためにコピーした 2 次元配列です。右のブロッキング有りの例では，次のように処理します。

(i) 正値対称行列 B の対角ブロック①と行列 WK のブロック③を参照して，行列 WK のブロック③を更新

(ii) 次に，更新が終わった行列 WK のブロック③と正値対称行列のブロック②を参照して，行列 WK のブロック④の消去計算を実施

上記のように処理することで，参照する行列データーを常にキャッシュメモリーに入れておき，高速な前進消去演算を実現しています。

４．インターフェイス仕様

４．１　並列化の概要

並列化にはノード内並列化とノード間並列化があり，それぞれに対応したインターフェイスがあります。概要を図 4.1 に示します。

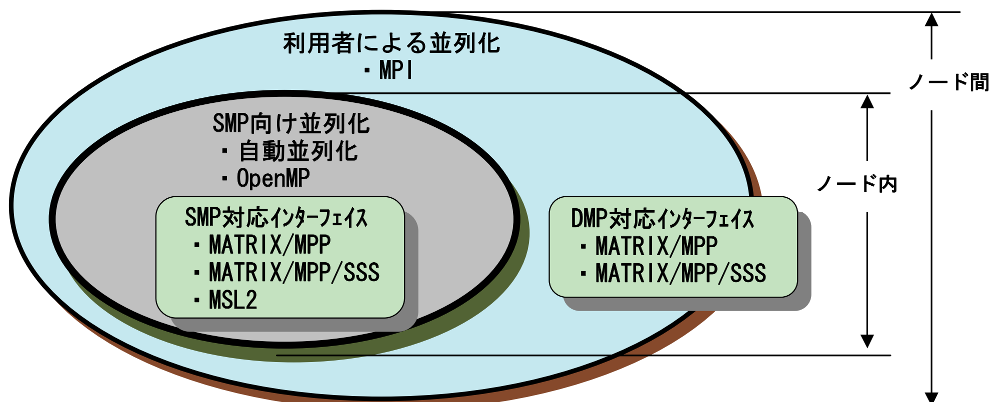


図 4.1 並列化の概要

SR11000 のノード内並列化は共有メモリー型並列（SMP:Shared Memory Parallel）で，ノード間並列化は分散メモリー型並列（DMP:Distributed Memory Parallel）です。

ノード内並列化は自動および OpenMP で並列化します。ノード間並列化は利用者プログラムで MPI(Message Passing Interface)を用いて並列化します。

MATRIX/MPP，MATRIX/MPP/SSS，および MSL2 は，ノード内並列化への対応として SMP 対応インターフェイスを有しています。

MATRIX/MPP および MATRIX/MPP/SSS は，SMP 対応インターフェイスに加え，ノード間並列化への対応として DMP 対応インターフェイスを有しています。

４．２　利用形態

ライブラリーでは利用形態に応じて，SMP 対応インターフェイスとして逐次処理用インターフェイスを，DMP 対応インターフェイスとして並列処理用インターフェイスを提供しています。次に各インターフェイスの概要を示します。

(1) 逐次処理用インターフェイス

1 ノード内の複数プロセッサーを使用して実行する場合に利用します。逐次に実行する利用者プログラム，ノード内およびノード間で並列に実行する利用者プログラムから利用可能です。ライブラリー内部ではノード間通信を行いません。

逐次処理の利用形態を図 4.2 に示します。

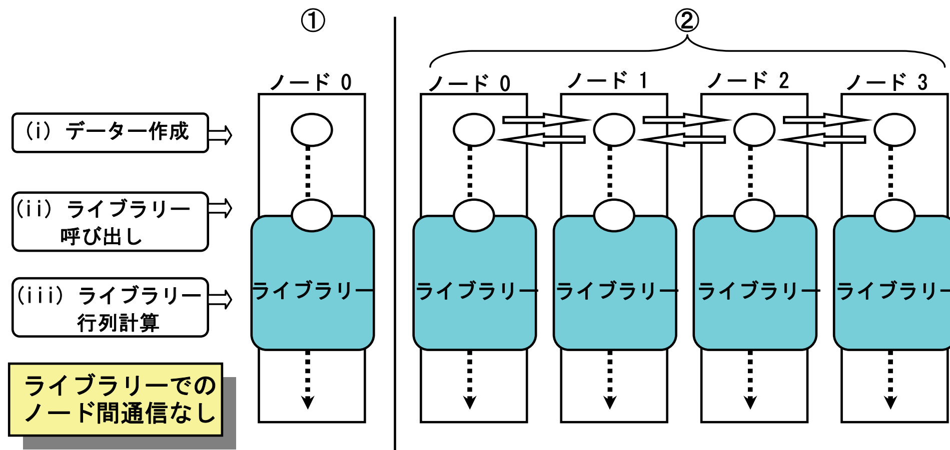


図 4.2 逐次処理の利用形態

図 4.2 において，(i)は利用者プログラムによるデーター作成，(ii)は利用者プログラムからのライブラリー呼び出し，(iii)はライブラリー内での行列計算を示します。

①利用者プログラムを 1 ノードで実行

利用者プログラム，ライブラリーともに，ノード間通信なし。

②利用者プログラムを 4 ノード間で並列に実行（逐次処理用インターフェイスを使用）

(i)　データー作成は，利用者プログラムがノード間でデーター通信

(ii) 各ノードでライブラリーを呼び出す

(iii)ライブラリーはそれぞれのノード内で動作し，ライブラリーの内部ではノード間通信なし

(2) 並列処理用インターフェイス

複数ノードを使用して実行する場合に利用します。ノード間で並列に実行する利用者プログラムから利用可能です。計算の対象となるデーターを分散メモリー上に分散して配置し利用するインターフェイスで，ライブラリー内部でノード間のデーター通信を行います。

並列処理の利用形態を図 4.3 に示します。

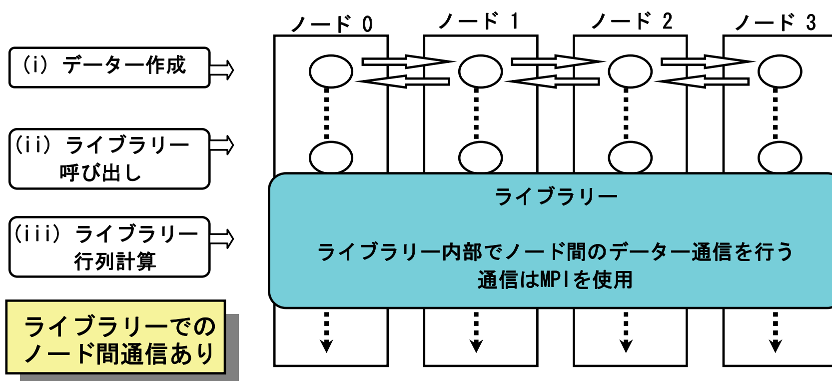


図 4.3 並列処理の利用形態

図 4.3 は，利用者プログラムを 4 ノード間で並列に実行し，並列処理用インターフェイスを使用する例です。

(i)　データー作成は，利用者プログラムがノード間でデーター通信

(ii) 各ノードでライブラリーを呼び出す

(iii)ライブラリーの内部で MPI を用いてデーター通信

４．３　データーの与え方

ライブラリーでは利用するインターフェイスにより，行列種別ごとにデーターの与え方が異なります。データーの与え方を次に示します。

(1) 逐次処理用インターフェイス

逐次処理用インターフェイスのデーターの与え方は，表 4.1 のように分類されます。

表 4.1 逐次処理用インターフェイスのデーター格納配列

行列種別	データー格納配列
密非対称行列	1 次元配列
密対称行列	圧縮 1 次元配列
帯行列	圧縮 2 次元配列
疎行列	圧縮 1 次元配列 圧縮 2 次元配列
スカイライン行列	圧縮 1 次元配列
ベクトル	1 次元配列

(2) 並列処理用インターフェイス

並列処理用インターフェイスでは複数ノードで実行するため，各ノードにデーターを分割して与えます。データー分割方式を表 4.2 に示します。

表 4.2 並列処理用インターフェイスのデーター分割方式

行列種別	データー分割方式
密非対称行列	Scattered Square 分割 ブロックサイクリック列分割
密対称行列	Scattered Square 分割 ブロックサイクリック列分割
帯行列	Scattered Square 分割
疎行列	差分法領域分割による配列 ブロック行分割による圧縮型 2 次元配列
スカイライン行列	Scattered Square 分割 ブロックサイクリック列分割
ベクトル	Scattered Square 分割

４．４　並列処理用インターフェイスのデーター割り当てと動作例

並列処理用インターフェイスではノード間でデーター通信を行います。並列処理用インターフェイスでのデーター分割と動作の例として，Scattered Square 分割とブロックサイクリック列分割の概要を次に説明します。

(1) Scattered Square 分割

(a) データー分割

ブロック形式 Scattered Square 分割の，16 ノード使用時の構造例を図 4.4 に示します。

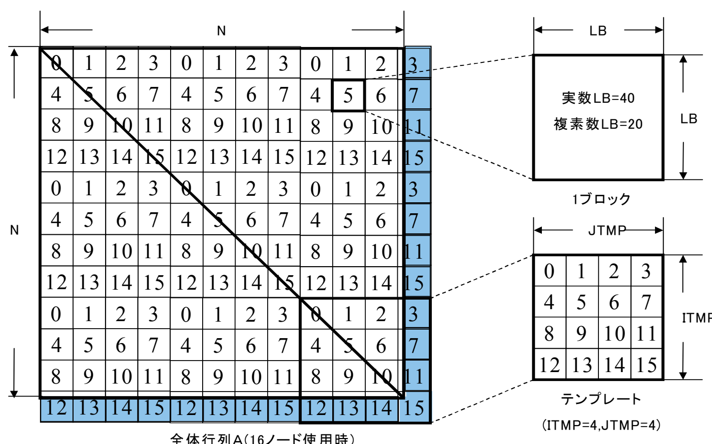


図 4.4 ブロック形式 Scattered Square 分割の構造例

ブロック形式 Scattered Square 分割では，全体行列をブロックの集まりとして捉えます。1 ブロックの LB（ブロックサイズ）は，実数では 40，複素数では 20 です。

図 4.4 では，図中の数字は，データーを割り当てるノード番号です。行列データーの分配では，使用するノード台数によってテンプレートを決定し，そのテンプレートにしたがって行列データーを分配します。

テンプレートの形状である ITMP と JTMP の大きさは，使用するノード台数を NPU とすると，NPU=ITMP*JTMP の関係です。このため，ITMP と JTMP の大きさは，計算で使用するノード台数によって決まります。この例ではノード台数が 16 であることから，ITMP=4, JTMP=4 となります。

図 4.4 のように分割したデーターは，部分行列に格納してライブラリーに引き渡します。部分行列への格納方式を図 4.5 に示します。

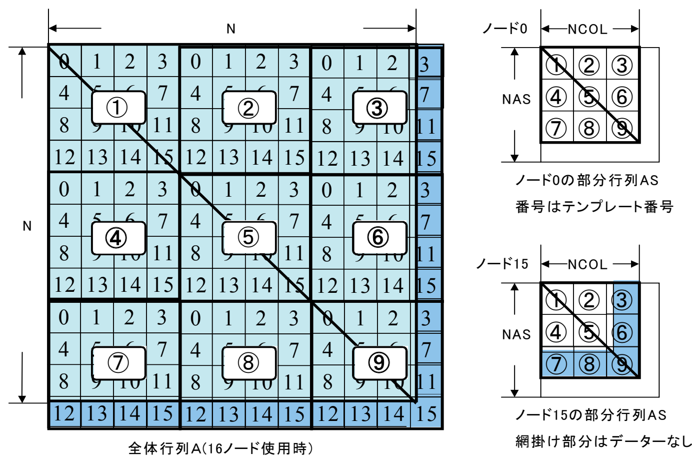


図 4.5 部分行列への格納方式

図 4.5 では，右の部分配列 AS 内の数字は，左の全体行列 A のテンプレート番号です。

網掛け部分は，データーの無い部分であり，初期化の必要はありませんが，領域は確保しておく必要があります。

(b) 動作概要

4 ノード使用時の，行列乗算の動作概要例を図 4.6 に示します。

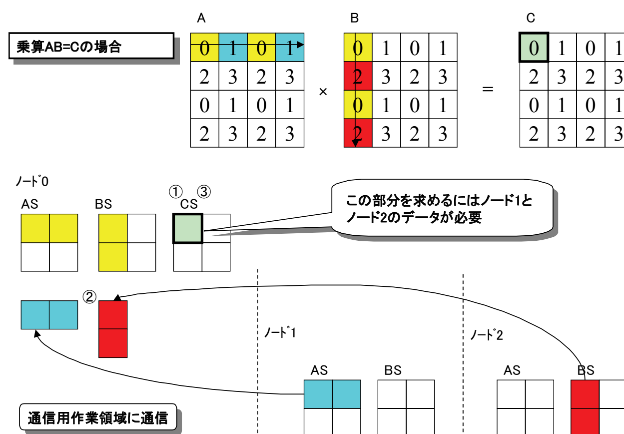


図 4.6 行列乗算の動作概要例

図 4.6 のノード 0 の動作は次のとおりです。

① 自ノードに割り当てられた AS および BS を参照し CS を計算

② 通信用作業領域を用いて，CS を求める際に必要となるノード 1 の AS とノード 2 の BS をノード 0 で受信

③ 受信したデーターを参照し CS を計算

(2) ブロックサイクリック列分割

(a) データー分割

ブロック形式サイクリック列分割の，4 ノード使用時の構造例を図 4.7 に示します。

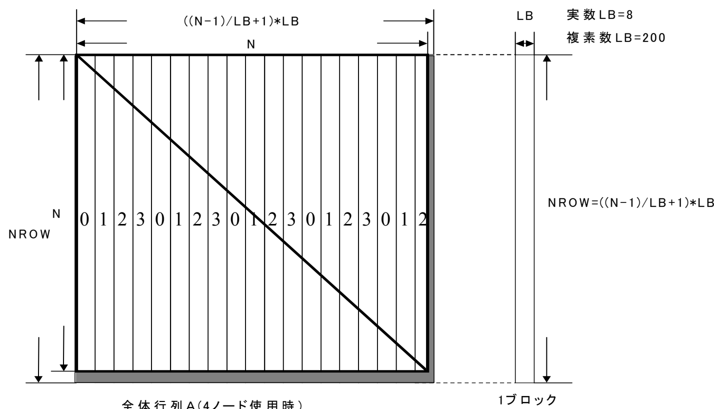


図 4.7 ブロック形式サイクリック列分割の構造例

ブロック形式サイクリック列分割では，全体行列のデーターをブロック単位に分割して，ブロックの集まりを部分行列に格納してライブラリーに引き渡します。図 4.7 の図中の数字は，データーを割り当てるノード番号です。1 ブロックの LB（ブロックサイズ）は，実数では 8，複素数では 200 です。

部分行列への格納方式を図 4.8 に示します。

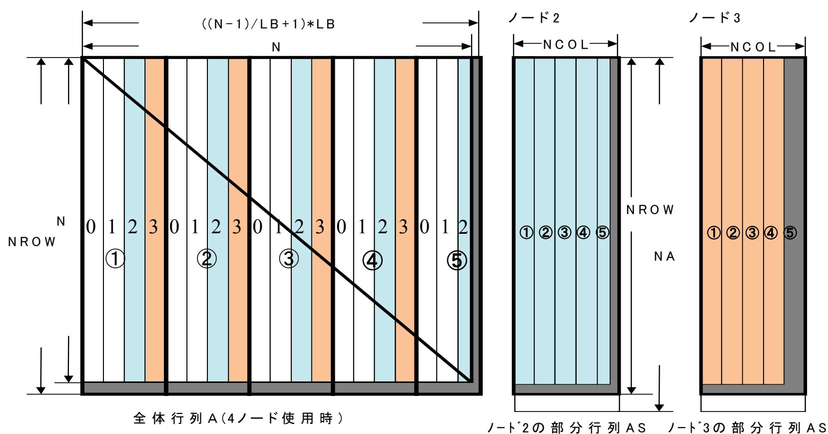


図 4.8 部分行列への格納方式

図 4.8 では，右の部分配列 AS 内の数字はテンプレート番号で，左の全体行列 A のテンプレート番号に対応した当該ノード用のデーターを集めた形となります。

(b) 動作概要

4 ノード使用時の連立 1 次方程式の LU 分解における動作概要例を図 4.9 に示します。

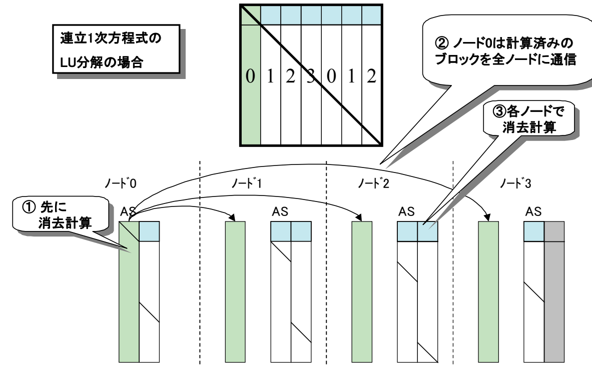


図 4.9 連立 1 次方程式の LU 分解における動作概要例

図 4.9 では，次のとおり動作します。

① ノード 0 で軸列ブロックの消去計算

② 消去計算した軸列ブロックをノード 0 から各ノードに対して送信

③ 各ノードではノード 0 から受け取った軸列ブロックを参照して消去計算

５．性能測定結果

ライブラリーはノード内並列化およびノード間並列化に対応しているため，使用するプロセッサー数およびノード数により計算時間が短縮でき，プログラムの高速化が図れます。MATRIX/MPP での性能評価結果を次に示します。

(1) 3 次元複素フーリエ変換(HZFT7M)

3 次元複素フーリエ変換で変換点数が 512×512×512 の場合の性能評価結果を図 5.1 に示します。

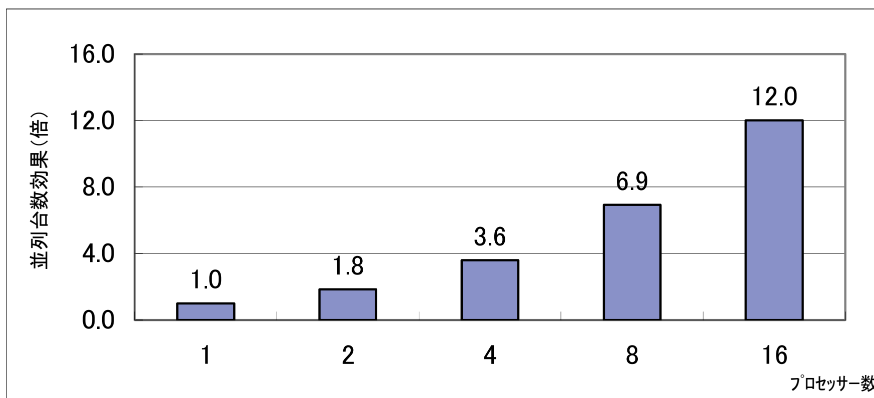


図 5.1 3 次元複素フーリエ変換(HZFT7M) 性能比較

図 5.1 は，横軸にノード内並列化による使用プロセッサー数を，縦軸に 1 プロセッサーで計算した場合の計算時間を 1 とし，1 プロセッサーで計算した場合の計算時間を複数プロセッサーで計算した場合の計算時間で割ったプロセッサー数による並列台数効果を示した図です。

1 プロセッサーに比べて，2 プロセッサーでは 1.8 倍，4 プロセッサーでは 3.6 倍，8 プロセッサーでは 6.9 倍，16 プロセッサーでは 12.0 倍となります。

(2) 連立 1 次方程式分散版(HDLPPMDP)

30000 次元の実対称行列を係数行列とする連立 1 次方程式の解を求めた場合の性能評価結果を図 5.2 に示します。

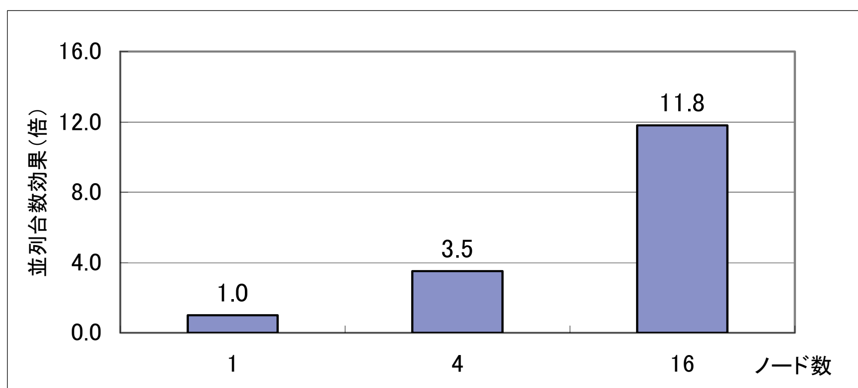


図 5.2 連立 1 次方程式（実対称）性能比較

図 5.2 は，横軸にノード間並列化による使用ノード数を，縦軸に 1 ノードで計算した場合の計算時間を 1 とし，1 ノードで計算した場合の計算時間を複数ノードで計算した場合の計算時間で割ったノード数による並列台数効果を示した図です。

1 ノードに比べて，4 ノードで計算を行うと 3.5 倍，16 ノードで計算を行うと 11.8 倍になります。

６．まとめ

以上で示しましたとおり，ライブラリーのノード内並列化およびノード間並列化に対応したインターフェイスを使用することで利用者プログラムを簡略化できるとともに，並列性の高いアルゴリズムやキャッシュブロッキング等を採用して SR11000 向けにチューニングしたライブラリーを使用することで容易に並列台数効果を得ることができます。

７．ライブラリー使用時の注意点

ライブラリーは，利用者プログラムで注意が必要な点や，指定値によって性能に差異が出る場合などがあります。ここでは例として，並列処理用インターフェイス使用時の注意点と，整合寸法指定値による性能差について示します。

(1) 並列処理用インターフェイス使用時の注意点

ライブラリーは内部で MPI を使用してノード間でデーター通信を行います。ライブラリーの初期処理用副プログラム(HMATINIT または HCOMINIT)および終了処理用副プログラム(HMATEXIT または HMATFINL）と，MPI の初期処理（MPI_INIT）および MPI の終了処理（MPI_FINALIZE）との関係を表 7.1 に示します。

表 7.1 ライブラリーの初期処理/終了処理と MPI_INIT/MPI_FINALIZE の関係

副プログラム	名称	利用者プログラムでの MPI_INIT の発行	動作	注意事項
初期処理用副プログラム	HMATINIT または HCOMINIT	発行済	MPI_INIT は発行しない	－
		発行していない	MPI_INIT を発行する	初期処理用副プログラムで MPI_INIT を発行するため，以降は利用者プログラムでも MPI 関数を使用できる
終了処理用副プログラム	HMATFINL	発行済	MPI_FINALIZE は発行しない	利用者プログラムで MPI_INIT を発行した場合，MPI_FINALIZE も利用者プログラムで発行する必要がある
		発行していない	MPI_FINALIZE を発行する	終了処理用副プログラムで MPI_FINALIZE を発行するため，以降は利用者プログラムでも MPI 関数は使用できない
	HMATEXIT	－	MPI_FINALIZE は発行しない	利用者プログラムでの MPI_INIT の発行有無に関係しない

並列処理用インターフェイスの副プログラムを実行する場合は，副プログラムの実行前に初期処理用副プログラムで環境設定をしなければなりません。

また副プログラムの実行終了後には，終了処理用副プログラムの実行が必要です。終了処理用副プログラムにより，初期処理で設定した環境を解放します。

HMATFINL では，それ以前に MPI_INIT が発行されているかを判別し，判別結果により必要に応じて MPI_INIT および MPI_FINALIZE を発行します。

利用者プログラムで MPI_INIT を発行している場合は，HMATFINL では MPI_FINALIZE を発行しません。そのため，MPI 通信関数を使用しなくなる時点で，利用者プログラム内で MPI_FINALIZE の発行が必要です。

MPI_FINALIZE を発行すると，それ以降は並列処理用副プログラムだけでなく利用者プログラム中でも MPI 関数が使用できなくなりますので，注意が必要です。

(2) 整合寸法指定値による性能差

配列の大きさによっては,キャッシュラインのスラッシング[3]の影響で性能に差異が出ます。そのため，整合寸法の大きさによって性能に差異が出ることがあります。

3次元複素フーリエ変換での入出力データー形式の例を図7.1に示します。

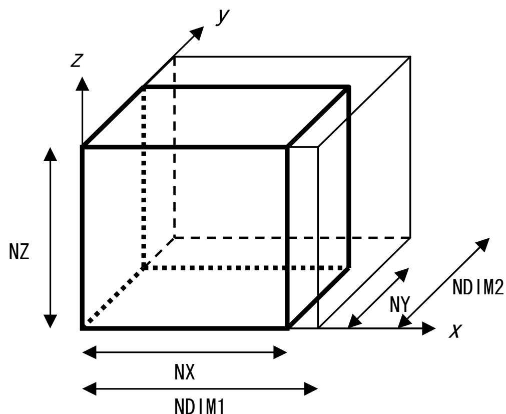


図7.1 入出力データー形式の例

図7.1に示すとおり，実際に変換を行う3次元配列の範囲はNX×NY×NZですが，NXに対してNDIM1(NDIM1≧NX),NYに対してNDIM2(NDIM2≧NY)を使用し,確保する配列の大きさはNDIM1×NDIM2×NZで指定します。

NX=NY=NZ=512の場合の性能比較を図7.2に示します。

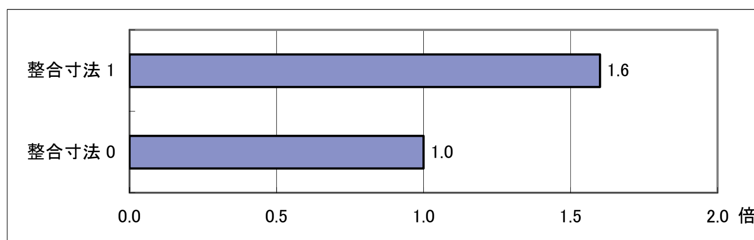


図7.2 性能比較

図7.2は，整合寸法0がNDIM1=NX,NDIM2=NY，整合寸法1がNDIM1=NX+1，NDIM2=NY+1で，縦軸に整合寸法の違いを，横軸に整合寸法0の場合の計算時間を1とし，整合寸法0の場合の計算時間を異なる整合寸法の場合の計算時間で割った整合寸法による性能改善効果を示した図です。整合寸法を調整することで，整合寸法0に比べて整合寸法1では約1.6倍となります。

2基底の3次元複素フーリエ変換では，変換点数が2ベキとなるためにキャッシュラインのスラッシングが発生して性能が劣化する可能性が高くなります。そのため，整合寸法に奇数値を指定することでキャッシュラインのスラッシングを大幅に低減することができます。

７．参考文献

[1] 森　正武，名取　亮，鳥居　達生，岩浪講座　情報科学－18　数値計算，岩波書店 (1982)，pp.160-162

[2] 小国　力，村田　健郎，三好　俊郎，Dongarra,J.J.，長谷川　秀彦，行列計算ソフトウェア，丸善(1991)，pp.105-111

[3] Kevin Dowd 著，久良知　真知子訳，ハイ・パフォーマンス・コンピューティング，オーム社(1992)，pp.53-60

[4] (株)日立製作所，SR11000 行列計算副プログラムライブラリ MATRIX/MPP (3000-3-C87)，(2007)

[5] (株)日立製作所，SR11000 行列計算副プログラムライブラリー疎行列解法 MATRIX/MPP/SSS (3000-3-C88)，(2007)

[6] (株)日立製作所，SR11000 数値計算副プログラムライブラリ MSL2 行列計算 (3000-3-C83)，(2004)

[7] (株)日立製作所，SR11000 数値計算副プログラムライブラリ MSL2 関数計算 (3000-3-C84)，(2004)

[8] (株)日立製作所，SR11000 数値計算副プログラムライブラリ MSL2 統計計算 (3000-3-C85)，(2004)

[9] (株)日立製作所，SR11000 数値計算副プログラムライブラリ MSL2 操作 (3000-3-C86)，(2004)

GeoFEM ベンチマークによる Hitachi SR11000/J2 の性能評価

中 島 研 吾

東京大学大学院理学系研究科地球惑星科学専攻

1. はじめに：Flat MPI vs. Hybrid

近年のハードウェア技術の発展によって，単一のメモリに多くのプロセッサ（Processing Element : PE，最近は「プロセッサコア」，または単に「コア」と呼ばれることも多い）が効率的にアクセスすることが可能となり，SMP（Symmetric Multiprocessors）のクラスタによる並列計算機が数多く開発されている。米国エネルギー省の ASC 計画（Advanced Simulation & Computing）[1]，「地球シミュレータ」[2] 等のテラフロップス級の超並列計算機は，すべてこのアーキテクチャによっている。

このような計算機において最大の性能を発揮するために，多段階ハイブリッド手法（multi-level hybrid，以下「Hybrid」）に基づく並列プログラミングモデルがしばしば使用されている（図 1 参照）。この手法はディレクティブによる「fine-grain parallelism」と，メッセージパッシングによる「coarse-grain parallelism」の融合であり，一般的には OpenMP[3] と MPI（Message Passing Interface）[4]を組み合わせたプログラミングスタイルである。各共有メモリユニット（SMP ノード）に OpenMP，SMP ノード間の通信に MPI が適用される。SMP クラスタ型アーキテクチャにおけるもう一つのアプローチは，個々のプロセッサを独立に扱う単段階の「Flat MPI」である（図 1）。Hybrid と Flat MPI の優劣は，さまざまなハードウェア性能諸元（コア単体性能，通信バンド幅，メモリバンド幅等）とそのバランス，アプリケーションの特性，問題サイズに依存するものである〔1〕。近年，マルチコアプロセッサの普及により，Hybrid と Flat MPI の比較については，ふたたび注目される傾向にある。

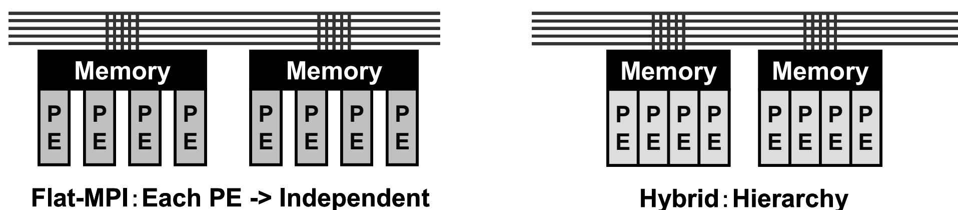


図 1　SMP クラスタにおける並列プログラミングモデル

筆者らは，固体地球シミュレーション用並列有限要素法（Finite Element Method : FEM）プラットフォーム「GeoFEM」[5]の開発の一環として，非構造格子向け前処理付き反復法による線形ソルバーを，「地球シミュレータ（ES）」上で最適化し，Hybrid および Flat MPI の両プログラミングモデルを様々なアプリケーションに適用し，検討を実施してきた〔2,3,4,5,6〕。

---

[1] https://asc.llnl.gov/
[2] http://www.es.jamstec.go.jp/
[3] http://www.openmp.org/
[4] http://www-unix.mcs.anl.gov/mpi/index.htm
[5] http://geofem.tokyo.rist.or.jp/

本学情報基盤センターに導入されている Hitachi SR11000/J2 は，IBM POWER5+プロセッサ（2.30 GHz）に基づく SMP クラスタ型の並列計算機である[6]。本研究では，並列有限要素法コード GeoFEM に基づく様々なベンチマークを，各 SMP ノードの 8 個または 16 個のコアを使って実施し，性能を評価した。

以下，第 2 章では並列有限要素法の概要について「GeoFEM」を例にとって説明する。第 3 章では GeoFEM ベンチマークの概要を紹介し，第 4 章（1 ノード）および第 5 章（複数ノード）で性能評価結果について説明し，第 6 章で本稿をまとめる。第 3 章では，文献〔7〕で紹介した，Hitachi SR11000/J1（IBM POWER5 プロセッサ，1.90 GHz）による結果との比較も実施した。

2. 並列有限要素法について

（1）概　要

有限要素法（FEM）は偏微分方程式の数値解法として，科学技術シミュレーションでは広く使用されている。近年はより詳細な計算のために，並列計算機を使用した大規模シミュレーションが盛んに実施されている。FEM の処理は，線形，非線形いずれの場合も，最終的には疎な全体剛性マトリクスを係数マトリクスとする大規模連立一次方程式を解くことに帰着される。このような方程式の係数マトリクスは，線形化した支配方程式に対して，要素単位で重みつき残差法あるいは変分法を適用して得られる要素剛性マトリクスを足し合わせて得られる。FEM の計算のほとんどの部分は：

- 係数マトリクス生成
- 線形ソルバによる大規模連立一次方程式求解

に費やされる。有限要素法は差分法などと比較して並列化が困難であると考えられてきた。間接データ参照があるため，コアあたりの計算効率は差分法と比較して低いが，有限要素法では基本的に要素単位の局所的な処理が中心となるため，並列化には適している。特に，係数マトリクス生成に関しては，要素単位での実行が可能なため，並列化は容易であり，領域間の通信無しに実行することが可能である。すなわち，1CPU の PC 向けに開発されたプログラムをそのまま並列計算機上で実行することが可能である。

科学技術シミュレーションいおける連立一次方程式の解法としてはガウスの消去法などの直接法（Direct Method）が広く使用されてきたが，問題規模と計算量，必要記憶容量の非線形性のため大規模シミュレーションには適していない。大規模シミュレーション，並列計算に適した手法として共役勾配法（Conjugate Gradient Method：CG）などの Krylov 型反復法（Krylov Iterative Method）が利用されている。「GeoFEM」では Krylov 型反復法を使用している。

反復法の収束特性は係数マトリクスの固有値分布に依存するため，実用的な問題に適用するためには前処理（Preconditioning）を施し，固有値分布を変えたマトリクスを解く手法が一般的である。反復法の前処理手法としては不完全 LU 分解（Incomplete LU Factorization：ILU）あるいは対称行列向けの不完全コレスキー分解（Incomplete Cholesky Factorization：IC）などがよく使用される〔8〕。IC／ILU 前処理では，前処理行列を係数行列とする方程式を前進後退代入によって解く必要がある。IC／ILU 前処理，前進後退代入によるプロセスでは大域的な依存性が

[6] http://www.cc.u-tokyo.ac.jp/service/intro/index.html

あるため並列化，ベクトル化が困難とされてきた。筆者らは，「GeoFEM」プロジェクト以来，これらの問題の解決に取り組んできた〔2,3,4,5,6〕。ベクトル計算機，並列計算機で性能を発揮するためには：

① 局所的処理と依存性の排除
② 連続メモリアクセス
③ 充分に長い（最内）ループ長

という条件を満たしていなければならない〔2,3〕。「GeoFEM」では：

- マルチカラーあるいは Reverse Cuthill-Mckee（RCM）法によるオーダリング〔8,9〕
- ブロックヤコビ法のアイディアに基づいた局所 IC／ILU 前処理法
- 最内ループ長を大きくするための係数格納法
- GeoFEM の並列分散データ構造

等によって非常に高いベクトルおよび並列性能が得られている。これらについては既にいくつかの文献で既に紹介しているので，興味のある読者は文献〔2,3〕などを参考にされたい。また OpenMP による並列化については「スーパーコンピューティングニュース」にも連載記事[7]を執筆中であるので，本格的にプログラミングに取り組みたい場合はそちらを参照されたい。

（2）並列分散データ構造と通信

並列計算で扱うデータのサイズ（メッシュ数）は非常に大きいため，全体領域を一括して取り扱うことは困難で，全体領域のデータを部分領域（局所データ）に分割する必要がある。それぞれの領域（domain，partition）は，Hybrid 並列プログラミングモデルの場合は各 SMP ノード, Flat MPI の場合は各コアに割り当てられる。並列 FEM の計算においては図 2 に示すように，領域分割された局所データを各部分領域に関して独立に読み込み，係数マトリクスを生成することが可能であり，領域間の通信が発生する可能性があるのは線形ソルバの部分のみである。この特性を最大限利用し，適切なデータ構造を設定，並列計算に適した反復法を採用することによって，100%に近い並列化効率を達成することも可能である。

FEM に代表される非構造格子（Unstructured Grids）を使用したアプリケーションにおいて，適切な分散データ（局所データ）構造を決定することは，並列計算を効率的に実施する上で重要である。GeoFEM の局所メッシュデータは図 3 に示すような節点ベース（node-based）の領域分割に拠っており，領域間オーバーラップ要素を含んでいる。このようなデータ構造は係数マトリクスを各部分領域で独立に生成し，更に線形ソルバとして前処理つき反復法を適用する場合に有効である〔2,3〕。

FEM において，速度，温度，変位など，線形方程式の解となるような変数は節点において定義される。したがって，並列計算における効率という観点からは領域間の節点数が均等であることが望ましい。これが節点ベースの領域分割法を採用した理由である。節点ベースの領域分

[7] 中島研吾「OpenMP によるプログラミング入門（I）～（III)」スーパーコンピューティングニュース Vol.9 No.5（2007 年 9 月号）より 3 回にわたって連載中

割を使用した場合，剛性マトリクス生成に代表されるような要素単位の処理を各領域において局所的に実施するためには，領域間のオーバーラップ要素が必要である。図 4 はこのようなオーバーラップ要素の例を示したものである。ここで，各節点の色（白，黒，灰色）は所属領域を表す。灰色に塗られた要素は複数の領域によって共有されており，各領域における各節点に対する剛性マトリクスの足し込みなどの処理を，並列に実施するためにはこれらオーバーラップ要素の情報が必要である。

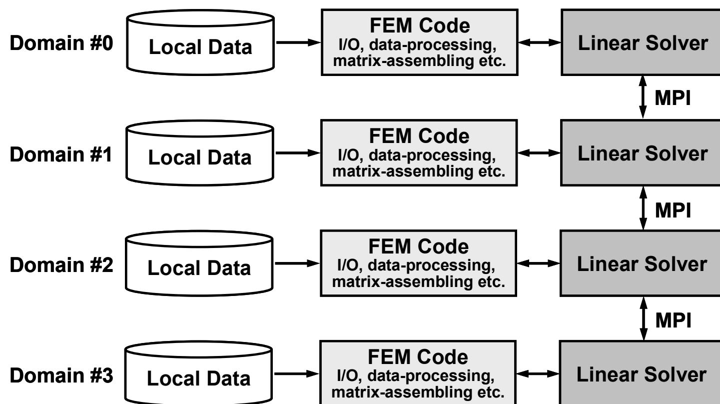


図 2　GeoFEM における分散データ処理，有限要素法処理手順（4 領域の場合）

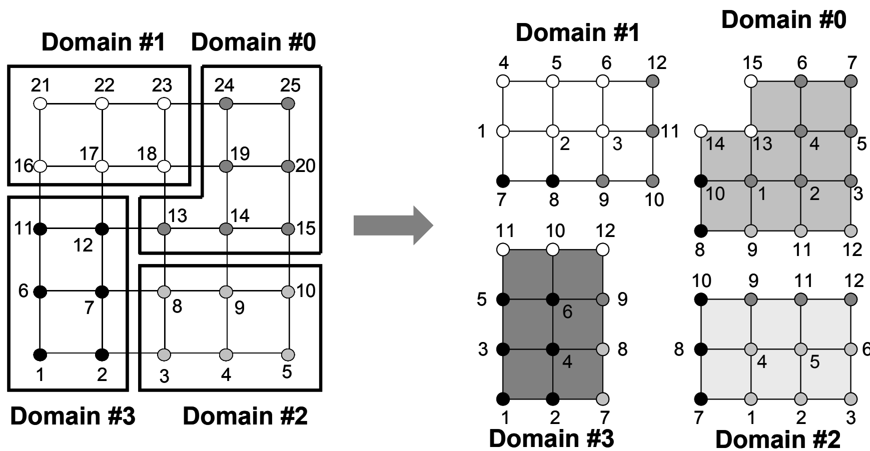


図 3　「節点ベース」領域分割の例（4 領域の場合）〔2,3〕

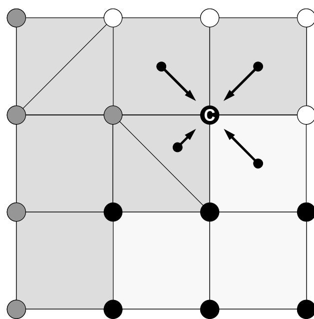


図 4　節点 C 周囲の要素単位オペレーションの概要，灰色に塗られた要素は領域間のオーバーラップ要素である（節点の色（白，黒，灰色）は所属領域を表す）

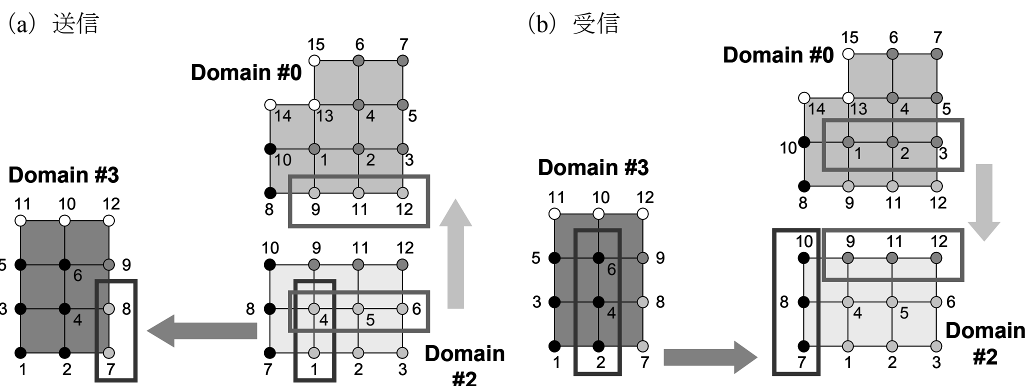


図 5　GeoFEM の局所データ構造と領域間通信

GeoFEM では領域間の通信の記述には MPI を使用している。差分法などに使用されている構造格子（Structured Grids）に関しては MPI 固有の領域間通信用のサブルーチンが準備されているが，非構造格子では，プログラム開発者が独自にデータ構造と領域間通信を設計しなくてはならない。

GeoFEM において，各領域は以下の情報を含んでいる：

- 各領域に割り当てられた節点
- 各領域に割り当てられた節点を含む要素
- 他の領域に割り当てられているが上記の要素に含まれている節点
- 領域間の通信テーブル（送信，受信用）
- 節点グループ，要素グループ，面グループ
- 材料物性

節点は，通信という観点から以下の 3 種類に分類される：

- 内点（Internal Nodes）：各領域に割り当てられた節点
- 外点（External Nodes）：他領域に属しているが，各領域の要素に含まれている節点
- 境界点（Boundary Nodes）：他領域の外点となっている節点

図 3 と図 5 における Domain #2 において，節点は以下のように分類される：

- 内点　　{1, 2, 3, 4, 5, 6}
- 外点　　{7, 8, 9,10,11,12}
- 境界点　{1, 4, 5, 6}

局所データには領域間の「通信テーブル」の情報も含まれる。境界点における値は隣接領域へ「送信（send）」され，送信先では外点として「受信（receive）」される。図 3，図 4 に示す局所データ構造と図 5 に示す領域間通信によって非常に高い並列性能が達成されている〔2,3,4,5,6〕。

図 6 は GeoFEM における通信処理部分のサブルーチンである。ここで，EXPORT_INDEX，EXPORT_NODE という配列が送信用の通信テーブルであり，IMPORT_INDEX，IMPORT_NODE という配列が受信用の通信テーブルである。以下のような手順で実施される。

① 従属変数ベクトル X の中身を転送（送信）ベクトル WS に代入する。neib 番目の隣接領域に対して：

```
istart= EXPORT_INDEX(neib-1) + 1
iend  = EXPORT_INDEX(neib)
```

とすると WS(istart)から WS(iend)の連続したデータが neib 番目の部分領域に転送される：

```
do neib= 1, NEIBPETOT
  istart= EXPORT_INDEX(neib-1)
  inum  = EXPORT_INDEX(neib  ) - istart
  do k= istart+1, istart+inum
    WS(k)= X(EXPORT_NODE(k))
  enddo
  call MPI_ISEND (WS(istart+1), inum, etc.)
enddo
```

② ①とは逆の手順で転送（受信）ベクトル WR を受け取る。neib 番目の隣接領域に対して：

```
istart= IMPORT_INDEX(neib-1) + 1
iend  = IMPORT_INDEX(neib)
```

とすると，WR(istart)から WR(iend)までの連続したデータが neib 番目の部分領域から転送される。

```
do neib= 1, NEIBPETOT
  istart= IMPORT_INDEX(neib-1)
  inum  = IMPORT_INDEX(neib  ) - istart
  call MPI_IRECV (WR(istart+1), inum, etc.)
enddo
```

③ 従属変数ベクトル X に転送（受信）ベクトル WR の中身を代入する。

```
do neib= 1, NEIBPETOT
  istart= IMPORT_INDEX(neib-1)
  inum  = IMPORT_INDEX(neib  ) - istart
  do k= istart+1, istart+inum
    X(IMPORT_NODE(k))= WR(k)
  enddo
enddo
```

図 6 に示されるような通信は，Krylov 反復解法の行列ベクトル積（matrix-vector product）の部分で発生する。

(a) 領域間通信サブルーチンの呼び出し（スカラー型，3×3 ブロック型）

```
1x1 Scalar
      allocate (WS(NP), WR(NP), X(NP))
      call SOLVER_SEND_RECV                                          &
     &    ( NP, NEIBPETOT, NEIBPE, IMPORT_INDEX, IMPORT_NODE,        &
     &      EXPORT_INDEX, EXPORT_NODE, WS, WR, X , SOLVER_COMM,      &
     &      my_rank)
```

```
3x3 Block
      allocate (WS(3*NP), WR(3*NP), X(3*NP))
      call SOLVER_SEND_RECV_3                                        &
     &    ( NP, NEIBPETOT, NEIBPE, IMPORT_INDEX, IMPORT_NODE,        &
     &      EXPORT_INDEX, EXPORT_NODE, WS, WR, X , SOLVER_COMM,      &
     &      my_rank)
```

(b) 領域間通信サブルーチンの概要

－ 送信フェーズ

```
do neib= 1, NEIBPETOT
  istart= EXPORT_INDEX(neib-1)
  inum  = EXPORT_INDEX(neib  ) - istart
  do k= istart+1, istart+inum
    WS(k)= X(EXPORT_NODE(k))
  enddo
  call MPI_ISEND
       (WS(istart+1), inum, MPI_DOUBLE_PRECISION, &
        NEIBPE(neib), 0, SOLVER_COMM,             &
        req1(neib), ierr)
enddo
```

－ 受信フェーズ

```
do neib= 1, NEIBPETOT
  istart= IMPORT_INDEX(neib-1)
  inum  = IMPORT_INDEX(neib  ) - istart
  call MPI_IRECV
       (WR(istart+1), inum, MPI_DOUBLE_PRECISION, &
        NEIBPE(neib), 0, SOLVER_COMM,             &
        req2(neib), ierr)
enddo

call MPI_WAITALL (NEIBPETOT, req2, sta2, ierr)

do neib= 1, NEIBPETOT
  istart= IMPORT_INDEX(neib-1)
  inum  = IMPORT_INDEX(neib  ) - istart
  do k= istart+1, istart+inum
    X(IMPORT_NODE(k))= WR(k)
  enddo
enddo

call MPI_WAITALL (NEIBPETOT, req1, sta1, ierr)
```

図 6　GeoFEM における領域間通信プロセス〔2,3〕

GeoFEM では初期全体メッシュデータから局所分散メッシュデータを自動的に生成するためのツールとして領域分割ツール（Partitioner）が用意されている。利用者は実際には上記の通信テーブルについては意識することなく並列有限要素法コードの開発，利用が可能である。領域分割にあたっては：

- 各領域の負荷が均等となっていること
- 領域間の通信が少ないこと

が重要である。特に前処理付き反復法を使用する場合には収束を速めるために後者が重要なポイントである〔2,3,4,5,6〕。この両条件を満たす手法としては METIS[8] が良く知られている。GeoFEM の領域分割ツールでは文献〔10〕で紹介されている RCB 法（Recursive Coordinate Bisection）等のほか，METIS に関するインターフェースも提供している。

3. GeoFEM ベンチマーク

（1）概　要

本研究ではGeoFEMプロジェクトで開発された並列有限要素法アプリケーションを元に整備した性能評価のためのベンチマークプログラム群〔7〕を使用した。

GeoFEM ベンチマークは，①三次元弾性問題（Cube モデル，PGA モデル），②三次元接触問題，③二重球殻間領域三次元ポアソン方程式，に関する並列前処理付き反復法ソルバーの実行性能（GFLOPS 値）を様々な条件下で計測するものである。プログラムは全て OpenMP ディレクティヴを含む FORTRAN90 および MPI で記述されている。

各ベンチマークプログラムでは, GeoFEM で採用されている局所分散データ構造（図 3～図 5）を使用しており，マルチカラー法に基づくオーダリング手法によりベクトルプロセッサ，SMP 並列計算において高い性能が発揮できるように最適化されている。また，MPI，OpenMP，Hybrid（OpenMP＋MPI）の全ての環境で稼動し，SMP クラスタの性能評価に適している。①～③の各ベンチマークは 8 コアから成る SMP ノードの性能評価のためのものであるが，①の Cube モデルは任意の問題サイズで任意のコア数を使用したベンチマークテストが可能である。GeoFEM ベンチマークに基づいた，三次元非定常伝熱解析プログラム「GAPgeofem」は「SPEC MPI 2007」ベンチマーク[9]の一つとして採用されている。

様々なハードウェアに対応可能なように，連立一次方程式の係数マトリクスの格納法として図 7 に示す 2 種類の方法が準備されている。ベクトルプロセッサ向けには，長いループ長が得られるように図 7（a）に示す Descending order Jagged Diagonal Storage（DJDS）法を採用している。スカラープロセッサ向けには非対角成分の走査方向を変えた Descending order Compressed Row Storage（DCRS）（図 7（b））を利用可能である。DCRS では最内ループ長が短くなるが，最内ループにおけるデータの局所性を保つことが可能であり，キャッシュの有効利用に適している〔4,6〕。

以下に各ベンチマーク問題について説明し，「地球シミュレータ（ベクトルプロセッサ）（ES）」および「IBM SP3（スカラープロセッサ）（SP3）（米国国立ローレンスバークレイ研究所）」[10]の 1 ノード，8 コアを使用した場合の計算結果〔2,3,4,5,6〕について紹介する。表 1 は「地球シミュレータ」，「IBM SP3」および 4 章以降で扱う「Hitachi SR11000/J2」のハードウェア諸元について示したものである。「Hitachi SR11000/J2」の MPI latency については公表された値が

[8] http://glaros.dtc.umn.edu/gkhome/views/metis/
[9] http://www.spec.org/auto/mpi2007/Docs/128.GAPgeofem.html
[10] http://www.nersc.gov/

ないため，文献〔11〕に示されている，ほぼ同じ性能と推定される IBM p5-575（米国国立ローレンスバークレイ研究所）の値を，参考値として記載している。メモリバンド幅は STREAM ベンチマーク[11]による実測値である。

「地球シミュレータ」は NEC SX-6 に基づく並列ベクトル計算機であり, 640 の SMP ノード, 5,120 のベクトルプロセッサと 10 TB のメモリから構成されている。ピーク性能は 40 TFLOPS である。各 SMP ノードは 8 個のベクトルプロセッサ，16GB のメモリから構成されており，各 PE のピーク性能は 8 GFLOPS である。各 SMP ノードは単段クロスバーにより接続されており，双方向の転送速度は 12.3 GB/sec.である。

IBM SP-3 は IBM POWER3 に基づいたスカラーシステムであり，380 の SMP ノード，6,080 の PE，7.3TB のメモリから構成されており，ピーク性能は 9.12 TFLOPS である。各 SMP ノードは 16 個の PE，16～64GB のメモリから構成されており，各 PE のピーク性能は 1.50 GFLOPS である。各 PE は 64KB の L1 キャッシュと 8MB の L2 キャッシュをそれぞれ独立に持っている。各 SMP ノードは双方向の転送速度が 2.00 GB/sec.のスイッチにより接続されている。本研究では，ES との比較のため，各 SMP ノード 8 PE を使用した。Hitachi SR11000/J2 については次章で説明する。

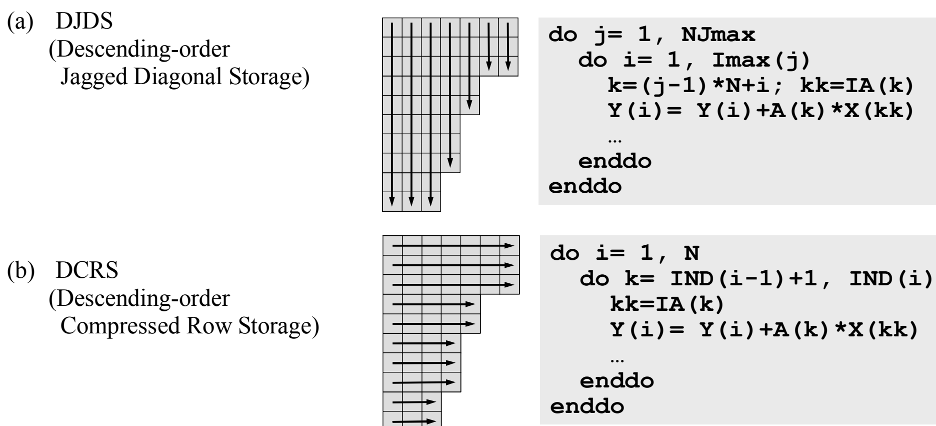


図 7　係数マトリクスの格納方法とループ構造

表 1　地球シミュレータ，IBM SP3，Hitachi SR11000/J2 のハードウェア諸元〔4,6,11〕

	Earth Simulator	IBM SP3	Hitachi SR11000/J2
Core#/node	8	16	16
Architecture of each Core	NEC SX6	IBM POWER3	IBM POWER5+
Clock Tate (MHz)	500	375	2300
Peak Performance/core (GFLOPS)	8.00	1.50	9.20
Memory/node (GB)	16	64	128
Measured Memory BW (GB/sec/core)	26.6	0.623	6.40
Network BW (GB/sec/node)	12.3	2.00	12.0
MPI Latency (μsec)	5.6-7.7	16.3	4.7〔11〕

[11] http://www.streambench.org/

（2）三次元弾性問題

三次元弾性問題の対象は，単純形状（Cube）モデル（図 8）と図 9 に示すような PC のマイクロプロセッサの Pin Grid Array（PGA）を模擬したモデルである。いずれも，三次元弾性問題を局所不完全コレスキー分解付き共役勾配法（局所 ICCG 法）により解く。Cube モデルは任意の問題サイズ，領域数でのベンチマークを実施可能である。PGA モデルは問題規模が固定されており（1,012,354 節点，3,037,062 自由度（Degrees of Freedom : DOF）），マルチカラーの色数の効果，OpenMP と MPI（8 領域）の比較検討に使用される。

図 10 は，Cube 型モデルについて，1 ノード（8 コア）において，色数＝100 または最内ループ長＞256 とした場合の，様々な問題サイズ（自由度数 : degrees of freedom，以下 DOF）における計算結果（GFLOPS 値）である。ES では，ベクトル機の特徴として 3.81 GFLOPS（ピークの 6%）から 22.7GFLOPS（ピークの 35.5%）まで，問題サイズとともに増加する。また，ループ長を長くとれる DJDS（●○）が DCRS（■□）よりも性能が高い。MPI（●■）と OpenMP（○□）の差はほとんど無いが，MPI が若干速い。SP3 では L2 キャッシュの効果により，問題サイズが小さい場合に性能が高い。DJDS と DCRS，MPI と OpenMP の違いは少ないが，特に問題サイズが小さい場合は，各 PE に独立に装着された L2 キャッシュを有効利用できる手法（DCRS，MPI）の性能が高い。表 2 は，ハードウェアの諸元（ピーク性能，メモリバンド幅）を元に，1 コアあたりの性能を予測し，図 10 に示した測定値と比較したものである。MPI（8 コア）を使用して，3,000,000（$=3\times100^3$）DOF の問題を色数 100 で計算した場合の性能を 8 で割ったものである。性能は文献〔12〕に示した推定法に基づき，表 1 に示した各プロセッサのピーク性能とメモリバンド幅（STREAM ベンチマークによる実測値）によって推定した。SP3 においてキャッシュの効果は考慮していない。ES では推定値と実測値は非常によく一致している。SP3 のようなスカラープロセッサではキャッシュの効果を考慮していないこともあり，実測値は概して予測値を上回っている。

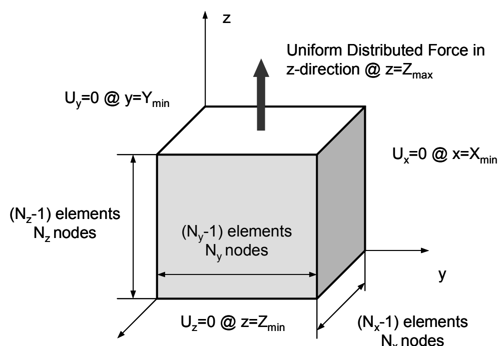


図 8　Cube モデルの概要，境界条件

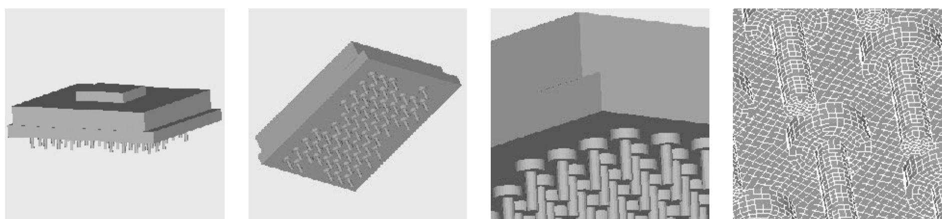
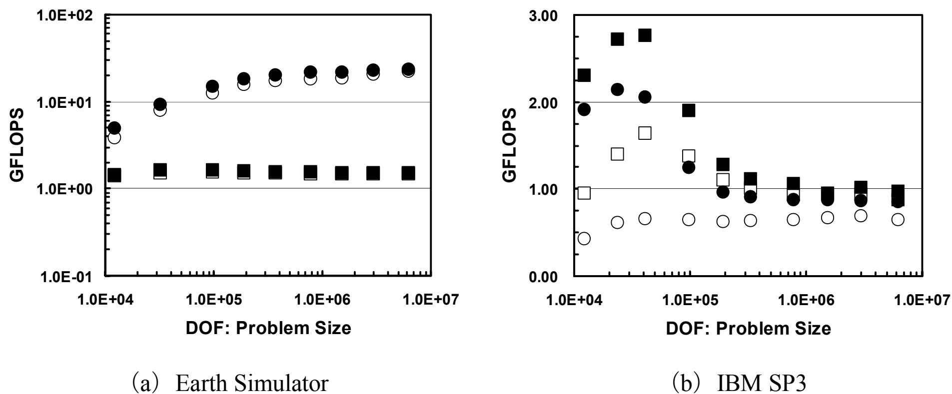


図 9　PGA モデルの概要（956,128 要素，1,012,354 節点，3,037,062 自由度）



（a）Earth Simulator　　　　（b）IBM SP3

図 10　Cube モデルの計算結果（問題規模と性能の関係），並列プログラミングモデル，行列格納方法の影響（●：MPI/DJDS，○：OpenMP/DJDS，■：MPI/DCRS，□：OpenMP/DCRS）

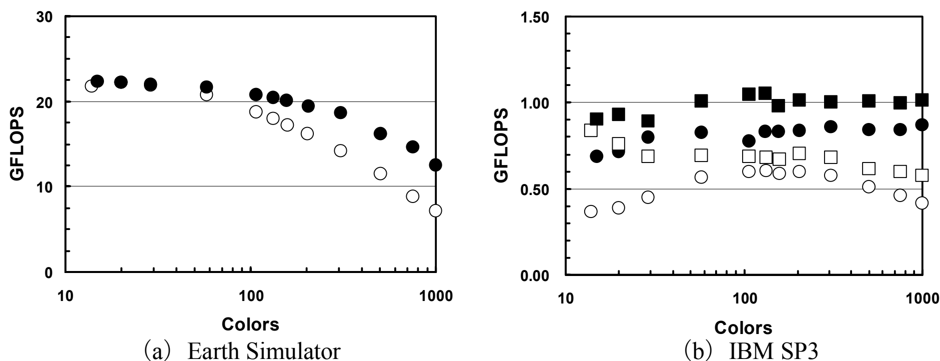
表 2　地球シミュレータ，IBM SP-3 のハードウェア諸元と Cube モデルの性能〔4,6〕

	Earth Simulator	IBM SP-3
Peak Performance/core (GFLOPS)	8.00	1.50
Measured Memory BW (GB/sec/core)	26.6	0.623
Estimated Performance (GFLOPS (% of peak))	2.31-3.24 (28.8-40.5)	0.072-0.076 (4.80-5.05)
Measured Performance (GFLOPS (% of peak))	2.93 (36.6) (DJDS)	0.122 (8.11) (DCRS)

図 11 は PGA モデルの実行例（色数の効果）である。マルチカラーオーダリングに基づく反復解法では，色数を増加させることによって反復回数は減少する〔2,3,4,5,6〕。しかしながら，各色内での要素数が減少するため，ループ長が短くなり，ベクトルプロセッサにおける性能（GFLOPS 値）は低下することが知られている〔2,4,6〕。ES では MPI，OpenMP のいずれも色数が増加すると GFLOPS 値は低下している。この傾向は OpenMP において特に顕著である。これは主として，不完全コレスキー分解 (IC) による前処理の前進後退代入処理における OpenMP のオーバーヘッドによるものと考えられる（図 12）。

SP3 ではキャッシュを有効利用できる手法 (DCRS，MPI) の性能が高い (■＞●＞□＞○)。スカラープロセッサにおいては色数の性能に対する影響は顕著ではないが，DJDS（●○）を採用すると，図 11 に示すように色数が増加するほど性能は向上する。これは，図 7（a）における最内ループが色数増加とともに短くなり，データの局所性が増大し，キャッシュがより有効に利用されるためと考えられる。この結果からベクトル計算機向けに開発されたマルチカラーオーダリングを使用したコードは，色数を多くすることにより，スカラープロセッサでも高い性能を達成することが可能になる。

OpenMP の場合（○□）は DJDS，DCRS に関わらず色数が増加すると，図 8 に示した前進後退代入処理における OpenMP のオーバーヘッドにより性能が低下する。OpenMP/DJDS（○）の場合は 10 色～100 色程度までは色数の増加によって性能が向上する傾向が見られるが，色数が 100 から 1000 まで増加すると，OpenMP のオーバーヘッドによる性能の低下が見られる。



（a）Earth Simulator　　　　（b）IBM SP3

図 11　PGA モデルの計算結果（色数と性能の関係），並列プログラミングモデル，行列格納方法の影響（●：MPI/DJDS，○：OpenMP/DJDS，■：MPI/DCRS，□：OpenMP/DCRS）

```
do iv= 1, NCOLORS
!$omp parallel do private (iv0,j,iS,iE… etc.)
do ip= 1, PEsmpTOT
  iv0= STACKmc(PEsmpTOT*(iv-1)+ip- 1)
  do j= 1, NLhyp(iv)
    iS= INL(npLX1*(iv-1)+PEsmpTOT*(j-1)+ip-1)
    iE= INL(npLX1*(iv-1)+PEsmpTOT*(j-1)+ip  )
!CDIR NODEP
    do i= iv0+1, iv0+iE-iS
      k= i+iS - iv0
      kk= IAL(k)
      X(i)= X(i) - A(k)*X(kk)*DINV(i) etc.
    enddo
  enddo
enddo
enddo
```

SMP parallel

Vectorized

図 12　OpenMP による IC 前処理における前進後退代入の並列化例〔2,3,4,5〕

（3）三次元接触問題

プレート境界の断層接触面（図 13 参照）における応力蓄積と地震発生サイクルのシミュレーションは，地震発生メカニズムに関する知見を得るために重要なアプリケーションであり，「GeoFEM」のターゲットアプリケーションの一つである〔2,3〕。「選択的ブロッキング（Selective Blocking：SB）」前処理はこのような問題を効率良く計算するために著者によって開発された手法である〔2〕。ペナルティ数を導入することによって断層周辺の拘束条件を表現している場合に特に有効である。

三次元固体力学においては 1 節点に 3 方向の変位成分が自由度として存在するため，これらの 3 自由度をブロック化して取り扱っている。IC／ILU 型前処理では，この 3×3 行列に完全 LU 分解を適用することによって，より安定な収束性を得ている。選択的ブロッキングでは，「接触グループ」に属する節点群を並べ替えによって，1 つのグループとして扱い，このグループ（選択的ブロック）における（3×NB）×（3×NB）行列（但し NB は選択的ブロック内の節点数）に対して完全 LU 分解を適用するものである（図 14 参照）。対象とする行列が対称正定の場合には，選択的ブロッキング（SB）を，Fill-in なしのブロック ICCG 法と組み合わせた SB-BIC(0)-CG 法が非常に有効であり，広範囲のペナルティ数に関して安定である〔2〕。図 13 に示す西南日本領域を対象とした固定サイズのモデル（784,000 要素，2,471,439 DOF）を解き，マルチカラーの色数の効果，OpenMP と MPI（8 領域）の比較検討を実施する。マトリク

ス格納法は DJDS のみである。図 15 に示すように，色数と性能の関係については三次元弾性解析（PGA モデル）と同様である。

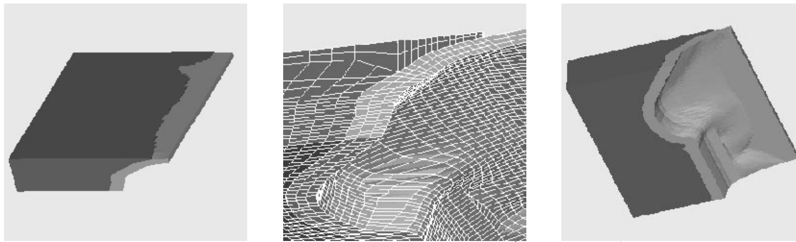


図 13　接触解析のための西南日本モデル

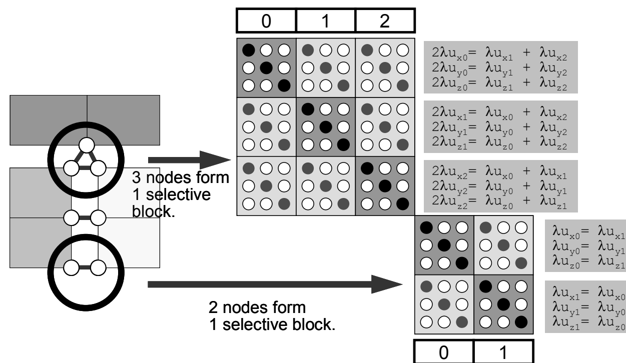
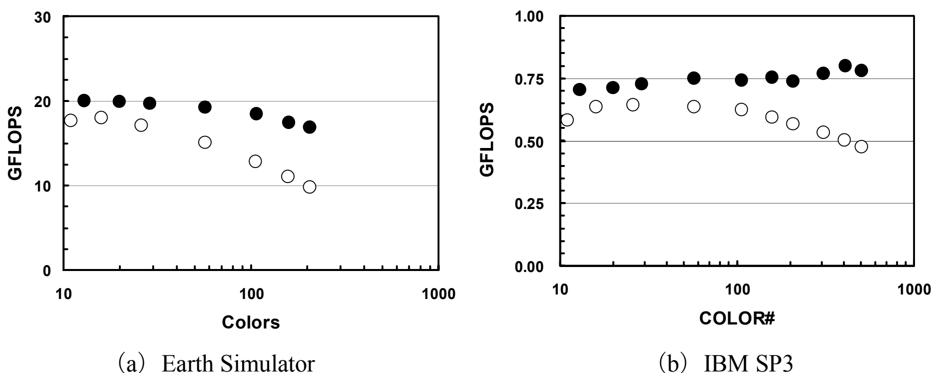


図 14　選択的ブロッキング（selective blocking）前処理の概要：同じ「接触グループ」に属する節点群をブロック化して解く



（a）Earth Simulator　　（b）IBM SP3
図 15　接触問題（西南日本モデル）の計算結果（色数と性能の関係），並列プログラミングモデルの影響（●：MPI/DJDS，○：OpenMP/DJDS）

（4）二重球殻間領域三次元ポアソン方程式

マントル対流，海洋大循環モデルなどで使用される，二重球殻間の領域における非圧縮性流体のシミュレーションにおいて得られるポアソン方程式を，Gauss-Seidel 法を緩和演算子とするマルチグリッド前処理付き CG 法（MGCG）で解く。空間は図 16 に示すように正二十面体を分割して得られる三角形を底面とする三角柱メッシュによって離散化されており，メッシュ分割のための階層構造をマルチグリッドに使用する。問題規模は，6,144,000 要素に固定されてお

り，マルチカラーの色数の効果，OpenMP と MPI（8 領域）の比較検討を実施する。マトリクス格納法は DJDS のみである。

図 17 は ES, SP3 を使用した場合の色数と GFLOPS 値の関係である〔4.6〕。色数を増やすと，MPI ではほとんど計算性能の変化は無いが，OpenMP では色数の増加とともに計算時間が増加する。色数が 12 色と 2000 色の場合を比較すると計算性能の比は 6.02（SR8000/G1），3.90（SP3），となる。これは（2），図 12 で述べた前進後退代入処理における OpenMP の同期オーバーヘッドによるものと考えられる。マルチグリッド法では，Gauss-Seidel 法による緩和計算において図 12 に示すのと同様な前進後退代入処理が発生するが，粗い格子上では計算量そのものが減るため，同期オーバーヘッドの影響を受けやすくなる。したがって，マルチカラーオーダリングによる ICCG 法と比較して，色数によるオーバーヘッドの増加はより顕著である。

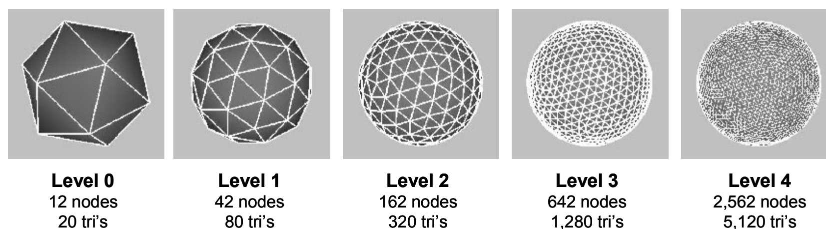
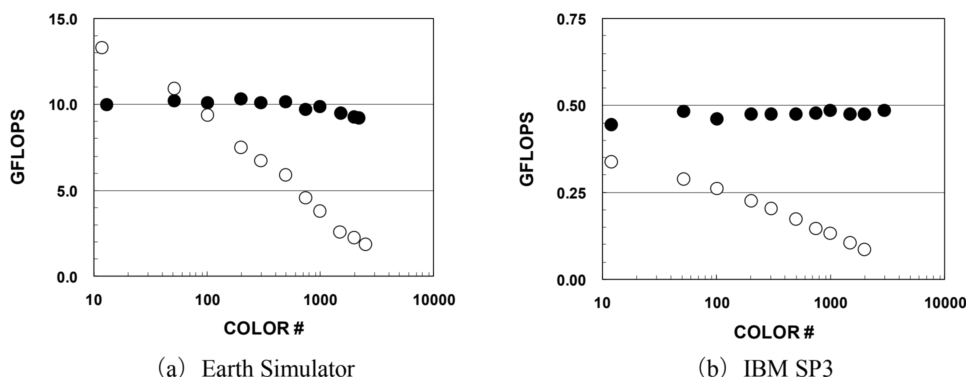


図 16　正二十面体の分割によって生成した球面上の三角形メッシュ



（a）Earth Simulator　　（b）IBM SP3

図 17　二重球殻モデル（多重格子法）の計算結果（色数と性能の関係），並列プログラミングモデルの影響（●：MPI/DJDS，○：OpenMP/DJDS）

（5）三次元弾性問題（複数ノード）

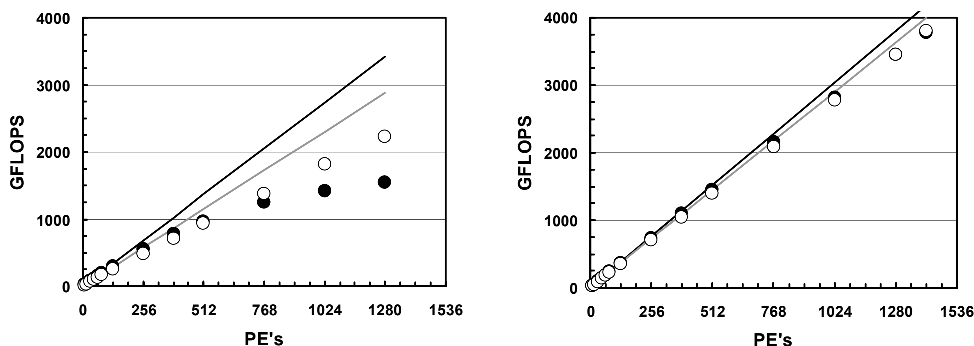
最後に図 8 に示した Cube モデルにおける三次元弾性解析を 100 SMP ノード以上を使用して計算した場合の結果について示す。Hybrid と Flat MPI それぞれの性能を評価した。PE（または SMP ノード）あたりの問題規模を固定し，ノード数を変化させた場合の計算（Weak Scaling）を実施した。ES の場合は 1 ノード（8 PE）～176 ノード（1,408 PE），IBM SP-3 の場合は 1 ノード（8 PE）～128 ノード（1,024 PE）を使用した。

ES では DJDS を使用した場合，最大問題サイズは 2.21×10^9 DOF，最高性能は 3.80 TFLOPS であり，これは ES の 176 ノード（1,408 PE）のピーク性能（10.24 TFLOPS）の 33.7%に相当する（図 18）。直線は，1 ノード（8 PE）における性能を基準とした，Flat MPI，Hybrid の場合の理想値である。Hybrid と Flat MPI はほぼ同じ性能であるが，SMP ノード数が増加し，かつ各

PE あたりの問題規模が比較的小さいとき，Hybrid の性能が卓越する（図 18）。これは文献〔13〕で紹介されているように，ES の MPI latency の値が比較的高いため，MPI プロセス数が増加すると通信の遅延効果が大きくなるためと予想される。Flat MPI は Hybrid と比較して MPI プロセスの数が 8 倍となるため，SMP ノード数が大きく，PE 当たりの問題規模が比較的小さいときはこの効果は顕著になる。

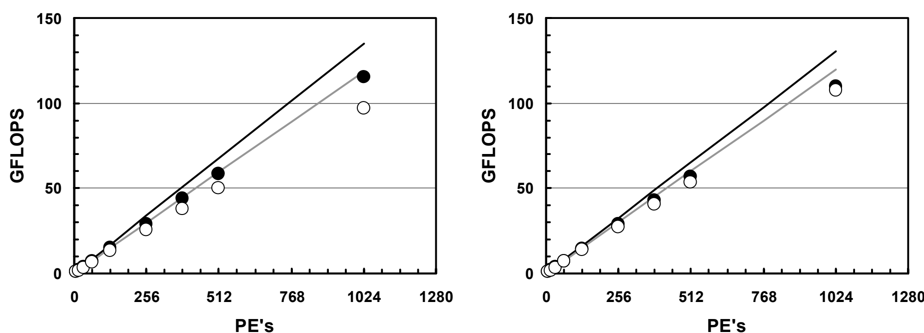
図 19 は IBM SP3（DCRS）による結果である。最大問題サイズは 3.84×10^8 DOF，最高性能は 110 GFLOPS であり，これは 128 ノード（1,024 PE）のピーク性能の 7.16 %に相当する。高い並列性能が得られているが，ES の場合に見られたような，SMP ノード数が増加した場合の Flat MPI の性能低下は観察されなかった。128 ノードにおける結果は 1 ノードの性能の外挿に近い値である。ES，SP3 いずれの場合も，PE あたりの問題規模が大きい場合は，通信のオーバーヘッドの影響が少なく，SMP ノード数が増えても理想値に近い性能が得られている。

図 20 は，更にわかりやすいように，ES および SP3 について，PE あたりの問題規模が大きいケースについて，1 ノード（8 PE）からの計算性能増加率（Speed-Up）として表した図である。Hybrid, Flat MPI ともにほぼ理想値に近い性能の増加傾向を見せているが, ES, SP3 ともに Hybrid の方が若干増加率が高い。



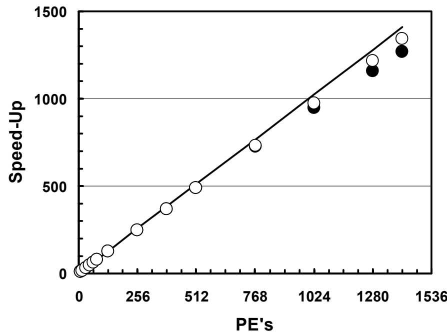
（a）3×32^3=98,304 DOF/PE　　（b）3×64×64×128=1,572,864 DOF/PE

図 18　「地球シミュレータ（ES）」における Cube モデルの計算結果（Weak Scaling）（●：Flat MPI/DCRS，○：Hybrid/DCRS，－：Flat MPI/DCRS（ideal），－：Hybrid/DCRS （ideal））

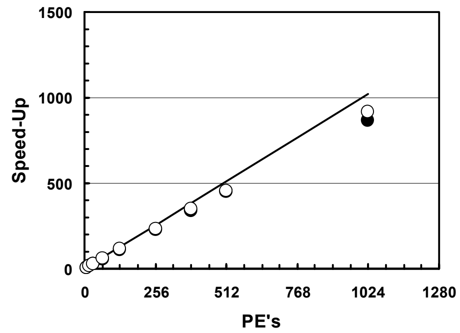


（a）3×32^3=98,304 DOF/PE　　（b）3×50^3=375,000 DOF/PE

図 19　「IBM SP3」における Cube モデルの計算結果（Weak Scaling）（●：Flat MPI/DCRS，○：Hybrid/DCRS，－：Flat MPI/DCRS（ideal），－：Hybrid/DCRS （ideal））



(a) ES：3×64×64×128=1,572,864 DOF/PE



(b) IBM SP3：3×50^3=375,000 DOF/PE

図 20　「地球シミュレータ」，「IBM SP3」における Cube モデルの計算結果，1 ノードからの計算性能増加率（Weak Scaling）（●：Flat MPI/DCRS，○：Hybrid/DCRS，－：ideal）

表 2 からもわかるように，Cube モデルの計算を ES で実行した場合 IBM SP3 と比較して約 25 倍程度の GFLOPS 値が得られている。しかしながら，表 1 からも明らかなように，MPI latency の値はそれほど大きく変わらない。

三次元固体力学に対して反復法を適用した場合，行列ベクトル積の計算（Matrix-Vector Multiplication：mat-vec）においては以下に示すように，1 つの非対角成分に対して 18 回の浮動小数点演算が必要になる〔12〕。

```
do j= 1, NLU
do i= 1, N
  k= ITEM(i,j)
  Y(3*i-2)= Y(3*i-2) + AMAT(1,i,j)*X(3*k-2) +                    &
                       AMAT(2,i,j)*X(3*k-1) + AMAT(3,i,j)*X(3*k)
  Y(3*i-1)= Y(3*i-1) + AMAT(4,i,j)*X(3*k-2) +                    &
                       AMAT(5,i,j)*X(3*k-1) + AMAT(6,i,j)*X(3*k)
  Y(3*i  )= Y(3*i  ) + AMAT(7,i,j)*X(3*k-2) +                    &
                       AMAT(8,i,j)*X(3*k-1) + AMAT(9,i,j)*X(3*k)
enddo
enddo
```

非対角成分の数を 30 とすると，一回の mat-vec の計算における浮動小数点演算の回数は，FEM における節点数を N とすると 540N 程度となる。例えば 1 PE において $N=32^3$ の場合（= 98,304 DOF/PE= 786,432 DOF/SMP-node，図 18（a），図 19（b）の小規模問題ケースに相当する），ES の性能が 2.80 GFLOPS（ピークの 35%）とすると，計算時間は約 6 msec である。並列有限要素法では mat-vec を一回実行するたびに，領域境界のデータ交換が必要になり，通信が発生する（図 5，図 6 参照）〔2〕。通信時間は 50 μsec のオーダーであり，ほとんど無視できる。また，また，図 6 に示すように，領域間のデータ交換には，送信・受信バッファへのコピーが必要となるが，ES では表 1 に示すように非常に高いメモリバンド幅を実現しているので，これもほとんど無視できる。表 1 によると ES の MPI latency が 7 μsec 程度であるので，もし 1,000 PE 以上を使用した場合，計算時間と遅延時間が同じオーダーになり，影響は深刻となる可能性がある。

実際，図 21 に示すように，ES では，1 反復あたりの通信オーバーヘッド（1 ノード＝8PE の計算時間を基準として算出）は問題サイズ（すなわち通信量）に依存せず，特に SMP ノード数

が増加した場合は，Flat MPI と Hybrid の差のみが顕著に現れており，MPI latency の影響が大きいであることがわかる〔6〕。IBM SP3 の場合はこれほど顕著では無いが，図 20 に示すように，問題規模が大きい場合でも，SMP ノード数が増加した場合には，Hybrid が Flat MPI よりも性能増加率が高くなる傾向にある。

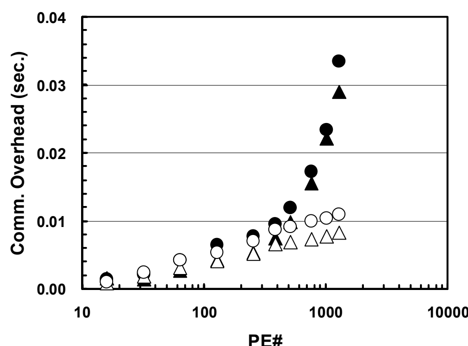


図 21 「地球シミュレータ」における 1 反復あたりの通信オーバーヘッド（1 ノードの計算時間との差から算出，DJDS）（●：Flat MPI: 3×50^3=375,000 DOF/PE，○：Hybrid: 3×50^3=375,000 DOF/PE，▲：Flat MPI: 3×32^3=98,304 DOF/PE，△：Hybrid: 3×32^3=98,304 DOF/PE）

4. Hitachi 11000 における実行例（1 ノード）

（1）概要

Hitachi SR11000/J2（SR11000/J2）[12]の 1 ノードを使用して，GeoFEM ベンチマークを実施した。2007 年 3 月以前は, 本学情報基盤センターにおいては, IBM POWER5 プロセッサ（1.90 GHz）に基づく Hitachi SR11000/J1（SR11000/J1）が導入されていた。Hitachi SR11000 は 1 ノード 16 コアから構成される SMP クラスタ型アーキテクチュアを採用しているが〔14〕，SR11000/J1 ではこの 1 ノードを 8 コアずつから構成される SMP ノード 2 つに更に分割して, 使用した〔7〕。比較のため，ここでは SR11000/J1，SR11000/J2 ともに，1 ノードあたり 8 コアを使用した場合についての検討を中心に実施する。プログラムのコンパイルにあたっては，推奨オプションである「-Oss -64 -looptiling (-noparallel または -omp)」を適用した。

Hitachi SR11000/J2 のハードウェア諸元は表 1 に既に示した通りである。2 つの IBM POWER5+ コア（2.3 GHz，ピーク性能：9.2 GFLOPS）によって，POWER5+チップが構成されている。各コアは 32KB の L1 キャッシュを持ち，L2 キャッシュ（1.875 MB），L3 キャッシュ（36 MB）は各チップ内で 2 つのコアに共有されている。チップ内にはメモリコントローラが内蔵されており，高速で信頼性の高いメモリへのアクセスが可能である。4 つのチップ，すなわち 8 つのコアからモジュール（Multi Chip Mudule：MCM）が構成され，2 つの MCM，すなわち 16 個のコアが 1 つの SMP ノードを形成している。POWER5+は大容量のキャッシュを搭載しているが，広範囲なアプリケーションで高性能を実現するためには，メモリ上の大規模データへのアクセス機能を高める必要がある。Hitachi SR11000/J2 ではこのために擬似ベクトル処理（Pseudo Vector Processing：PVP），コンパイラによるソフトウェアアシストプリフェッチがサポートされており，安定した高いメモリアクセス性能が実現されている〔14〕。各 SMP ノードは 128 GB

[12] http://www.cc.u-tokyo.ac.jp/

のメモリを搭載しており，全体システムでは，128 ノード（2,048 コア），18.8 TFLOPS のピーク性能，16.4 TB の主記憶容量である。各 SMP ノードは三次元クロスバーにより接続されており，双方向の転送速度は 12.0 GB/sec.である。

（2）三次元弾性問題（Cube モデル）

図 22 は SR11000/J1，　SR11000/J2 の 8 コアを使用して様々な規模で三次元弾性解析（Cube モデル（図 8），色数＝100 または最内ループ長＞256）を実施した場合の結果である。表 3 は，表 2 と同様に，1 コアあたりの性能を予測した値と実測値とを比較したものである。図 10（b）で示したスカラープロセッサの典型的な挙動を示しており，問題規模が小さい場合はキャッシュを有効利用できているが，規模が大きくなると若干性能が低下する。しかしながら，その低下は SP3 の場合と比較して顕著ではない。表 3 に示すように，メモリバンド幅（実測）をピーク性能で割った BYTE/FLOP の値は，POWER3 と比較して POWER5，POWER5+は約 1.5 倍になっている。また，SR11000/J1，SR11000/J2 ともに，実測性能は予測値を大きく上回っている。予測値算定の際には，〔12〕の手法に基づき，キャッシュの効果を無視している。しかしながら，各 SMP ノードの 8 コアを使用した場合には，各チップ（2 コア）に装着された L2 キャッシュ（1.875 MB），L3 キャッシュ（36 MB）を 1 コアで利用できるため，問題規模が大きくなっても，キャッシュが効果的に機能しているものと考えられる。

図 23 は Hitachi SR8000/MPP（東京大学情報基盤センター）を使用した場合の結果である〔4,5,6〕。SR8000 では擬似ベクトル処理（PVP）の効果が高く，図 10（a）に示した「地球シミュレータ」の挙動にむしろ近くなっている。それと比較すると，SR11000/J1，SR11000/J2 では擬似ベクトル処理の効果は小さい。SR11000/J2 と SR8000 を比較すると，約 4 倍の速度向上が達成されている。

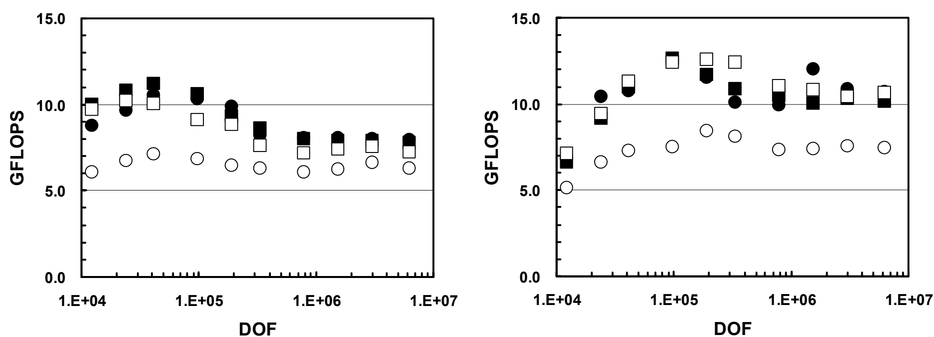
MPI と（●■）OpenMP（○□）の性能は変わらないが，OpenMP/DJDS（○）の性能は，他の場合と比較して低い。各チップに装着されたキャッシュを効率的に利用できないため，特に小規模問題では性能が悪い。

図 24 は，行列格納方法として DCRS を採用した場合に：

① SR11000/J1　8 コア（図 22 と同じ）●
② SR11000/J2　8 コア（図 22 と同じ）●
③ SR11000/J2　16 コア（全体問題規模が図 22 の場合と同じ）□
④ SR11000/J2　16 コア（コアあたり問題規模が図 22 の場合と同じ）△

について問題規模と計算性能（対ピーク性能比）の関係を示したものである。表 3，図 24 からわかるように，8 コアで比較すると，SR11000/J2（●）は SR11000/J1（●）よりも対ピーク性能比は高く，図 24（b）からもわかるように特に OpenMP の場合に顕著である。SR11000/J2 において，16 コアを使用した場合，コアあたりの問題規模による性能の差はほとんどなく（□，△），8 コアの場合からの性能低下はそれぞれ約 15%（MPI），約 21%（OpenMP）であり，OpenMP の場合の方が影響が大きい。また，OpenMP と MPI を比較すると，MPI の方が若干性能が良い。図 25 は SR11000/J2 において，コアあたりの問題規模を固定して，8 コア，16 コアを使用した場合の比較である。図 25（b）に見られるように，16 コアを使用した場合，

OpenMP/DJDS（○）と他の手法との差は8コアの場合と比較して更に顕著である。また8コア使用の場合（図25（a））では，IBM SP3で見られたような問題規模増加による性能低下（図10 (b)）はほとんど認められなかったが，16コアの場合はより明瞭に認められる（図25 (b)）。8コア使用の場合には，各チップに装着されたL2キャッシュ，L3キャッシュを全て1コアで使用することが可能であるが，16コアの場合は2コアでの共有となるため，問題規模増加による性能低下の効果がより顕著に認められる。



（a）Hitachi SR11000/J1　　（b）Hitachi SR11000/J2

図 22　Cube モデルの計算結果（問題規模と性能の関係），並列プログラミングモデル，行列格納方法の影響（●：MPI/DJDS，○：OpenMP/DJDS，■：MPI/DCRS，□：OpenMP/DCRS）

表 3　Hitachi SR11000/J1，Hitachi SR11000/J2 のハードウェア諸元と Cube モデルの性能〔4,6,7〕

	Hitachi SR11000/J1	Hitachi SR11000/J2	ES	IBM SP-3
Peak Performance/core (GFLOPS)	7.60	9.20	8.00	1.50
Measured Memory BW (GB/sec/core)	4.62	6.40	26.6	.623
Estimated Performance (GFLOPS (% of peak))	0.643-0.703 (8.45-9.24)	0.880-0.973 (9.56-10.6)	2.31-3.24 (28.8-40.5)	0.072-0.076 (4.80-5.05)
Measured Performance (GFLOPS (% of peak))	0.998 (13.1) (DCRS)	1.34 (14.5) (DCRS)	2.93 (36.6) (DJDS)	0.122 (8.11) (DCRS)
BYTE/FLOP	0.608	0.696	3.325	0.413

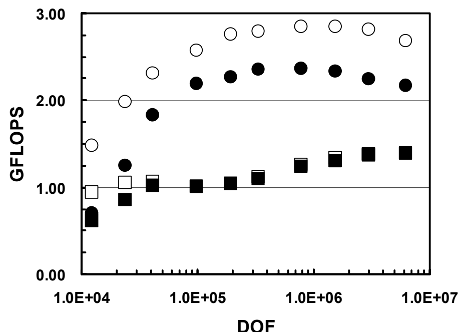
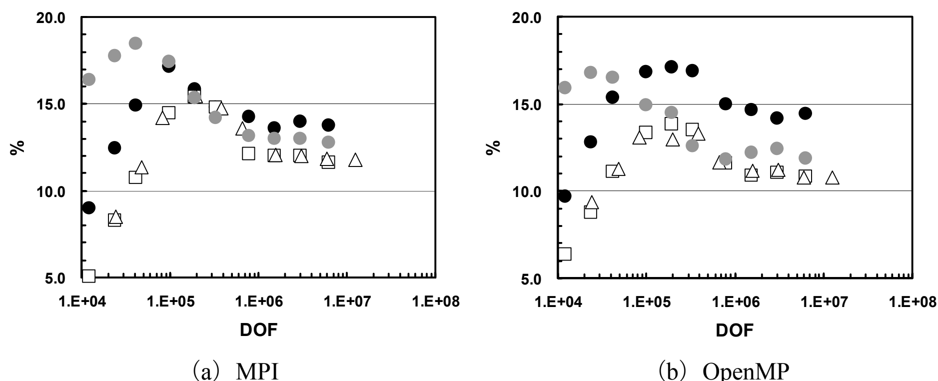
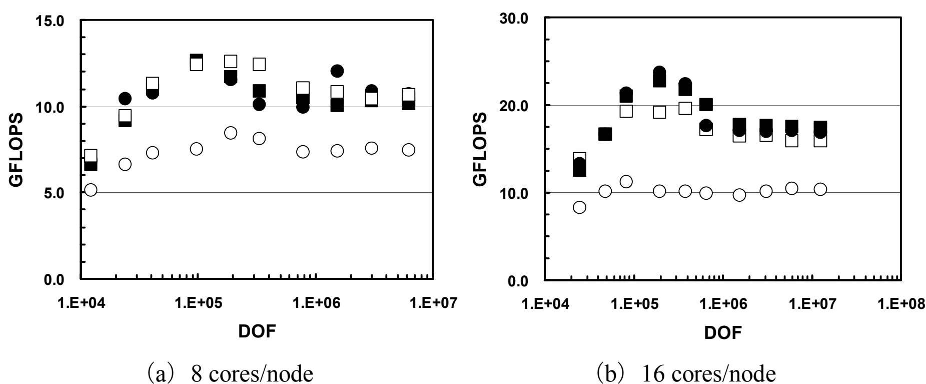


図 23　「Hitachi SR8000/MPP」による Cube モデルの計算結果（問題規模と性能の関係），並列プログラミングモデル，行列格納方法の影響（●：MPI/DJDS，○：OpenMP/DJDS，■：MPI/DCRS，□：OpenMP/DCRS）〔4,5,6〕



（a）MPI　　（b）OpenMP

図 24　Cube モデルの計算結果（問題規模と性能（対ピーク性能比）の関係），並列プログラミングモデル，プロセッサ，SMP ノードあたりコア数の影響（●：SR11000/J1/DCRS 8 cores, ●：SR11000/J2/DCRS 8 cores, □：SR11000/J2/DCRS 16 cores（全体問題規模が●の場合と同じ），△：SR11000/J2/DCRS 16 cores（コアあたり問題規模が●の場合と同じ））

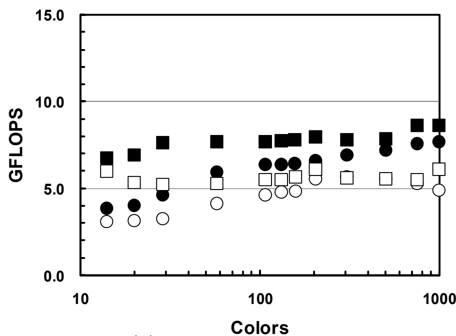


（a）8 cores/node　　（b）16 cores/node

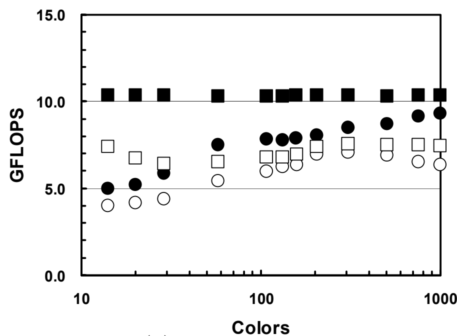
図 25　「Hitachi SR11000/J2」における Cube モデルの計算結果（問題規模と性能の関係），並列プログラミングモデル，行列格納方法の影響，1 ノード 8 コアまたは 16 コアを使用した場合の比較，コアあたり問題規模は同じ（●：MPI/DJDS，○：OpenMP/DJDS，■：MPI/DCRS，□：OpenMP/DCRS）

（3）三次元弾性問題（PGA モデル）

図 26 は様々な色数で三次元弾性解析（PGA モデル（図 9））を実施した場合の結果である。ここでは，スカラープロセッサに特有な DCRS（■□）＞DJDS（●○），MPI（●■）＞OpenMP（○□）という傾向がより顕著である（全体としては，■＞●＞□＞○）。SR11000/J1 と J2 の性能比は，図 22，表 3 の割合と対応している。DCRS では色数の性能に対する影響は小さいが，DJDS では色数が増加すると，3.（2）で述べたようにキャッシュがより有効に利用できるため，性能が高くなる。MPI/DJDS（●）の性能は色数の増加によって向上し，OpenMP/DJDS（○）の場合も 300 色程度までは色数の増加によって性能が向上する。色数が 300 以上では，OpenMP のオーバーヘッドによる性能の低下が見られるが，図 11 に示す IBM SP3 の OpenMP/DJDS（○）の場合と比較すると低下の度合いは小さい。これも BYTE/FLOP 値の増加，チップ内にメモリコントローラ内蔵，および L2 キャッシュ，L3 キャッシュの効果によるものと考えられる。



（a） Hitachi SR11000/J1

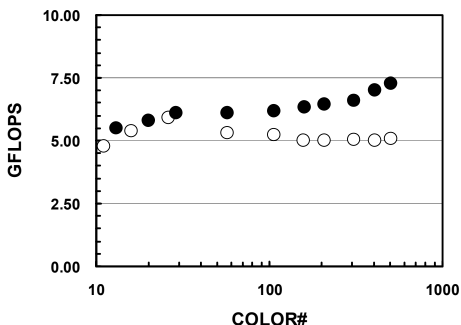


（b） Hitachi SR11000/J2

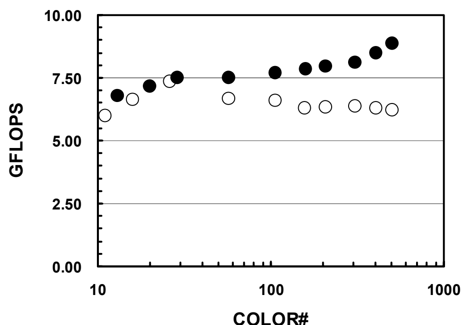
図 26　PGA モデルの計算結果（色数と性能の関係），並列プログラミングモデル，行列格納方法の影響（●：MPI/DJDS，○：OpenMP/DJDS，■：MPI/DCRS，□：OpenMP/DCRS）

（4）三次元接触問題

図 27 は様々な色数で三次元接触問題（西南日本モデル（図 13））の計算を実施した場合の結果である。係数行列格納法としては DJDS のみ考慮した。色数の性能に対する効果は PGA の場合と同様である。



（a） Hitachi SR11000/J1

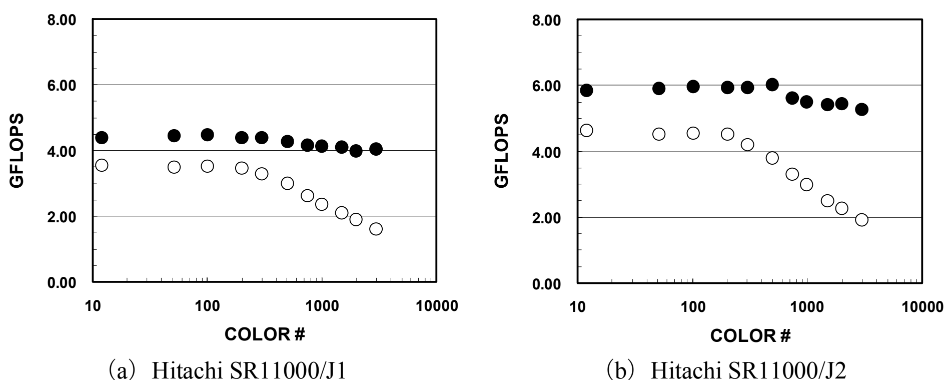


（b） Hitachi SR11000/J2

図 27　接触問題（西南日本モデル）の計算結果（色数と性能の関係），並列プログラミングモデルの影響（●：MPI/DJDS，○：OpenMP/DJDS）

（5）二重球殻間領域三次元ポアソン方程式

図 28 は，様々な色数について，図 16 に示す二重球殻間領域におけるポアソン方程式をマルチグリッド前処理付き CG 法（MGCG）で解いた場合の計算性能である。係数行列格納法としては DJDS のみ考慮した。色数の性能に対する効果は PGA，接触問題の場合と同様である。また，図 17 で示した ES，IBM SP3 と比較すると，OpenMP を適用した場合の色数の増加による性能低下の割合は少ない。色数が 12 色と 2000 色の場合を比較すると計算性能の比はそれぞれ 1.86（SR11000/J1），2.07（SR11000/J2）となっている。これは図 17 に示した SP3（3.90）の場合と比較すると大幅な改善である。これも BYTE/FLOP 値の増加，チップ内にメモリコントローラ内蔵，および L2 キャッシュ，L3 キャッシュの効果によるものと考えられる。



（a）Hitachi SR11000/J1　　（b）Hitachi SR11000/J2

図 28　二重球殻モデル（多重格子法）の計算結果（色数と性能の関係），並列プログラミングモデルの影響（●：MPI/DJDS，○：OpenMP/DJDS）

5. Hitachi 11000 における実行例（複数ノード）

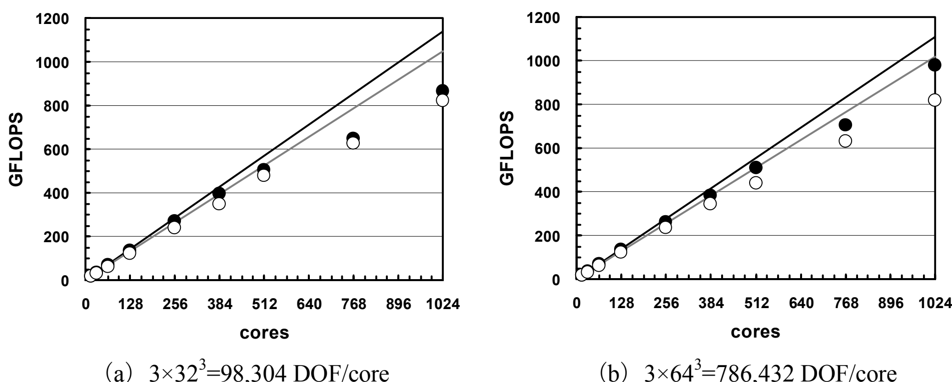
図 8 に示す三次元弾性問題（Cube モデル）について，SR11000/J2 において，コアあたりの問題規模を固定し，SMP ノード数を変化させた場合の計算（Weak Scaling）を実施した。係数行列格納法としては DCRS を適用した。各 SMP ノード 16 コアを使用して，1 コアあたり 98,304 DOF，786,432 DOF の場合について，64 ノード（1,024 コア）まで計算を実施した。1 ノードあたりの問題規模は，1,572,864 DOF（$=16\times3\times32^3$）および 12,582,912 DOF（$=16\times3\times64^3$）であり，最大問題規模は，100,663,296 および 805,306,368 DOF である。最高性能は 977 GFLOPS であり，これは SR11000/J2 の 64 ノード（1,024 コア）のピーク性能（9.42 TFLOPS）の 10.4%に相当する（図 29）。直線は，1 ノード（16 コア）における性能を基準とした，Flat MPI，Hybrid の場合の理想値である。図 30 は更にわかりやすいように 1 ノード（16 コア）からの計算性能増加率（Speed-Up）として表した図である。図 18～図 20 の場合と同様に，コアあたりの問題規模が小さい場合は，SMP ノード数が増加すると，通信のオーバーヘッドの影響が顕著となり，理想値との差が大きくなるが，問題規模が大きい場合は，オーバーヘッドの影響は比較的少ない。

ここで注意しなければならないのは，ES，IBM SP3 の場合と異なり，コアあたりの問題規模が大きい場合，SMP ノード数が増加すると，Flat MPI の方が Hybrid よりも計算性能増加率が大きくなっていることである。3．で述べた〔2,4,5,6〕の結果によると，並列有限要素法の場合，SMP ノード数が増加すると，MPI latency の影響が顕著となり，特に MPI プロセス数の多い Flat

MPI はその影響を受けやすいため，Hybrid と比較して，相対的に性能が大幅に低下するような現象が見られた。しかしながら図 29，図 30 の例ではこれとは逆の現象が生じている。

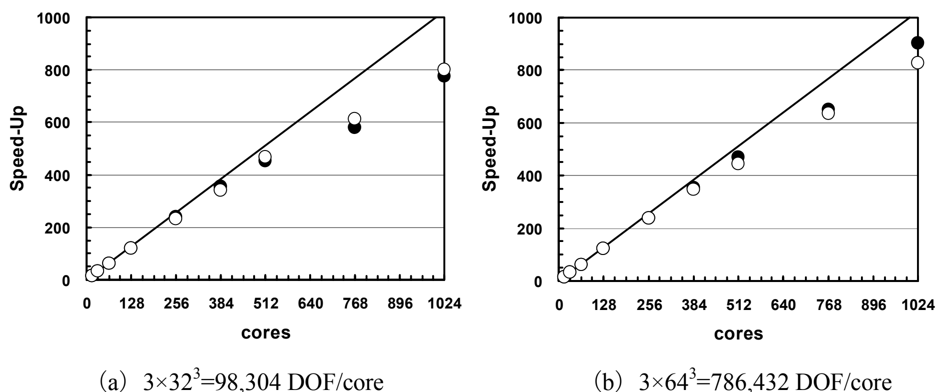
Hitachi SR11000/J2 では，SMP ノード間の通信用ポートとして各 SMP ノードに 2GB/sec のものが 6 本用意されている。Flat MPI の場合はこれが全て効率よく利用されているものと考えられる。Hybrid の場合，SMP ノードあたりの通信プロセスは図 31 に示すように一つであり，MPI_ISEND，MPI_IRECV の呼び出しの前後に送信バッファ（WS）へのコピー，受信バッファ（WR）からのコピーがあり，この部分は OpenMP によって並列化されている。

Hitachi SR11000/J2 では，Hybrid の場合は，使用するポート数は基本的に一つであるが，SMP ノード間通信量に応じて，使用ポートの数が動的に変化するような設定となっており，ポートあたりの通信量が 256KB を超える場合には，複数のポートに分割して送受信が行われる。有限要素法では領域間の通信量は比較的少ないが，コアあたり 3×64^3=786,432 DOF の場合は，各 SMP ノードの通信量は 4MB 弱，98,304 DOF（$=3\times32^3$）の場合は 1MB 弱のオーダーとなり，複数のポートが使用されている可能性がある。コアあたり（すなわち SMP ノードあたり）問題規模が大きくなり，SMP ノード間通信量が増加して，複数のポートが利用される場合の効果については，様々な問題規模，SMP ノード数における検討が必要である。



（a）3×32^3=98,304 DOF/core　（b）3×64^3=786,432 DOF/core

図 29 「Hitachi SR11000/J2」における Cube モデルの計算結果（Weak Scaling）（●：Flat MPI/DCRS，○：Hybrid/DCRS，－：Flat MPI/DCRS（ideal），－：Hybrid/DCRS （ideal））



（a）3×32^3=98,304 DOF/core　（b）3×64^3=786,432 DOF/core

図 30 「Hitachi SR11000/J2」における Cube モデルの計算結果，1 ノードからの計算性能増加率（Weak Scaling）（●：Flat MPI/DCRS，○：Hybrid/DCRS，－：ideal）

```
  do neib= 1, NEIBPETOT
    istart= STACK_EXPORT(neib-1)
    inum  = STACK_EXPORT(neib  ) - istart
!$omp parallel do private (k,ii)
    do k= istart+1, istart+inum
      ii   = 3*NOD_EXPORT(k)
      WS(3*k-2)= X(ii-2);WS(3*k-1)= X(ii-1);WS(3*k)= X(ii)
    enddo
!$omp end parallel do

    call MPI_ISEND (WS(3*istart+1), 3*inum, …)
  enddo

  do neib= 1, NEIBPETOT
    istart= STACK_IMPORT(neib-1)
    inum  = STACK_IMPORT(neib  ) - istart
    call MPI_IRECV (WR(3*istart+1), 3*inum, …)
  enddo

  call MPI_WAITALL (NEIBPETOT, req2, sta2, ierr)

  do neib= 1, NEIBPETOT
    istart= STACK_IMPORT(neib-1)
    inum  = STACK_IMPORT(neib  ) - istart
!$omp parallel do private (k,ii)
    do k= istart+1, istart+inum
          ii   = 3*NOD_IMPORT(k)
      X(ii-2)= WR(3*k-2);X(ii-1)= WR(3*k-1);X(ii)= WR(3*k)
    enddo
!$omp end parallel do
  enddo

  call MPI_WAITALL (NEIBPETOT, req1, sta1, ierr)
```

図 31　並列有限要素法（三次元弾性問題）の領域間通信（Hybrid 並列プログラミングモデル）（図 5，図 6 参照）

6.まとめ

本研究では，固体地球シミュレーション用並列有限要素法プラットフォーム「GeoFEM」において開発された並列有限要素法アプリケーションを元に整備した性能評価のためのベンチマークプログラム群（GeoFEM ベンチマーク）を使用して，Hybrid および Flat MPI の両プログラミングモデルについて，本学情報基盤センターの Hitachi SR11000/J2 の性能評価を実施した。

GeoFEM ベンチマークでは，①三次元弾性問題（Cube・PGA モデル），②三次元接触問題，③二重球殻間領域三次元ポアソン方程式，における並列前処理付き反復法ソルバーの実行性能（GFLOPS 値）を様々な条件下で計測する。プログラムは全て OpenMP ディレクティヴを含む FORTRAN90 および MPI で記述されており，SMP クラスタの性能評価に適している。様々なハードウェアに対応可能なように，連立一次方程式の係数マトリクスの格納法として，ベクトルプロセッサ向けの DJDS とスカラープロセッサ向けの DCRS の両方が使用可能である。

Hitachi SR11000/J2 では，IBM POWER5+　2 コアから構成されるチップ内にメモリコントローラが内蔵されており，かつ，大容量の L2 キャッシュ，L3 キャッシュを利用可能である。従来のスカラープロセッサ（IBM POWER3）を使用した IBM SP3 と比較して，高い BYTE/FLOP 値を示しており，スカラープロセッサ特有の問題規模の増加に伴う性能低下が非常に少なく，高い対ピーク性能比を示している。また，従来機種である Hitachi SR8000/MPP と比較すると，実効性能で約 4 倍の性能増加が得られている。

1 ノード内における MPI と OpenMP の差はほとんど無く，これもメモリコントローラ内蔵，大容量キャッシュの効果であると考えられる。OpenMP 使用時にマルチカラーオーダリングの色数を増加させることによる性能低下もほとんど見られず，DJDS によって係数行列を格納している場合には，逆にデータの局所性が増すことによる性能向上が顕著に見られた。特に，

MGCG 法による二重球殻間領域三次元ポアソン方程式のベンチマークの場合は，色数 2000 色まで増加させても， Hitachi SR11000/J2（OpenMP）で半分程度の性能低下に抑えられている。

複数ノードを使用した，Weak Scaling による性能評価については，これまでの「地球シミュレータ」，「IBM SP3」の性能評価から得られた知見によれば，SMP ノード数が増加した場合は Flat MPI よりも Hybrid の方が有利と考えられていた。これは，本研究で対象としている並列有限要素法においては，通信バンド幅（すなわち通信量）よりも，通信プロセス数（すなわちレイテンシ）に起因するオーバーヘッドの影響をより顕著に被るためである。

しかしながら，Hitachi SR11000/J2 の場合は，特にコアあたりの問題規模が大きい場合には，SMP ノード数が増加した場合, Flat MPI の方がむしろ Hybrid よりも性能が高いことがわかった。これは，Flat MPI と Hybrid において各 SMP ノード間の使用通信ポート数の決定法が異なり，Flat MPI では 6 本の通信ポート全てを常時使用するのに対して，Hybrid では通信量に応じて使用ポート数を調節していることと関連している可能性がある。更に詳細な検討が必要であるが，Flat MPI と Hybrid の選択にあたっては，こうしたハードウェアの特性を理解した上で，決定する必要がある。逆に，ハードウェア，特に大規模並列計算機のノード間ネットワークの設計にあたっては, アプリケーション特有の通信パターンを考慮することも重要であると考えられる。

1．でも触れたように，マルチコアプロセッサの普及によって，Flat MPI と Hybrid の優劣に関する議論は再び注目をあびつつある。現状では，いわゆるマルチコアプロセッサの能力は，本稿で紹介したハードウェアと比較して，特にメモリ関連の能力が劣るため，Hybrid，OpenMP では中々性能が出ない。ハードウェア能力の向上に期待する，というのも一つの考え方であるが，現状のハードウェアに適した，安定で効率の高い手法（特に反復法のための並列前処理手法）の開発は重要である。

謝 辞

本研究は, 東京大学 21 世紀 COE プログラム「多圏地球システムの進化と変動の予測可能性」，および科学技術振興機構戦略的創造研究推進事業（CREST）の補助を受けている。計算機環境を提供いただいた東京大学情報基盤センター，地球シミュレータセンターおよび Lawrence Berkeley National Laboratory に謝意を表する。

参 考 文 献

[1] Rabenseifner, R. (2002) Communication Bandwidth of Parallel Programming Models on Hybrid Architectures. Lecture Notes in Computer Science 2327, 437-448

[2] Nakajima, K. (2003) Parallel Iterative Solvers of GeoFEM with Selective Blocking Pre-conditioning for Nonlinear Contact Problems on the Earth Simulator. ACM/IEEE Proceedings of SC2003

[3] 奥田洋司，中島研吾　共編 (2004)「並列有限要素解析〔I〕クラスタコンピューティング」，培風館

[4] Nakajima, K. (2004) Preconditioned Iterative Linear Solvers for Unstructured Grids on the Earth Simulator, IEEE Proceedings of 7th International Conference on High Performance Computing and Grid in Asia Pacific Region (HPC Asia 2004), 150-169

[5] 中島研吾（2005） SMPクラスタ型並列計算機におけるプログラミングモデル：Flat MPI vs. Hybrid，京都大学学術情報メディアセンター全国共同利用版広報5-2, 2-10

[6] Nakajima, K. (2005) Parallel programming models for finite-element method using preconditioned iterative solvers with multicolor ordering on various types of SMP cluster, IEEE Proceedings of 8th International Conference on High Performance Computing and Grid in Asia Pacific Region (HPC Asia 2005), 83-90

[7] 中島研吾（2006）GeoFEMベンチマークによる Hitachi SR11000／J1およびIBM p5-595のノード性能評価, 情報処理学会研究報告 2006-HPC-105-11, 61-66

[8] Saad, Y. (2003) “Iterative Methods for Sparse Linear Systems 2nd Edition”, SIAM

[9] Doi, S. and Washio, T. (1999) Using Multicolor Ordering with Many Colors to Strike a Better Balance between Parallelism and Convergence. Proceedings of RIKEN Symposium on Lin-ear Algebra and its Applications, 19-26

[10] Simon, H.D. (1991) Partitioning of unstructured problems for parallel processing, Computing Systems in Engineering 2, 135-148

[11] Carter, J., Oliker, L., and Shalf, J. (2007) Performance Evaluation of Scientific Applications on Modern Parallel Vector Systems, Lecture Notes in Computer Science 4395, 490-503

[12] Nakajima, K. (2005) Three-Level Hybrid vs. Flat MPI on the Earth Simulator: Parallel Iterative Solvers for Finite-Element Method, Applied Numerical Mathematics 54, 237-255

[13] Kerbyson, et al. (2002) A Comparison Between the Earth Simulator and Alpha Server Systems using Predictive Application Performance Models, LA-UR-02-5222, Los Alamos National Laboratory

[14] 青木秀貴，中村友洋，助川直伸，齋藤拡二，深川正一，中川八穂子，五百木伸洋（2005）スーパーテクニカルサーバーSR11000 モデル J1 のノードアーキテクチュアと性能評価，情報処理学会論文誌：コンピューティングシステム Vol.45 No.SIG12（ACS11），27-36

実対称固有値問題に対する多分割の分割統治法のSR11000への一実装

桑 島 豊，坪 谷 怜，田 村 純 一，重 原 孝 臣
埼玉大学理工学研究科

1. はじめに

本稿でとり扱う固有値問題は，量子力学，量子化学，建築，経済など様々な分野で現れる問題である。量子力学，量子化学においてはシュレーディンガー方程式として陽に現れ，建築の分野においては建物の振動を扱う場合などに，経済の分野では市場の安定性に関する指標を与えるためなどに現れ，これら以外の分野においてもその他多様な場面で必要となる。また，基本的な統計解析手法として広く用いられている主成分分析は，固有値計算が主要な部分を占める。大規模な問題を扱う分野も多く，解法の高速化の需要は大きい。そのため，並列計算機を有効に活用できる固有値問題解法が重要である。

固有値問題解法の 1 つである 2 分割の分割統治法は，1 つの CPU 上であっても高速な解法であり，また本質的に並列計算に向く性質を持つ。その 2 分割の分割統治法を，本質的な並列性を損なうことなく，演算回数を減らすことにより，さらなる速度の向上を目指すべく拡張したアルゴリズムが，多分割の分割統治法である。このアルゴリズムは，未だ解決すべき点を残すものの，逐次計算では，LAPACK [*1]に実装されている 2 分割の分割統治法と同等程度の速度で計算できるという結果が得られている。

本稿では，我々が提案した「実対称行列の固有値問題に対する多分割の分割統治法」の並列計算機への実装の第一段階として，SR11000 の 1 ノード上における実装について紹介する。SR11000 において，共有メモリ型の並列化による実装を行い，重点的に並列化を行い効果をあげた箇所や，実行速度による他の固有値問題解法との比較など，現状で得られている結果を報告する。

本原稿の構成は以下の通り。2 節では，実対称固有値問題の数値解法の流れと，代表的な数値解法について述べる。2 節で述べた数値解法の中で本稿で主に取り扱う分割統治法を，3 節では 2 分割について，4 節では多分割について少し詳細に説明する。5 節では並列化の方法の種類と，本研究で用いた並列化の方法について述べる。6 節では数値実験の方法，結果とその考察について述べる。7 節はまとめを行い，8 節で今後の課題について述べる。9 節では，実験中に遭遇した事例などを紹介する。

2. 固有値問題の数値解法

本節では，まず固有値問題を数学的側面から紹介し，その後一般的な固有値問題の数値解法の流れを説明し，代表的な数値解法の簡単な紹介とそれぞれの比較を行う。

[*1] 標準的な行列計算ライブラリの一つ。ソースが無料で公開されている (www.netlib.org/lapack/)。SR11000 では，最適化されたライブラリが提供されている。

2.1　固有値問題の数学的な定式化

この小節では，以降必要となる用語の説明を兼ねて，固有値問題を数学的に述べる。

行列に対する固有値問題とは，n 次複素正方行列 A に対して，

$$A\boldsymbol{q} = \lambda \boldsymbol{q}, \quad \boldsymbol{q} \neq \boldsymbol{o} \tag{1}$$

を満たす複素数 λ と n 次複素ベクトル $\boldsymbol{q}$ の対を求める問題である。λ は固有値，$\boldsymbol{q}$ は固有ベクトルと呼ばれる。

特に，A を実対称行列に限れば，固有値として（重複を含めて）n 実数 $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$ を持ち，それぞれに対応して $A\boldsymbol{q}_j = \lambda_j \boldsymbol{q}_j$, $(\boldsymbol{q}_i, \boldsymbol{q}_j) = \delta_{ij}$ を満たす固有ベクトル $\boldsymbol{q}_1, \boldsymbol{q}_2, \cdots, \boldsymbol{q}_n$ が存在する。ただし，δ_{ij} はクロネッカのデルタで，$i = j$ なら 1，$i \neq j$ なら 0 である。

これらを行列形式にまとめると，直交行列 $Q = (\boldsymbol{q}_1\ \boldsymbol{q}_2\ \cdots\ \boldsymbol{q}_n)$，実対角行列 $\Lambda = \mathrm{diag}(\lambda_1, \lambda_2, \cdots, \lambda_n)$ を用いて，$AQ = Q\Lambda$ であり，従って $A = Q\Lambda Q^T$ という対角化が得られる。逆に，直交行列 $Q = (\boldsymbol{q}_1\ \boldsymbol{q}_2\ \cdots\ \boldsymbol{q}_n)$ と実対角行列 $\Lambda = \mathrm{diag}(\lambda_1, \lambda_2, \cdots, \lambda_n)$ を用いて，実対称行列 A が $A = Q\Lambda Q^T$ と分解できれば，Λ の対角成分 λ_j が A の固有値で，Q の各列ベクトル $\boldsymbol{q}_j$ が対応する固有ベクトルである。

2.2　実対称行列の実対称三重対角行列化

実対称行列の固有値問題を数値的に解く場合には，実対称行列のまま計算を開始することは実用的でなく，前処理を行い，行列を簡単化した後に固有値計算を行う。一般的な前処理は，実対称行列を実対称三重対角行列に相似変換するというもので，三重対角化と呼ばれる。実対称三重対角行列とは，対角成分とその両隣の成分以外が全て 0 の行列である。

三重対角化には，**ハウスホルダー法**が数値的に安定で，よく用いられる。ハウスホルダー法は，ハウスホルダー行列 $H = I - 2\boldsymbol{u}\boldsymbol{u}^T$, $(\|\boldsymbol{u}\|_2 = 1)$ を用いる方法である。$H = H^T$, $H^T H = H^2 = I$ より，H は対称な直交行列である。ただし，I は単位行列。3 次実対称行列

$$A = \begin{pmatrix} a_1 & b_1 & c_1 \\ b_1 & a_2 & b_2 \\ c_1 & b_2 & a_3 \end{pmatrix} \tag{2}$$

に対して三重対角化を行う場合は，ハウスホルダー行列を $H = I - 2\boldsymbol{u}\boldsymbol{u}^T$,

$$\boldsymbol{u} = \frac{\boldsymbol{u}'}{\|\boldsymbol{u}'\|_2}, \quad \boldsymbol{u}' = \begin{pmatrix} 0 \\ b_1 - b_1' \\ c_1 \end{pmatrix}, \quad b_1' = \sqrt{b_1^2 + c_1^2} \tag{3}$$

と選ぶことで，

$$T = HAH = \begin{pmatrix} a_1 & b_1' & 0 \\ b_1' & a_2' & b_2' \\ 0 & b_2' & a_3' \end{pmatrix} \tag{4}$$

と三重対角化される。4 次以上の実対称行列に対しても，これと同様のハウスホルダー行列による変換を繰り返すことで，三重対角化することができる。

以下で紹介する実対称固有値問題解法は，全て実対称三重対角行列に対して適用される。

表 1　実対称三重対角行列の固有値問題解法の比較

解法	1) 理論	2) 精度	3) 演算量	4) 並列性	5) 行列積	6) 領域
2 分割の分割統治法	○	○	$O(n^2)$ $\sim (4/3)n^3$	○	○	$O(n^2)$
QR 法	○	○	$6n^3$	×	×	$O(n)$
二分法・逆反復法	○	○	$O(n^2)$ $\sim O(n^3)$	×	×	$O(n)$
MRRR 法	△	△	$O(n^2)$	△	×	$O(n)$
k 分割の分割統治法	△	○	$O(n^2)$ $\sim (2k/(k^2-1))n^3$	○	○	$O(n^2)$

2.3　代表的な実対称三重対角行列の固有値問題の数値解法の紹介と比較

本節では，代表的な実対称三重対角行列の固有値問題の数値解法を紹介し，それらの解法と多分割の分割統治法をいくつかの観点から比較を行う。

代表的な解法として，本稿で取り扱う分割統治法と，その他に MRRR 法，QR 法，二分法・逆反復法を挙げる。MRRR 法は比較的新しい解法であり，QR 法，二分法・逆反復法は古典的な方法である。

2.3.1　比較の観点

それぞれの観点の意味づけは以下の通り。

1) 理論は，解法に理論的，数学的に確立しているかを意味している。

2) 精度は，各解法で求められる結果の精度を意味している。無限精度演算であれば 0 になるべき，相対残差と直交誤差を評価基準とした。

3) 演算量は，n 次行列の問題を解く際に必要な浮動小数点演算の理論的な回数を表している。現在の計算機において，この値が速度と比例するとは限らない。

4) 並列性は，解法が並列計算機に向いた性質を持っているかを表している。

5) 行列積は，解法に行列積演算が使用されているかを表し，6) 領域は，n 次行列の問題を解く際に必要なメモリ領域を表している。この 2 つは密接な関連があり，行列積が必要な場合には $O(n^2)$ のメモリ領域が必要となる。しかし，並列計算機を含む現在の計算機において，行列積演算は高速に計算可能である。このことは，分割統治法を紹介する上で大きな意味を持つため，次の小々節で少し詳しく述べる。

また，2.3.3 小々節からは，以上の観点を含めて，表 1 にある 5 つの解法を紹介する。

2.3.2　行列積

行列同士の積を，本稿では行列積と呼ぶ。行列積は，本稿で取り扱う固有値問題に対する分割統治法において，演算回数の面で支配的であり，この計算の速度が分割統治法の速度に直結する。

a_{ij} を (i,j) 成分に持つ n 次正方行列 A と，b_{ij} を (i,j) 成分に持つ n 次正方行列 B との積

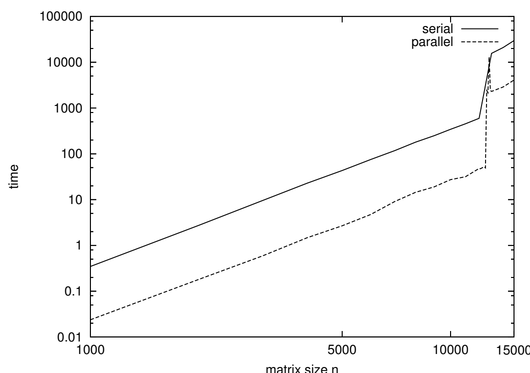


図 1　SR11000 における行列積の計算時間

$C = AB$ の (i, j) 成分 c_{ij} は，

$$c_{ij} = \sum_{l=1}^{n} a_{il} b_{lj} \tag{5}$$

と定義されるので，一成分を求めるための演算は加算と乗算がそれぞれ n 回で，全成分を求めるためには合計 $2n^3$ 回の浮動小数点演算が必要である。これを本稿では，演算量が $2n^3$ であると言う。

これを実際に SR11000 上で検証した結果が図 1 である。このグラフは，n 次行列同士の積の計算にかかる時間を表している。行列積計算には，提供されている BLAS ライブラリ*2を用いた。両対数グラフで，横軸は行列の次数 n で，縦軸は実行時間（単位は秒）である。実線は，1 CPU による逐次実行であり，破線は，16 CPU による並列計算である。このグラフを見ると，逐次，並列ともに $n = 12000$ 程度までは，n^3 に比例する実行時間であり，理論的な演算量評価と一致する。ただし，$n = 12000$ 以降において，実行時間が急激に増加している。この現象の原因は，現在究明中である。

また，$n = 5000$ では，逐次計算で 43.05 秒，並列計算で 2.67 秒で，並列計算は逐次計算の 1/16 程度の時間で終了しており，最大限に並列化がなされている。これは，行列積が並列計算機上で非常に高速であることを示している。ただし，n が 7000 以上，例えば $n = 10000$ では，逐次計算で 342 秒，並列計算で 27 秒で，13 倍弱の速度向上であり，必ずしも並列計算機上で非常に高速とはいえないという結果も得られている。この現象の原因も，現在究明中である。

なお，本原稿で示す数値実験で行列積計算を行う行列は，16 倍の高速化がなされている範囲の次元である。

2.3.3　2 分割の分割統治法と多分割の分割統治法

ここでは，実対称三重対角行列の固有値問題に対する 2 分割の分割統治法と，それを我々が拡張した多分割の分割統治法のアイデアを述べる。詳しい説明は後述する。一般的な意味での 2 分割の分割統治法は，大きな問題を解くために，2 つの小問題に分割し，その小問題の解を利用する，という方法である。実対称三重対角行列の固有値問題に適用した場合，まず 2 つの小さ

*2 LAPACK の下位ルーチンを集めたライブラリ。行列積など基本的な演算を行う。SR11000 では，LAPACK 同様最適化されたライブラリが提供されている。

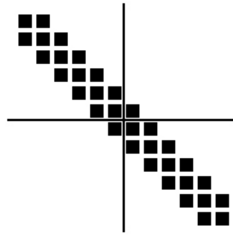


図 2　2 分割の場合の分割の方法

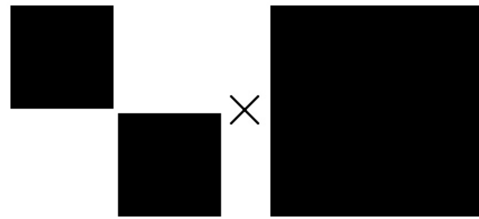


図 3　2 分割の場合の行列積

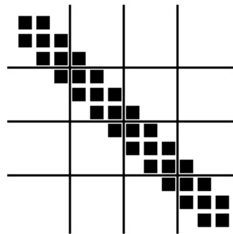


図 4　4 分割の場合の分割の方法

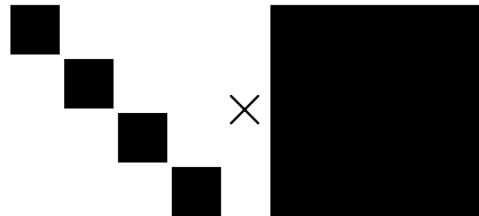


図 5　4 分割の場合の行列積

な実対称三重対角行列の固有値問題とわずかな残りの部分に分割する。図 2 は，分割の方法を模式的に表した図である。この図で，黒い部分はある要素で，白い部分は要素が零を意味する。1 つの実対称三重対角行列を，左上と右下の 2 つの半分の大きさの実対称三重対角行列と，わずかな余りの部分に分けている。

2 つの小さな実対称固有値問題を解を利用することにより，元の問題を別の固有値問題へと変換できる。変換後の問題は，変換前の問題と比較して容易に解くことが可能である。

最後に，元の問題を解くために，図 3 の形をした行列同士の積を計算する。分割統治法では，この行列積計算が演算量の面で支配的な部分を占める。左の行列の半分は零であるから，全てが零でない行列である場合の半分の演算量ですむ。

2 つに分割したために，左の行列の半分の要素が零となったのであるから，分割の数を増やせば，より零の部分を増やすことができ，行列積計算に必要な演算量が減少すると期待される。それは，実際に正しい。すなわち，4 分割ならば，図 5 のような形となり，その行列積は 2 分割の場合の半分の演算量となる。しかし，4 分割にした場合には，図 4 のように，分解時に発生する余りの部分が 2 分割の場合の 3 倍に増大し，その部分を処理するための演算量が増加するという問題は存在する。

2 分割の場合の演算量は，行列積が支配的であるため，行列積の演算量自体の $(4/3)n^3$ となる。k 分割であっても，行列積が支配的な分割数の範囲ならば，同様に $(2k/(k^2-1))n^3$ となる。また，行列積を用いるため，必然的に $O(n^2)$ のメモリ領域が必要となる。

また，2 分割，多分割ともに，いくつか分割した小さな実対称固有値問題は独立なため，この部分は並列に計算できる。

多分割の分割統治法には，最適な分割数の決定法など，未だ理論面での課題が残っている。

2.3.4　QR 法

正則な n 次行列 A に対して，直交行列 Q と右上三角行列 R との積 $A=QR$ への分解は QR 分解と呼ばれる。この分解は，A を構成する列ベクトル $\boldsymbol{a}_1, \boldsymbol{a}_2, \cdots, \boldsymbol{a}_n$ に対するグラム・シュ

ミットの直交化法

$$
\begin{aligned}
&\boldsymbol{q}'_1 := \boldsymbol{a}_1 \\
&\boldsymbol{q}_1 := \boldsymbol{q}'_1/\|\boldsymbol{q}'_1\|_2 \\
&\textbf{for}\ \ i := 2\ \textbf{to}\ n \\
&\qquad \boldsymbol{q}'_i := \boldsymbol{a}_i - \textstyle\sum_{j=1}^{i-1}(\boldsymbol{a}_i, \boldsymbol{q}_j)\boldsymbol{q}_j \\
&\qquad \boldsymbol{q}_i := \boldsymbol{q}'_i/\|\boldsymbol{q}'_i\|_2 \\
&\textbf{end for}
\end{aligned}
$$

により構成できる。このアルゴリズムの結果は，

$$
(\boldsymbol{a}_1\ \boldsymbol{a}_2\ \cdots\ \boldsymbol{a}_n) = (\boldsymbol{q}_1\ \boldsymbol{q}_2\ \cdots\ \boldsymbol{q}_n)
\begin{pmatrix}
\|\boldsymbol{q}'_1\|_2 & (\boldsymbol{a}_2, \boldsymbol{q}_1) & (\boldsymbol{a}_3, \boldsymbol{q}_1) & \cdots & (\boldsymbol{a}_n, \boldsymbol{q}_1) \\
0 & \|\boldsymbol{q}'_2\|_2 & (\boldsymbol{a}_3, \boldsymbol{q}_2) & \cdots & (\boldsymbol{a}_n, \boldsymbol{q}_2) \\
\vdots & \ddots & \ddots & \ddots & \vdots \\
0 & 0 & 0 & \|\boldsymbol{q}'_{n-1}\|_2 & (\boldsymbol{a}_n, \boldsymbol{q}_{n-1}) \\
0 & 0 & 0 & 0 & \|\boldsymbol{q}'_n\|_2
\end{pmatrix}
\tag{6}
$$

と行列形式で表すことができ，これにより A の QR 分解が完了する。

この QR 分解に基づいて，正則な n 次行列 A の固有値問題を解くための方法が QR 法であり，基本的なアルゴリズムは，擬似コードを用いて以下のように記述できる。$A_1 = A$ を入力とし，

$i := 1$ に初期化

do

A_i を $A_i = Q_i R_i$ と QR 分解する

$A_{i+1} := R_i Q_i$ を計算する

while i を 1 づつ増やしながら，収束するまで繰り返す

アルゴリズムで，A_i の QR 分解の逆順の積で計算される A_{i+1} は，構成法より $A_{i+1} = R_i Q_i = Q_i^T A_i Q_i$ であり A_i と同じ固有値をもつ。$A_1, A_2, \cdots$ の極限は，対角線より下の要素が全て 0 に収束し，そのとき対角成分は A の固有値となることが知られている。

A が三重対角行列の場合，$A_i, (i = 1, 2, \cdots)$ も三重対角行列であるため，反復計算が可能である。この他，いくつかの高速化手法を取り入れた，実用的な QR 法の演算量は $6n^3$ である。

2.3.5 二分法・逆反復法

二分法は固有値を求める方法であり，逆反復法は固有ベクトルを求める方法である。これらは独立な方法であるが，併せて用いることが多いため，まとめて紹介する。

二分法　　二分法は，実対称三重対角行列 T に対して適用することができ，T の固有値を一部，もしくは全て求める方法である。この解法では，T に依存した関数

$$
N(x) \equiv x \text{ 以上の } T \text{ の固有値の数} \tag{7}
$$

を定義する。この関数は単調に減少するから，**二分探索**を用いることで，固有値を求めることができる。それぞれの固有値は独立に求められるため，固有値の計算を並列化することができる。$N(x)$ の計算は $O(n)$ の演算量であるので，二分法で n 個の固有値を求めるための演算量は，$O(n^2)$ である。

続いて，二分法の中核をなす $N(x)$ の計算法を述べる。T_k を T の k 次首座小行列とする。k

次首座小行列とは，左上の $k \times k$ を切り出した行列である。また，$p_k(x)$ を

$$\begin{cases} p_0(x) = 1 \\ p_k(x) = \det(xI - T_k), \quad (k = 1, 2, \cdots, n) \end{cases} \tag{8}$$

とする。このとき，$p_0(x), p_1(x), p_2(x), \cdots, p_n(x)$ はスツルム列をなす。従って，スツルム列の理論より，$N(x)$ は $p_0(x), p_1(x), p_2(x), \cdots, p_n(x)$ の符号の変化した数と一致する。

$n = 2$ の場合，

$$T = \begin{pmatrix} a & b \\ b & c \end{pmatrix} \tag{9}$$

を例に挙げる。このとき，$p_0(x) = 1, p_1(x) = x - a, p_2(x) = (x-a)(x-c) - b^2$ であるため，以下の表が書ける。

x	$\cdots$	λ_1	$\cdots$	a	$\cdots$	λ_2	$\cdots$
$p_0(x)$	$+$	$+$	$+$	$+$	$+$	$+$	$+$
$p_1(x)$	$-$	$-$	$-$	0	$+$	$+$	$+$
$p_2(x)$	$+$	0	$-$	$-$	$-$	0	$+$
$N(x)$	2		1	1	1		0

この表より，ここで構成した $N(x)$ は，(7) で定義した関数となっていることがわかる。

逆反復法　一方，逆反復法は，与えられた固有値に対応する固有ベクトルを求める方法である。

逆反復法を説明するための準備として，基本となる，べき乗法を説明する。べき乗法は，正方行列 A に対して，絶対値が最大の固有値に対応する固有ベクトルを求める方法である。擬似コードでは，A を入力として，

初期ベクトル $\boldsymbol{x}_1$ を，長さ 1 のベクトルにとる
$i = 1$ に初期化
do
　$\boldsymbol{x}'_{i+1} := A\boldsymbol{x}_i$
　$\boldsymbol{x}_{i+1} := \boldsymbol{x}'_{i+1}/\|\boldsymbol{x}'_{i+1}\|_2$ と正規化
while $\boldsymbol{x}_i$ が収束するまで繰り返す

となる。このアルゴリズムにおいて，$\boldsymbol{x}_i$ が A の絶対値が最大の固有値に対応する固有ベクトルに収束する。

ここで，A の十分な精度の近似固有値 $\hat{\lambda}$ が求まっているとき，べき乗法を用いて，対応する固有ベクトル $\boldsymbol{q}$ を求めることを考える。$\tilde{A} = (A - \hat{\lambda}I)^{-1}$ とすれば，$\tilde{A}$ の絶対値最大の固有値に対応する固有ベクトルは，$\boldsymbol{q}$ に一致する。そのため，$\tilde{A}$ に対してべき乗法を適用することにより，$\boldsymbol{q}$ を求めることができる。このアルゴリズムが逆反復法である。

実際には，$(A - \hat{\lambda}I)^{-1}$ を陽に求めて，$\boldsymbol{x}_{i+1} = (A - \hat{\lambda}I)^{-1}\boldsymbol{x}_i$ を計算するのではなく，連立一次方程式 $(A - \hat{\lambda}I)\boldsymbol{x}_{i+1} = \boldsymbol{x}_i$ を解くことで $\boldsymbol{x}_{i+1}$ を求める。この解法で固有ベクトルを 1 つ求めるための演算量は $O(n)$，n 本全て求めるためには $O(n^2)$ の演算量である。

逆反復法は，それぞれの固有ベクトルが独立に求められるため，並列計算することができる。しかし，大規模な行列に対する数値計算では，直交するべき固有ベクトルが互いに直交せず，直

交性を確保するために，それらを事後に直交させる，再直交化をしなければならないことが多い。再直交化は並列計算に向かない。そのため，大規模な問題に対しては，$O(n^3)$ の演算量が必要であることが多い。

2.3.6 MRRR 法

MRRR (Multiple Relatively Robust Representations) 法は，紙面の都合で簡単に紹介する。MRRR 法は，最新の固有値問題解法の一つで，演算量 $O(n^2)$ で全ての固有値と固有ベクトルを求めることができることが特長である。新しい解法であるため，未だ理論面に不十分な面がある。各固有ベクトルを並列に求めることが可能であるとされるが，SR11000 のライブラリにある MRRR 法では，並列計算と逐次計算の実行時間の比がほぼ 1 であるため，ここでは並列性が低いとした。[山本，2005] に詳しい。

3. 2 分割の分割統治法

この節では，実対称行列に対する 2 分割の分割統治法について述べる。このアルゴリズムは，本稿でとり扱う多分割の分割統治法の基礎となっているアルゴリズムである。

3.1 メタアルゴリズムとしての分割統治法

一般に，分割統治法は以下の 3 つのステップからなる**メタアルゴリズム**[*3]である。

proc MetaDC ······ 問題 P を分割統治法で解く（k 分割）

MetaDC-1　問題 P を k 個の小さな問題 $P_1, P_2, \cdots, P_k$ に k 分割する。
（ここで，k を分割数と呼ぶことにする。）

MetaDC-2　小さな問題 $P_1, P_2, \cdots, P_k$ をそれぞれ解く。

MetaDC-3　小さな問題の解を利用して，大きな問題 P を解く。

ここで，P_j が P とサイズが異なっているだけで，P と同様の問題であるときにのみ，このメタアルゴリズムを用いることができる。同様の問題であるので，P_j のそれぞれについてもメタアルゴリズム MetaDC を再帰的に用いることができる。ここに，分割統治法の本質的な並列性が存在する。また，高速化のために，P_j の規模が十分小さいときには，再帰的に計算をすることなく，より単純な別の解法を用いて P_j を解くことが一般的である。

整列（ソーティング）問題が代表的な適用例で，その中で**クイックソート**がよく知られている。

3.2 実対称行列に対する 2 分割の分割統治法

前小節で述べた，メタアルゴリズムとしての分割統治法を，分割数が 2 の場合に，実対称三重対角行列の固有値問題に適用した解法を紹介する。このアルゴリズムは，Cuppen によって 1981 年に提案された。

T は対角成分が a_i $(i = 1, 2, \cdots, n)$，副対角成分が b_i $(i = 1, 2, \cdots, n-1)$ であるとする。た

[*3] 問題に依存しない形式で記述されたアルゴリズム。

だし，ある b_i が 0 のとき T は可約のため，一般性を失うことなく $b_i \neq 0$ を仮定する。

まず，[MetaDC-1] のステップとして実対称三重対角行列 T を 2 つの実対称三重対角行列に分割する。$n=6$ の場合を例にして説明すると，

$$
T = \left(\begin{array}{ccc|ccc} a_1 & b_1 & & & & \\ b_1 & a_2 & b_2 & & & \\ & b_2 & a_3 & b_3 & & \\ \hline & & b_3 & a_4 & b_4 & \\ & & & b_4 & a_5 & b_5 \\ & & & & b_5 & a_6 \end{array}\right)
$$
$$
= \left(\begin{array}{ccc|ccc} a_1 & b_1 & & & & \\ b_1 & a_2 & b_2 & & & \\ & b_2 & a_3 - b_3 & & & \\ \hline & & & a_4 - b_3 & b_4 & \\ & & & b_4 & a_5 & b_5 \\ & & & & b_5 & a_6 \end{array}\right) + b_3 \left(\begin{array}{c} 0 \\ 0 \\ 1 \\ \hline 1 \\ 0 \\ 0 \end{array}\right) \left(\begin{array}{ccc|ccc} 0 & 0 & 1 & 1 & 0 & 0 \end{array}\right) \tag{10}
$$

と分割する。これは，3 次実対称三重対角行列 T_1, T_2 を用いて，

$$
T = \left(\begin{array}{c|c} T_1 & \\ \hline & T_2 \end{array}\right) + b_3 \boldsymbol{v}\boldsymbol{v}^T, \quad \boldsymbol{v} = \left(\begin{array}{ccc|ccc} 0 & 0 & 1 & 1 & 0 & 0 \end{array}\right)^T \tag{11}
$$

と書ける。

次に，[MetaDC-2] のステップとして，分解した T_1, T_2 の固有値問題を解く。すなわち，T_1, T_2 の固有値問題 $T_i = P_i D_i P_i^T$ の解，実対角行列 D_1, D_2，直交行列 P_1, P_2 を求める。

最後に，[MetaDC-3] のステップとして，[MetaDC-2] の結果を利用して，T の固有値問題を解く。T_1, T_2 の固有値問題の解 D_1, D_2，P_1, P_2 を利用して，T は，

$$
\begin{aligned}
T &= \begin{pmatrix} P_1 D_1 P_1^T & \\ & P_2 D_2 P_2^T \end{pmatrix} + b_3 \boldsymbol{v}\boldsymbol{v}^T \\
&= \begin{pmatrix} P_1 & \\ & P_2 \end{pmatrix} \left\{ \begin{pmatrix} D_1 & \\ & D_2 \end{pmatrix} + b_3 \boldsymbol{u}\boldsymbol{u}^T \right\} \begin{pmatrix} P_1^T & \\ & P_2^T \end{pmatrix} \\
&\equiv P(D + b_3 \boldsymbol{u}\boldsymbol{u}^T) P^T
\end{aligned} \tag{12}
$$

と相似変形できる。ただし，実対角行列 $D \equiv D_1 \oplus D_2$，直交行列 $P \equiv P_1 \oplus P_2$，

$$
\boldsymbol{u} \equiv \begin{pmatrix} P_1^T & \\ & P_2^T \end{pmatrix} \boldsymbol{v} = \begin{pmatrix} P_1^T \text{の最終列} \\ P_2^T \text{の第 1 列} \end{pmatrix} \tag{13}
$$

このとき，実対角行列と階数 1 の摂動の和 $D + b_3 \boldsymbol{u}\boldsymbol{u}^T$ の固有値問題 $D + b_3 \boldsymbol{u}\boldsymbol{u}^T = Q \Lambda Q^T$ を解くことで

$$
T = PQ\Lambda Q^T P^T \tag{14}
$$

となる。これで，T の全ての固有値と固有ベクトルが求められた。

ここで，P と Q の積は，

$$
PQ = \left(\begin{array}{ccc|ccc} * & * & * & & & \\ * & * & * & & & \\ * & * & * & & & \\ \hline & & & * & * & * \\ & & & * & * & * \\ & & & * & * & * \end{array}\right) \begin{pmatrix} * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \end{pmatrix} \tag{15}
$$

の形式をしている（$*$ はある実数，空白は 0 を意味する）。この行列積は，通常の 6 次行列同士の積の半分の演算量で済む。これは，n 次行列でも同様であり，演算量は n^3 である。

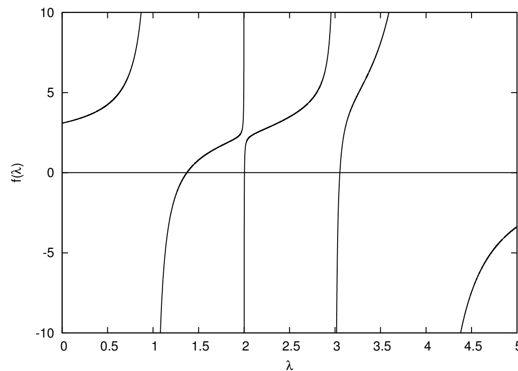


図 6　$f(\lambda)$ の例

実対角行列と階数 1 の摂動の和の固有問題を解くための演算量は，後述するように $O(n^2)$ である。また，メタアルゴリズムとしても述べたとおり，T_1, T_2 の固有値問題は，このアルゴリズムにより（再帰的に）解くことができる。

そのため，この素朴な 2 分割の分割統治法の演算量は，$(4/3)n^3$ である。すなわち，演算量の大部分を行列積が占めている。2.3.2 小節で述べたように，行列積は現在の計算機で高速に計算が可能であるため，2 分割の分割統治法も高速に計算される。

3.3　実対角行列と階数 1 の摂動の和の固有値問題

前小節で述べたように，分割統治法で実対称固有値問題を解くためには，実対角行列と階数 1 の摂動の和 $D + c\boldsymbol{u}\boldsymbol{u}^T$ の固有値問題を解かなければならない。

この行列の固有値は，$D + c\boldsymbol{u}\boldsymbol{u}^T$ の特性方程式を変形した，

$$f(\lambda) \equiv \frac{1}{c} - \sum_{j=1}^{n} \frac{u_j^2}{\lambda - d_j} \tag{16}$$

の零点を求める問題に帰着することができる。この非線形関数 $f(\lambda)$ の $n = 4$ の場合の例が，図 6 である。

この関数 $f(\lambda)$ の零点，すなわち $D + c\boldsymbol{u}\boldsymbol{u}^T$ の固有値は，変形ニュートン法によって求めることができる。n 次の問題に対するこの解法の演算量は，1 つの固有値のために $O(n)$，n 個の固有値を求めるためには $O(n^2)$ である。また，$f(\lambda)$ の零点は，その性質を利用して n 個それぞれを独立に求めることができ，ここに高い並列性が存在する。

一方，固有値 λ に対応する（ノルムが 1 の）固有ベクトルは

$$\frac{(\lambda I - D)^{-1}\boldsymbol{u}}{\|(\lambda I - D)^{-1}\boldsymbol{u}\|_2} \tag{17}$$

で求められる。この方法による，n 次行列の全ての固有ベクトルを求めるために必要な演算量は $O(n^2)$ である。なお，実装においては，数値的に精度を高めるための前処理が追加される。この処理が追加されても演算量が $O(n^2)$ であることは変わらない。

以上により，n 次元の実対称行列と階数 1 の摂動の固有値と固有ベクトルは，$O(n^2)$ の演算量で全て求められる。

3.4 アルゴリズム

ここまでのまとめとして，2 分割の分割統治法を擬似コードで表す。

アルゴリズム　分割統治法 (2 分割) を用いて，n 次実三重対角行列 T の固有値・固有ベクトルを求める:

proc dc_eig(T, Q, Λ) ⋯⋯ 入力 T から $T = Q\Lambda Q^T$ を満たす直交行列 Q, 実対角行列 Λ を出力する

if T が 1 次
　　return $Q = 1, \Lambda = T$
else
　　$T = \begin{pmatrix} T_1 & 0 \\ 0 & T_2 \end{pmatrix} + c\boldsymbol{v}\boldsymbol{v}^T$ の形式にする
　　call dc_eig(T_1, Q_1, Λ_1)
　　call dc_eig(T_2, Q_2, Λ_2)
　　$\Lambda_1, \Lambda_2, Q_1, Q_2$ から $D + c\boldsymbol{u}\boldsymbol{u}^T$ の形式にする
　　固有値問題 $D + c\boldsymbol{u}\boldsymbol{u}^T = Q'\Lambda Q'^T$ を解く
　　$Q = \begin{pmatrix} Q_1 & 0 \\ 0 & Q_2 \end{pmatrix} Q'$ を計算する
　　return Q, Λ
endif

なお，実装の際には，分割後の T の次元 n が 1 の場合のみではなく，n がある定数以下の場合に，他のより単純なアルゴリズムを利用することで，再帰の負荷を回避し，高速化を図る。

3.5 デフレーション

前小節で示した分割統治法を，多くの行列に対して，他の固有値問題解法と同等以上の速度に高速化する手法がデフレーションである。デフレーションは，対角行列と階数 1 の摂動の和の固有値問題に存在する，自明な固有値と固有ベクトルを取り除く手法である。

$n = 3$ の場合に，デフレーションが可能になる例を挙げると，

$$\left[\begin{pmatrix} d_1 & & \\ & d_2 & \\ & & d_3 \end{pmatrix} + c\begin{pmatrix} u_1 \\ 0 \\ u_3 \end{pmatrix}\begin{pmatrix} u_1 & 0 & u_3 \end{pmatrix}\right]\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = d_2\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \tag{18}$$

である。すなわち，摂動項の $\boldsymbol{u}$ の第 j 成分が 0 であるときに，自明な固有値は実対角行列の第 j 対角成分であり，対応する固有ベクトルは第 j 単位ベクトルである。ただし，第 j 単位ベクトルは，単位行列の第 j 列とする。

ここで分割統治法に話を限定し，6 次元の固有値問題に対する分割統治法の内部における問題を考える。いま，この行列の第 1 固有値が自明であると判定され，Q の第 1 列が第 1 単位ベクトルとなったとする。Q が直交行列であることより，Q の第 1 行の 2 列目以降が 0 となること

も自動的に決定する。したがって，P と Q の積は

$$PQ = \left(\begin{array}{ccc|ccc} * & * & * & & & \\ * & * & * & & & \\ * & * & * & & & \\ \hline & & & * & * & * \\ & & & * & * & * \\ & & & * & * & * \end{array}\right) \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & * & * & * \\ 0 & * & * & * & * & * \\ 0 & * & * & * & * & * \\ 0 & * & * & * & * & * \\ 0 & * & * & * & * & * \end{pmatrix} \tag{19}$$

となる。このとき，この積の第 1 列は計算する必要はないなどの演算量の削減が可能で，この例では，演算回数が 216 回から 150 回に減り，30 パーセント演算量が減少する。

分割統治法の一部として，対角行列と階数 1 の摂動の和の固有値問題を解いている場合，このデフレーションは，分割統治法に与えられた行列の性質によって発生率が変化する。デフレーションが多く発生する問題に対しては，$O(n^2)$ 以下の演算量で済む場合も存在し，平均して $O(n^{2.3})$ の演算量であるという報告もある。

4.　多分割へ拡張した分割統治法

2 分割の分割統治法を多分割に拡張することにより，行列積の演算量を削減することが可能で，アルゴリズム全体の演算量を削減することができる。

分割数が 3 の場合を例に，多分割の統治法のアルゴリズムを説明する。4 以上の分割数であっても，同様に拡張が可能である。

4.1　3 分割の固有値問題

この節では，2 分割の場合の説明と同様に，$n = 6$ を例に説明する。まず，実対称三重対角行列 T を

$$T = \left(\begin{array}{cc|cc|cc} a_1 & b_1 & & & & \\ b_1 & a_2 & b_2 & & & \\ \hline & b_2 & a_3 & b_3 & & \\ & & b_3 & a_4 & b_4 & \\ \hline & & & b_4 & a_5 & b_5 \\ & & & & b_5 & a_6 \end{array}\right)$$

$$= \left(\begin{array}{cc|cc|cc} a_1 & b_1 & & & & \\ b_1 & a_2 - b_2 & & & & \\ \hline & & a_3 - b_2 & b_3 & & \\ & & b_3 & a_4 - b_4 & & \\ \hline & & & & a_5 - b_4 & b_5 \\ & & & & b_5 & a_6 \end{array}\right) + b_2 \left(\begin{array}{c} 0 \\ 1 \\ \hline 1 \\ 0 \\ \hline 0 \\ 0 \end{array}\right) \left(\begin{array}{c} 0 \\ 1 \\ \hline 1 \\ 0 \\ \hline 0 \\ 0 \end{array}\right)^T + b_4 \left(\begin{array}{c} 0 \\ 0 \\ \hline 0 \\ 1 \\ \hline 1 \\ 0 \end{array}\right) \left(\begin{array}{c} 0 \\ 0 \\ \hline 0 \\ 1 \\ \hline 1 \\ 0 \end{array}\right)^T$$

と分割する。

このとき，2 次実対称三重対角行列 T_1, T_2, T_3 を用いて，

$$T = \left(\begin{array}{c|c|c} T_1 & & \\ \hline & T_2 & \\ \hline & & T_3 \end{array}\right) + b_2 \boldsymbol{v}_1 \boldsymbol{v}_1^T + b_4 \boldsymbol{v}_2 \boldsymbol{v}_2^T, \quad \boldsymbol{v}_1 = \left(\begin{array}{cc|cc|cc} 0 & 1 & 1 & 0 & 0 & 0 \end{array}\right)^T, \boldsymbol{v}_2 = \left(\begin{array}{cc|cc|cc} 0 & 0 & 0 & 1 & 1 & 0 \end{array}\right)^T$$

と書ける。

次に，分解した T_1, T_2, T_3 の固有値問題を解く。すなわち，T_1, T_2, T_3 の固有値問題 $T_i = P_i D_i P_i^T$ の解，実対角行列 D_1, D_2, D_3，直交行列 P_1, P_2, P_3 を求める。

最後に，T_1, T_2, T_3 の固有値問題の解 $D_1, D_2, D_3, P_1, P_2, P_3$ を利用して，T は，

$$\begin{aligned}
T &= \begin{pmatrix} P_1D_1P_1^T & & \\ & P_2D_2P_2^T & \\ & & P_3D_3P_3^T \end{pmatrix} + b_2\boldsymbol{v}_1\boldsymbol{v}_1^T + b_4\boldsymbol{v}_2\boldsymbol{v}_2^T \\
&= \begin{pmatrix} P_1 & & \\ & P_2 & \\ & & P_3 \end{pmatrix} \left\{ \begin{pmatrix} D_1 & & \\ & D_2 & \\ & & D_3 \end{pmatrix} + b_2\boldsymbol{u}_1\boldsymbol{u}_1^T + b_4\boldsymbol{u}_2\boldsymbol{u}_2^T \right\} \begin{pmatrix} P_1^T & & \\ & P_2^T & \\ & & P_3^T \end{pmatrix} \\
&\equiv P(D + b_2\boldsymbol{u}_1\boldsymbol{u}_1^T + b_4\boldsymbol{u}_2\boldsymbol{u}_2^T)P^T
\end{aligned} \tag{20}$$

と相似変形できる。ただし，実対角行列 $D \equiv D_1 \oplus D_2 \oplus D_3$，直交行列 $P \equiv P_1 \oplus P_2 \oplus P_3$，$\boldsymbol{u}_j \equiv P^T\boldsymbol{v}_j$ である。

このとき，実対角行列と階数 2 の摂動の和の固有値問題 $D + b_2\boldsymbol{u}_1\boldsymbol{u}_1^T + b_4\boldsymbol{u}_2\boldsymbol{u}_2^T = Q\Lambda Q^T$ を解くことで

$$T = PQ\Lambda Q^T P^T \tag{21}$$

となる。

ここで，P と Q の積は，

$$PQ = \left(\begin{array}{cc|cc|cc} * & * & & & & \\ * & * & & & & \\ \hline & & * & * & & \\ & & * & * & & \\ \hline & & & & * & * \\ & & & & * & * \end{array}\right) \begin{pmatrix} * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \end{pmatrix} \tag{22}$$

の形式をしている（$*$ はある実数，空白は 0 を意味する）。この行列積は，通常の 6 次行列同士の積の 1/3 の演算量で済む。これは，n 次行列でも同様であり，演算量は $(2/3)n^3$ である。k 分割の場合には，行列積の演算量は $(2/k)n^3$ である。すなわち，2 分割の分割統治法の $2/k$ の演算量に減少する。

また，実対角行列と低階数の摂動の和の固有値問題は $O(n^2)$ で解くことが可能である。そのため，k 分割の分割統治法は約 $(2/k)n^3$ の演算量であり，2 分割の分割統治法の $3/(2k)$ の演算量に減少する。ただし，実対角行列と低階数の摂動の和の固有値問題は，摂動の階数が増加すると解くために必要な演算量が増加し，分割統治法の中でこの部分が演算量の多くを占めるようになるため，問題に依存して演算量が最小となる分割数が存在する。

4.2 実対角行列と低階数の摂動の和の固有値問題

2 分割の場合には「実対角行列と階数 1 の摂動の和」の固有値問題を解く部分が，分割数が k に拡張された場合には,「実対角行列と階数 $k-1$ の摂動の和」の固有値問題を解くこととなる。

実対角行列と階数 $k-1$ の摂動の和の固有値は，実対角行列と階数 1 の摂動の和の固有値問題を $k-1$ 回逐次的に解くことで，求めることができる。実対角行列と階数 1 の摂動の和の固有値問題は，3.3 小節で述べたように，固有値を並列に求めることができるため，実対角行列と階数 $k-1$ の摂動の和の固有値の計算も，並列性を有する。

一方，固有ベクトルは，同じく 3.3 小節で述べた，実対角行列と階数 1 の摂動の和の固有ベクトル計算を拡張した方法により，求めることができる。その内部では，反復法による計算を行っている。この固有ベクトル計算は，それぞれの固有ベクトルについて独立であり，ここに高い並列性を有する。

両者の方法ともに，n 次元の問題を解くための演算量は $O(n^2)$ である。ただし，摂動の階数が増加したことで，問題の難度が上がり，解くために必要な演算量自体は増加する。

4.3 デフレーション

実対角行列と階数 1 の摂動の和の固有値問題と同様に，摂動の階数が 2 以上の場合でも，デフレーションが発生する。それにより，多分割の分割統治法でも，演算量の支配的な部分を占める行列積の演算量を削減することができる。

しかし，分割数の増加に従って，デフレーション発生率が減少するため，デフレーションによる演算量の削減効果は薄れる。

4.4 アルゴリズム

この節の最後に，3 分割の分割統治法を擬似コードとしてまとめる。

アルゴリズム　3 分割の分割統治法を用いて，n 次実三重対角行列 T の固有値・固有ベクトルを求める:

proc dc_eig3(T, Q', Λ) $\cdots\cdots$ 入力 T から $T = Q'\Lambda Q'^T$ を満たす直交行列 Q', 対角行列 Λ を出力する

if T が 1 次か 2 次
　　call eig(T, Q', Λ)　　(* eig() は固有値問題 $T = Q'\Lambda Q'^T$ を解く他の手続き *)
　　return Q', Λ
else
　　$T = \begin{pmatrix} T_1 & & \\ & T_2 & \\ & & T_3 \end{pmatrix} + c_1 \boldsymbol{v}_1 \boldsymbol{v}_1^T + c_2 \boldsymbol{v}_2 \boldsymbol{v}_2^T$　(* T_i は実三重対角行列，$\boldsymbol{v}_1$, $\boldsymbol{v}_2$ は実ベクトル，c_1, c_2 は実数 *)
　　call dc_eig3(T_i, P_i, D_i)　$(i = 1, 2, 3)$
　　D_i, P_i から $D + c_1 \boldsymbol{u}_1 \boldsymbol{u}_1^T + c_2 \boldsymbol{u}_2 \boldsymbol{u}_2^T$ の形式にする $(i = 1, 2, 3)$
　　固有値問題 $D + c_1 \boldsymbol{u}_1 \boldsymbol{u}_1^T + c_2 \boldsymbol{u}_2 \boldsymbol{u}_2^T = Q\Lambda Q^T$ を解く
　　$Q' = \begin{pmatrix} P_1 & & \\ & P_2 & \\ & & P_3 \end{pmatrix} Q$ を計算する
　　return Q', Λ
endif

なお，実装の際には，2 分割の場合と同様に，n がある定数以下の場合に，他のより単純なアルゴリズムに切り替えることで，高速化を図る。

5. 共有メモリ型並列計算機への実装

5.1 並列計算

並列計算には大きく分けて，共有メモリ型並列計算と分散メモリ型並列計算の 2 通りがある。この 2 つを SR11000 に当てはめた場合，128 個存在するノードのいくつかを使って並列化する方法が分散メモリ型並列計算であり，1 ノードのみを使い全 16 個のプロセッサの内の複数のプロセッサを用いて並列化する方法が共有メモリ型並列計算である。ノードとは，スーパーコンピュータの構成要素の一つで，単一のメモリ空間と複数のプロセッサからなるものである。

本研究では多分割の分割統治法の並列化の第一段階として，1 ノードを利用した共有メモリ型の並列化を施し高速化を図る。今後の研究では複数ノードを利用した分散メモリ型の並列化を目指している。

次の小節ではいくつかの用語説明を交えながら共有メモリ型並列プログラムの作成方法について説明する。

5.2 共有メモリ型の並列化

共有メモリ型の並列化を行う方法には，共有メモリ型並列化を施されたライブラリをリンクして用いる方法と，指示子と呼ばれる文をソースコード内に挿入して，コンパイラに並列化の指示を与える方法がある。SR11000 上では，LAPACK 等の，既に並列化を施されたライブラリが提供されているため，前者の並列化の方法はこれらをリンクすることで実現できる。一方，指示子を挿入して並列化を行う場合，一般的に OpenMP という API を用いる。OpenMP は，Unix や Windows NT のプラットフォームを含む多くのアーキテクチャでサポートされている API である。本研究においても OpenMP を用いて並列化を行った。

C 言語で書かれたプログラムを OpenMP によって並列化を行う場合，コード中に図 7 の命令を書くことで，波括弧内の命令が複数のプロセッサを用いて実行を行う並列区間であることを指示することができる。並列区間では，各プロセッサの処理は「スレッド」と呼ばれる並列実行単位で表される。

```
#pragma omp parallel
{
  /* 実行文 */
}
```

図 7　並列部の指定

図 8 のように並列区間中の for 文の前に一文 “#pragma omp for” を入れることで，この for 文は複数のスレッドで分担して実行を行うことをコンパイラに指示する。“#pragma omp for” を for 指示文と呼ぶ。本研究では，自作したコードはすべて for 指示文を用いて並列化した。

```
#pragma omp parallel
{
  int i ;
  /* 実行文 */
  #pragma omp for
  for ( i = 0 ; i < 100 ; i++ ) {
    /* 実行文 */
  }
  /* 実行文 */
}
```

図 8　ループ並列 (for 文の並列化)

for 指示文では，スレッドに仕事を割り振る方法を明示的に指示することもできる。チャンクサイズ 10 で静的に割り付けるには “#pragma omp for” の後に “schedule (static , 10)” を付け足すことで指示できる。以下でチャンクサイズ，静的および動的な割り付けについて説明する。

チャンクサイズとは 1 回の処理量のことで，図 8 を例に説明すればチャンクサイズ 10 は変数 i のカウント 10 個分を 1 回として考えることを意味し，5 スレッドを静的に割り付けて並列実行した場合，各スレッドで 2 回分ずつ演算を行う。もしチャンクサイズを指定しなければデフォルトのサイズ (各スレッドで 1 回分ずつ演算を行うサイズ) で割り付けを行う。

スレッドに仕事を割り振ることを割り付けといい，静的な方法と動的な方法がある。静的に割り付ける場合はチャンクサイズ分の演算を各スレッドへ順番に割り振る。動的に割り付ける場合は順次，演算の終わったスレッドにまだ終わっていない演算を割り振る。

動的に割り付けを指示する際は “static” の代わりに “dynamic” と宣言する。静的に割り付けを行った際は，割り振り方が決定的で時間がかからない。しかし，1 回の演算の計算時間が各々異なる場合，計算時間が偏り並列効果が下がることがある。その場合には，動的な割り付けを選択することにより，並列効果の向上が期待できる。しかし，動的な割り付けは静的な割り付けより割り付け時間を要する。

詳しい OpenMP の利用法についてはマニュアル「最適化 C 使用の手引」を参照されたい。また以後の小節では，本研究で施した並列化および並列化したアルゴリズムの項目の主要部分について説明を行う。

5.3 並列化項目

4 節で示した固有値問題に対する多分割の分割統治法のアルゴリズム中で，主に，以下に述べる 4 つの項目に対して並列化を行った。デフォルトのチャンクサイズで静的な割り付けを行った場合に並列効果が低い場合は，動的な割り付けを試し，並列効果の高い方法を選択した。

5.3.1 T_i の固有値問題計算

T_i の固有値問題計算とは，実対称三重対角行列 T を分割して得られた行列 T_i の固有値問題計算のことである (4.1 小節参照)。数値実験では，並列版 LAPACK ライブラリにある，2 分割の分割統治法を用いた実対称三重対角行列の固有値問題解法関数を用いて並列計算を行った。

明示的に再帰呼び出しをせず，ライブラリ関数を利用した理由は，1. 分割するにつれ，最適分割数が小さくなる傾向があること，2. 現在，与えられた問題に対し動的に最適分割数を選ぶ手法が明らかにされていないこと，3. 従来の分割統治法との差異を少なくし比較しやすくすることの 3 点である。最適分割数については以降で説明する。

5.3.2 対角行列と低階数の摂動の和の固有値計算

対角行列と低階数の摂動の和の固有値計算の主要部分は「対角行列と階数 1 の摂動の和の固有値計算」である (4.2 小節参照)。対角行列と階数 1 の摂動の和の固有値計算では，各固有値計算が独立した計算であるため，for 指示文を用いて並列化を行う。反復解法を用いているが，割り付けの方法による計算時間の差異が見られなかったので，本プログラムではデフォルトのチャンクサイズを用い，静的な割り付けを行った。

5.3.3 対角行列と低階数の摂動の和の固有ベクトル計算

各固有ベクトルは，内部的に反復解法を用いて求める (4.2 小節参照)。各固有ベクトルの計算は独立した計算であるため，for 指示文により効率的な並列化を行うことができる。本研究では各々の反復法による計算時間が一定でないため，チャンクサイズを 1 とし動的に割り付けを行った。

5.3.4 行列積計算

「行列積計算」は実対称三重対角行列の固有ベクトルを求めるための行列積計算で，式 (22) の Q と P の行列積を指す。この計算には並列版 BLAS ライブラリを用いた。このライブラリは SR11000 用にチューニングが施されていて，2.3.2 小々節で述べたように，自作するよりも高速に計算できる。

6. 数値実験

6.1 実験環境

本研究では SR11000 の 1 ノードを用いて実験を行った。並列計算に使用した CPU の台数は 16 台である。実行時間の計測には FORTRAN の関数 `xclock` を用いて，実時間の計測を行った。以降では，使用したライブラリおよびそのライブラリを利用したプログラムのコンパイル方法について説明する。

6.1.1 ライブラリ

数値計算ライブラリとして SR11000 で提供されている LAPACK(BLAS) ライブラリを用いた。比較対象として用いた関数は表 2 の通りである。

表 2 LAPACK 関数名

	関数名
QR 法	dsteqr
二分法・逆反復法	dstebz, dstein
MRRR 法	dstegr
分割統治法	dstevd

表 3 コンパイラ

言語	コンパイラ名	コマンド名
C 言語	最適化 C	cc
FORTRAN90	最適化 FORTRAN90	f90

6.1.2 コンパイル方法

コンパイラとして最適化 C コンパイラと最適化 FORTRAN90 コンパイラを使用した (表 3)。ソースコードは C 言語で作成したため，コンパイルには最適化 C コンパイラを用いた。オブジェクトのリンクには，FORTRAN 用に作成された LAPACK ライブラリをリンクするので，最適化 FORTRAN90 コンパイラを用いた。

最適化オプションは，-Os にすると本プログラムでは正常に計算できなかったので，正常に計算できる最大レベルである-O4 を選択した。並列化レベルは，最高レベルの 4 とした。並列プログラムでは OpenMP を使用しているので-omp を指定する。逐次プログラムでは逐次ライブラリをリンクするため-L/usr/local/lib -llapack_sc -lblas_sc を指定する。並列プログラムでは並列ライブラリをリンクするため-L/usr/local/lib -llapack -lblas を指定する。LAPACK ライブラリ

は FORTRAN 用に作成されているため，C 言語から呼び出せるように-lf90s を指定する。実際に指定したオプションは表 4 の通りである。

表 4　コンパイル命令

逐次	コンパイル	cc -64 -O4 +Op -noparallel -c
	リンク	f90 -64 -O4 -i,P -noparallel -L/usr/local/lib -llapack_sc -lblas_sc -L/opt/ofort90/lib/ -lf90s -lm
並列	コンパイル	cc -64 -O4 +Op -parallel=4 -pardiag=2 -omp -c
	リンク	f90 -64 -O4 -i,P -parallel=4 -pardiag=2 -omp -L /usr/local/lib -llapack -lblas -L/opt/ofort90/lib/ -lf90s -lm

6.2　実験方法

本実験では，大きく性質の異なる 2 種類の行列を入力とし，いくつかの分割数を用いて数値実験を行った。この時，計算に必要とした実行時間を測定し，計算結果の精度を評価するため残差と直交誤差を求めた。以降では，入力として用いた行列，最適分割数および精度の評価法について説明する。

6.2.1　対象行列

計算対象の行列として，主対角要素に $(2,4]$，副対角要素に $(1,2]$ の区間の一様乱数を持つ行列と，主対角の第 j 要素に $j\times 10^{-6}$，副対角要素に 1 を持つ行列の 2 種類を用いた。前者はデフレーションの発生が多い行列で行列 A と呼ぶ。後者はデフレーションの発生が非常に少ない行列で行列 B と呼ぶ。

6.2.2　時間計測

時間計測には前述の通り，`xclock` を用いて実時間を計測した。行列 A に対しては 50 回の計測を行って平均を取った（行列要素は計測ごとに異なる）。現状のプログラムでは正常に計算することができないことがあり，その場合は計算できた中で平均を取った。

6.2.3　精度の評価法

計算結果の精度については，相対残差 ϵ_R と直交誤差 ϵ_O，

$$\epsilon_R = \max_{1\le i\le n}\frac{||T\boldsymbol{q}_i-\lambda_i\boldsymbol{q}_i||_2}{||T||_2},\quad \epsilon_O=\max_{1\le i\le j\le n}|\boldsymbol{q}_i^T\boldsymbol{q}_j-\delta_{ij}| \tag{23}$$

を用いて評価する。ただし，T は入力した実対称三重対角行列，λ_i は T の固有値，$\boldsymbol{q}_i$ は λ_i に対応する T の固有ベクトル，δ_{ij} はクロネッカのデルタとする。

6.2.4　最適分割数

対象行列の性質やサイズによって，提案手法を用いて最も高速に計算できる行列の分割数は変化する。今回の数値実験では，一つの対象行列に対していくつかの分割数で実行時間を計測し，その中で最も速く計算できた時の分割数を，その対象行列に対する最適分割数と呼ぶこととする。

表 5　各アルゴリズムの実行時間：$n = 10000$

アルゴリズム		行列 A			行列 B		
		逐次 (秒)	並列 (秒)	台数効果	逐次 (秒)	並列 (秒)	台数効果
LAPACK	QR 法	1671.504	450.395	3.711	1489.022	424.209	3.510
	二分法	1074.868	483.799	2.222	2910.293	1221.946	2.382
	MRRR 法	24.191	23.689	1.021	19.137	18.614	1.028
	分割統治法	6.914	3.297	2.097	209.829	23.258	9.022
提案手法		8.137	3.256	2.499	120.540	14.060	8.573

表 6　各アルゴリズムの実行時間：$n = 5000$

アルゴリズム		行列 A			行列 B		
		逐次 (秒)	並列 (秒)	台数効果	逐次 (秒)	並列 (秒)	台数効果
LAPACK	QR	228.364	90.585	2.521	199.937	80.023	2.498
	二分法	97.980	77.412	1.266	383.135	259.976	1.474
	MRRR 法	6.226	6.144	1.013	4.713	4.593	1.026
	分割統治法	2.480	1.525	1.626	28.334	4.687	6.045
提案手法		4.726	1.947	2.427	12.240	2.509	4.878

6.3　実験結果

6.3.1　各アルゴリズムの比較

表 5, 6 はアルゴリズム別の実行時間についてまとめた表で，縦軸はアルゴリズム，横軸は各対象行列の計算に要した時間と台数効果である。行列 A, B のサイズ 10000 (表 5) と 5000 (表 6) について各々逐次プログラムと並列プログラムの計測を行った。提案手法の分割数は，並列計算時の最適分割数を選択した。

表 5, 6 より，行列 A の場合は，LAPACK の MRRR 法と比較すると逐次プログラムの段階で計算速度が速く，並列化することによりさらに高速化できている。LAPACK の分割統治法と比較すると $n = 10000$ の時は同等の速度だが $n = 5000$ の時は遅い。行列 B の場合は LAPACK の MRRR 法と比較すると，逐次プログラムでは遅いものの，並列プログラムでは台数効果が高く高速に計算できている。LAPACK の分割統治法と比較しても高速に計算できている。

表 5, 6 より台数効果の比較を行うと，多分割の分割統治法の台数効果は LAPACK の分割統治法と同等である。また，MRRR 法と比較すると，多分割の分割統治法の台数効果が高い。

6.3.2　最適分割数

表 7, 8 は並列プログラムの計算時間を分割数別にまとめた表で，縦軸は対象行列のサイズ，横軸は各分割数である。行列 A (表 7), B (表 8) のサイズ 10000 と 5000 について計測を行った。

表 7 より行列 A の最適分割数は $n = 10000$ では 3 分割であった。$n = 5000$ では 16 分割であったが 2 番目は 3 分割であった。このことから，行列 A の最適分割数は小さい傾向があるように見て取れる。表 8 より行列 B の最適分割数は $n = 10000$ でも $n = 5000$ でも 16 分割であった。このことから，行列 B の最適分割数は大きい傾向があるように見て取れる。よって，最適分割数は対象行列の性質に依存して大きく変化することが窺える。

表 7　分割数による実行時間の変化：行列 A

分割数	3	4	5	6	10	16	20
$n = 10000$ (秒)	3.256	3.346	3.504	3.583	3.917	3.897	3.955
$n = 5000$ (秒)	1.953	2.003	2.002	2.019	1.968	1.947	1.955

表 8　分割数による実行時間の変化：行列 B

分割数	3	4	5	6	10	16	20
$n = 10000$ (秒)	19.451	17.245	16.327	15.458	14.944	14.060	15.357
$n = 5000$ (秒)	3.405	2.995	3.002	2.716	2.682	2.509	2.848

表 9　並列プログラムの計算結果の精度：行列 A, 問題サイズ $n = 10000$

アルゴリズム		相対残差 ϵ_R	直交誤差 ϵ_O
LAPACK	MRRR 法	1.99×10^{-14}	5.36×10^{-12}
	分割統治法	6.84×10^{-15}	5.77×10^{-15}
研究手法	3 分割	5.49×10^{-15}	8.49×10^{-15}
	16 分割	5.84×10^{-15}	1.07×10^{-14}

6.3.3　計算結果の精度

表 9 は並列プログラムの計算結果の精度についてまとめた表で縦軸は各アルゴリズム，横軸は各項目である。各アルゴリズムを用いて，行列 A のサイズ 10000 について計測を行った。

表 9 より計算結果の精度について比較を行うと，残差について比較した場合，どのアルゴリズムも同等の精度で計算が行われている。直交誤差について LAPACK の分割統治法と比較した場合，同等の精度で計算が行えている。LAPACK の MRRR 法と比較した場合，研究手法の方がより小さな誤差で精度の高い計算が行えている。

6.3.4　多分割の分割統治法の実行時間の内訳

表 10 は，研究手法の並列プログラムにおける，各並列化項目別の計測結果の表である。縦軸は各分割数および各対象行列，横軸は各並列化項目である。計算時間の長い行列 B のサイズ 10000 について計測を行った。各並列化項目は以下の通りである。

小問題　「T_i の固有値問題計算」

固有値　「対角行列と低階数の摂動の和の固有値計算」

固有ベクトル　「対角行列と低階数の摂動の和の固有ベクトル計算」

行列積　「行列積計算」

表 10 より各並列化項目別に実行時間を計測した場合，行列積部分の台数効果が最も大きい。これは，ベンダによってチューニングを施されたライブラリを利用していることの恩恵である。固有値，固有ベクトル計算に関しては OpenMP による並列化を行ったが，16 台の台数効果は得られないまでも並列化効果が出ている。よって，行列積や「対角行列と低階数の摂動の和の固有値，固有ベクトル計算」は並列性が高く，多分割の分割統治法は並列化に向いているといえる。

表 10　多分割の分割統治法の実行時間の内訳：対象行列 B , $n = 10000$

分割数	プログラム	全体	小問題	固有値	固有ベクトル	行列積	その他
4 分割	逐次 (秒)	172.427	15.846	7.19	73.021	75.873	0.497
	並列 (秒)	17.245	4.936	0.756	6.908	4.592	0.053
	台数効果	9.999	3.210	9.511	10.570	16.523	9.377
16 分割	逐次 (秒)	120.54	1.679	9.766	86.924	21.693	0.478
	並列 (秒)	14.06	2.297	1.173	9.203	1.335	0.052
	台数効果	8.573	0.731	8.326	9.445	16.249	9.192

7.　まとめ

実対称固有値問題に対する多分割の分割統治法の並列化の第一段階として，OpenMP を用いた共有メモリ型の並列化を施した。次の 4 項目，分割した実対称三重対角行列の固有値問題計算，対角行列と低階数の摂動の和の固有値計算，対角行列と低階数の摂動の和の固有ベクトル計算，および実対称三重対角行列の固有ベクトルを求めるための行列積計算が全体の計算の大部分を占めているため，我々はこの 4 項目を中心に並列化を施した。

研究手法および LAPACK の MRRR 法や分割統治法の数値実験を行った結果，サイズ 10000 の行列では，研究手法が LAPACK の MRRR 法や分割統治法よりも高速に計算でき，研究手法は従来の分割統治法と並んで， MRRR 法よりも大きな台数効果が得られるという結果を得た。最適分割数は，並列計算においても逐次計算同様，行列の性質に依存して変化する結果が得られた。誤差評価の面では，残差は他のアルゴリズムと同等の精度を保持することができ，直交誤差は従来の分割統治法と並んで MRRR 法よりも高精度を実現できた。並列化を施したことにより，対角行列と低階数の摂動の和の固有値計算とその固有ベクトル計算，および，実対称三重対角行列の固有ベクトルを求めるための行列積計算の 3 項目の速度が向上された。以上のことから，サイズが 10000 程度の行列の固有値問題解法として，多分割の分割統治法は 2 分割の分割統治法や MRRR 法よりも高速であることが示された。

8.　今後の課題

8.1　デフレーション発生率の研究

多分割の分割統治法では逐次実行，並列実行ともに分割数によって実行時間は大きく異なる。このため，自動的に最適分割数を決定する方法が必要とされる。最適分割数はいくつかの要因によって決まっていると思われるが，その要因の一つにデフレーションの発生率があげられる。デフレーションの発生率が非常に低い場合，行列積の計算に時間を要すために最適分割数は大きくなり，逆に，デフレーションの発生率が高い場合,「対角行列と低階数の摂動の和の固有値，固有ベクトル計算」にかかる時間が重要となり，最適分割数は小さくなる傾向がある。このため，行列の分割を行う前に少ない計算量でデフレーションの発生率について知ることができたならば，最適分割数を検討することができる。このため，デフレーションの発生率について研究する必要がある。

8.2　プログラムの安定性

現在のプログラムでは，逐次版および並列版のプログラムで一部の問題に対して，正常に計算できないことがある。これは 0 に近い値の除算が数多く存在し，精度が確保できないことによる。また，逐次版プログラムで正常に計算できる問題でも並列版プログラムで正常に計算できないことがある。このため，アルゴリズムを改善し数値計算に適した方法を考案する必要や，安定に計算を進めるプログラムの作成が必要である。

8.3　並列化に伴う速度低下への対応

サイズの十分小さい問題に対して並列版 LAPACK ライブラリを用いると，逐次版よりも計算時間が掛かることがあった。この現象はライブラリ内部で並列化された処理が高速に計算できず，加えて，並列化に必要な処理時間が加算されたことによると思われる。表 10 の小問題の項目で台数効果が得られていないのはこのためである。本来この計算 (T_i の固有値問題計算) は並列性があり，並列効果が期待できる部分である。このため，この部分に関して改善できたならば今よりも高速に計算できるはずである。

8.4　分散メモリ型並列計算プログラムの開発

現在のプログラムは共有メモリ型並列計算プログラムであり，分散メモリ型並列計算プログラムへの実装は行っていない。SR11000 ではノードを複数利用した分散メモリ型並列計算も行うことができ，多分割の分割統治法を分散メモリ型に実装することで更なる高速化，および，対応できる問題の大規模化が期待できる。マシン性能を引き出すには分散メモリ型並列計算プログラムを開発する必要がある。

9.　付録

本節ではプログラムの実装を行う中で，高速化に成功したことやうまく高速化できなかったことについて記載する。

9.1　行列積の高速化

本研究では行列積の計算に BLAS ライブラリを用いた。BLAS を使う場合，必ずしも行列のデータの各列ベクトルデータがメモリ上で連続である必要はない。しかし，連続であるほうがキャッシュのアクセス性能があがる可能性があり，実際に「本問題の固有ベクトル計算」でメモリ上のデータを連続に並べた場合，並べない場合よりも行列積の計算が高速に行えた。

9.2　C 言語を用いた LAPACK の使用方法

C 言語から LAPACK を利用する際は注意が必要である。一つは関数名。ライブラリの作成に用いた言語と違う言語からそのライブラリを使用する時，環境によって関数名が異なる。SR11000 の環境で C 言語から LAPACK を呼ぶ場合は，関数名は全て小文字となる。二つ目はリンク方法。LAPACK が FORTRAN で書かれているため，コンパイルは C コンパイラで，リ

ンクは FORTRAN コンパイラで行わなくてはならない。リンク時には -lf90s オプションが必要である。三つ目はプロファイラが使えないこと。コンパイラにはプロファイル使用のためのオプションがついているが，コンパイル時に C 言語版のライブラリ呼び出しが書き込まれ，リンク時に FORTRAN 版をリンクしようとするためか，コンパイル出来なかった。

9.3 LAPACK を利用した並列プログラムの作成上のトラブル

本小節では LAPACK を利用した並列プログラムの作成に関し，問題のあった項目について説明し，本研究でとった対処法について述べる。

9.3.1 トラブル

プログラムの作成に当たり 3 つの問題点があった。以下，その問題点について説明する。

■並列版 LAPACK をサイズの小さい問題に使用した時の速度低下　前述の通り，サイズの十分小さい問題に対して並列版のライブラリを用いると，逐次版よりも計算時間が掛かる。このため，同一プログラム内で大規模な問題と小規模な問題を解く場合，並列版のリンクに注意する必要がある。

■LAPACK の逐次版, 並列版ライブラリの同時利用　前項で述べたように，サイズの小さい問題を解く際に並列版ライブラリを用いた場合，逐次版を用いた場合よりも遅いため，サイズの小さい問題を解く際は逐次版を用い，サイズの大きい問題を解く際は並列版を用いることが相応しい。しかし，SR11000 に備え付けの LAPACK ライブラリは逐次版でも並列版でも関数名は同一である。このため使い分けは出来ない。

■LAPACK のスレッドセーフ性　並列版 LAPACK を使用した場合には一部で速度低下が発生するため，OpenMP で並列化を行い，各スレッドで逐次版のライブラリを呼び出すことで高速化が期待できる。しかし，SR11000 上に備え付けの LAPACK ライブラリはスレッドセーフではない。このため，LAPACK 関数を並列に呼び出すコードを書くと正しい計算が行われない可能性がある。

9.3.2 対処法

以上 3 点の理由から本研究では並列版プログラムに並列版ライブラリのみを用いる。ただし，OpenMP を用いて並列化を行いたい部分に LAPACK 関数が含まれている場合，代用の関数を自作して置き換えた後並列化を行った。対象行列によっては全体の実行時間で逐次版よりも並列版の方が遅くなってしまうこともあるがこの問題は今後の課題とした。

9.4 LAPACK(BLAS) サービスへの要望

SR11000 に備え付けの LAPACK(BLAS) ライブラリはマシンに対して最低限のチューニングしか施されていないそうである。このため，十分にマシン性能を引き出せていなかったり並列化効果が上がらないことがある。また，SR11000 に備え付けの LAPACK(BLAS) ライブラリはスレッドセーフでないため，逐次ライブラリを並列に実行することができない。ライブラリのチューニングが強化されることを希望する。

参 考 文 献

桑島 豊, 重原 孝臣, 実対称三重対角固有値問題の分割統治法の拡張, 日本応用数理学会論文誌, 15 (2005), 89–115.

桑島 豊, 重原 孝臣, 実対称三重対角固有値問題に対する多分割の分割統治法の改良, 日本応用数理学会論文誌, 16 (2006), 453–480.

山本有作, 密行列固有値解法の最近の発展 (I) – Multiple Relatively Robust Representations アルゴリズム –, 日本応用数理学会論文誌, 15 (2005), 181–208.

William H. Press, 丹慶勝市 訳,『NUMERICAL RECIPES in C [日本語版]』,685 p, 技術評論社, 1999

OpenMP, http://www.openmp.org/

LAPACK (Linear Algebra PACKage),http://www.netlib.org/lapack/index.html

BLAS (Basic Linear Algebra Subprograms),http://www.netlib.org/blas/index.html

ATLAS(Automatically Tuned Linear Algebra Software),http://www.netlib.org/atlas/index.html

スーパーコンピュータ SR11000 でのプログラム開発事例：カップルドクラスター法による高分子の電子状態計算

片 桐 秀 樹

産業技術総合研究所　計算科学研究部門

1. はじめに

本稿では，**カップルドクラスター(coupled-cluster)法**という**電子状態計算**の手法を，高分子に適用するために行なったスーパーコンピュータ SR11000 でのプログラミングについて，その一部を紹介したいと思います。その前に，電子状態計算とは何なのか，そこから話を始めることにしましょう[1-3]。

我々の世界に存在する物質は，原子の集合体でできています。さらに，原子は核と電子から成り立っています。電子と核の間にはクーロン力による引力が働き，また，電子と電子の間は斥力が働いています。通常，電子は核に束縛された安定な状態にありますが，光と相互作用したり，原子や電子が衝突することによって，その状態は変化します。電子が作り出す様々な状態を**電子状態**と呼んでおり，我々は物質の電子状態を知ることによって，その物質の電気的・光学的性質や化学反応などの化学的性質を理解または予測することができます。

さて，電子状態を理論的に調べることとは，電子の**波動関数**を求めること，すなわち電子の**シュレディンガー方程式**を解くことに他なりません。量子力学が誕生して以来，電子状態計算の方法論は大きく発展しました。特に，コンピュータの進歩によって電子状態計算は格段の進歩を遂げ，物質科学に多大な貢献をするに至っています。しかし，例えば水分子のような電子を 10 個しか含まないような単純な系であっても，その波動関数を正確に求めることは，現在でも困難な問題です。これは，電子間に働くクーロン相互作用が長距離力であり，この相互作用のもとに多体問題を解くことが容易ではないことが関係しています。そのため，近似的に波動関数を求める様々な手法が提案されています。

現在，電子状態計算に広く使われている方法はいくつかあります。（但しここでは，経験的なパラメータを用いたモデル理論による方法は除き，そのようなパラメータを必要としない，いわゆる**第一原理計算**とか **ab initio 計算**と呼ばれるものに限定することにします。）

一つは，**ハートリーフォック(Hartree-Fock)法**と呼ぶ方法によって波動関数を求める方法，そしてそれに基づいてさらに精度の高い波動関数を求める方法です。ハートリーフォック法では，**平均場近似**（つじつまの合った場の方法）を用いてシュレディンガー方程式を解き，独立粒子の波動関数を求めます。これで用が足りる場合もありますが，より精度を求める場合や，基底状態だけではなく励起状態を求めたい場合は，さらに進んで**多体効果**を取り入れる必要があります。この多体効果のことを**電子相関**と呼んでいます。電子相関を取り入れる方法としては，配置間相互作用法，摂動法，また，本稿で取り上げるカップルドクラスター法，などがあります。これらの方法は，従来から主に原子・分子に適用されながら発展してきましたが，計算コストが高いことがネックとなり，固体への応用例は多くはありません。

もうひとつの有力な方法は，**密度汎関数法**と呼ばれる方法です。これは，**ホーヘンバーグ・コ**

ーン（Hohenberg-Kohn）の**定理**に基づき，系のエネルギーが電子密度の汎関数となっていることを利用して，電子状態を求める手法です。汎関数の形は未知なので厳密な解を求めることはできませんが，便宜的に**コーン・シャム（Kohn-Sham）方程式**を解く方法が使われます。密度汎関数理論では，電子の多体効果は，電子密度の関数で表される**交換相関ポテンシャル**によって取込むことができます。密度汎関数法は，アボガドロ数のオーダーとなる数の電子を含む凝縮系の計算に適しており，固体のバンド計算の分野で大きな成功をおさめました。分子計算の分野においても近年急速に普及しています。

密度汎関数法は，ハートリーフォック法に基づく方法に比べ，少ない計算量の割に精度の高い結果が得られるという利点があるのですが，少なからず問題点があることもわかってきています。それは，バンドギャップを過小評価することや，強相関電子系などいくつかの系に対して正しい計算結果が得られないという点です。これらの問題点を改善すべく，交換相関ポテンシャルを改良する方法や，GW 近似などの方法などが，現在精力的に研究されています。

これら以外の電子状態を研究する有力な方法としては，**量子モンテカルロ法**があります。この方法は，系のサイズを増やしたときに計算量が極端に増加しないこと，並列コンピュータに実装しやすいことなどのメリットがあり，今後有望な方法の一つです。

さて，筆者は，ハートリーフォック法およびカップルドクラスター法を固体（一次元高分子）に適用する研究[4]を行なっており，そのためのプログラム開発を行なっています。カップルドクラスター法は，ハートリーフォック法から出発した後に電子相関を考慮する方法の一つで，精度が高く，分子計算の分野ではすでに標準的な手法として使われています。しかし，カップルドクラスター法は非常に大きな計算量を必要とするため，系のサイズが大きい時に解くことは現在でも容易ではありません。このことは特に，固体に応用する場合には大きな問題になります。カップルドクラスター法を固体などの大きな系に適用するためには，高速なコンピュータの利用は必須であると言えるでしょう。

次節以降ではカップルドクラスター法の一般論を述べてから，高分子に適用する場合の具体的な計算手順を，計算量の大きい部分に重点を置いて説明します（２，３節）。ここでは読者に，ある程度の量子力学の知識があることを想定しています。後半では，カップルドクラスター法の計算で大きなウエイトを占める大次元配列の積和計算に焦点を当て，スーパーコンピュータでのプログラミングのテクニックを解説します（４節）。

２．カップルドクラスター法とは何か

カップルドクラスター法は上で説明したように，**一粒子の波動関数（ハートリーフォック軌道）**を出発点として，多体効果を取込んだ波動関数を求める方法です。この方法は Coester らによって核物理学における多体問題に使われたのが始まりで[5]，その後，Cizek によって電子の計算に応用されました[6]。

カップルドクラスター法では，まず最初にハートリーフォック法で次の**固有値方程式**を解き，ハートリーフォック軌道を求めます。

$$\hat{F} = \hat{h} + \sum_{j=1}^{N} (\hat{J}_j - \hat{K}_j) \qquad (1)$$

$$\hat{F} |\phi_i\rangle = \varepsilon_i |\phi_i\rangle \qquad (2)$$

$\hat{F}$ は**フォック演算子**で，$\hat{h}, \hat{J}, \hat{K}$ はそれぞれ**一電子**（電子の運動エネルギー項と電子−核の引力項の和），**クーロン**，**交換**の各相互作用を表す演算子を表します。Nはここでは電子数を表します。ϕ_iが求めるべきハートリーフォック軌道で，ε_iは対応するエネルギー固有値です。ハートリーフォック法では，個々の電子が他の電子が作る平均的な場の中を運動しているという近似のもとに電子の運動を記述します。 ϕ_iを適当な**基底関数**の展開で表せば，(2)式は行列の固有値問題になり，容易に解くことができます。基底関数としては， $\hat{h}, \hat{J}, \hat{K}$ の行列要素が簡単に計算できるという理由で，ガウス型関数 $\exp(-\alpha r^2)$ の線形結合で表した関数が使われます。これは**ガウス型基底関数**と呼ばれ，分子の電子状態計算で広く用いられています。ガウス型基底関数は周期律表の各原子に対して決められたものが多数あり，容易に入手することができます[7]。

さて，カップルドクラスター法は，**多体波動関数**が以下の様な形で書けるという ansatz[†]に基づいています。

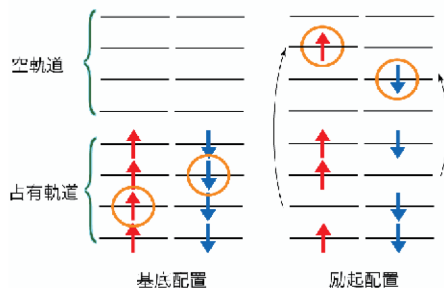
$$\left|\Psi_{CC}\right\rangle = \exp(\hat{T})\left|\Psi_0\right\rangle \qquad (3)$$

$\left|\Psi_0\right\rangle$は上で求めたハートリーフォック軌道関数$\{\phi_1, \phi_2, \phi_3, \cdots\}$から作られた，反対称化された**スレーター行列式**$\|\phi_1\phi_2\phi_3\cdots\|$です。$\hat{T}$ は**励起演算子**と呼ばれ，さらにプリミティブな励起演算子の和 $\hat{T} = \hat{T}_1 + \hat{T}_2 + \hat{T}_3 + \cdots$ で表されます。

励起演算子は，ハートリーフォック法で求めた**占有軌道**（電子が入った軌道）から**空軌道**（電子が空の軌道）に電子を励起させる働きを持ちます。**生成・消滅演算子**を用いて$\hat{T}_1, \hat{T}_2, \hat{T}_3, \cdots$を具体的に表せば，以下のようになります。

$$\hat{T}_1 = \sum_{ia} t_i^a a^\dagger i, \quad \hat{T}_2 = \frac{1}{(2!)^2} \sum_{ijab} t_{ij}^{ab} a^\dagger i b^\dagger j,$$
$$\hat{T}_3 = \frac{1}{(3!)^2} \sum_{ijkabc} t_{ijk}^{abc} a^\dagger i b^\dagger j c^\dagger k, \quad \cdots \qquad (4)$$

ここで，$i, j, k, \cdots$は被占軌道に対する消滅演算子，$a^\dagger, b^\dagger, c^\dagger, \cdots$は空軌道に対する生成演算子です。$t_i^a, t_{ij}^{ab}, t_{ijk}^{abc}, \cdots$は後に述べる**カップルドクラスター方程式**を解いて求められる係数で，それぞれ一電子励起，二電子励起，三電子励起に対応する係数です。二電子励起の例を第 1 図に示します。



第 1 図： 二電子励起配置の例。

矢印はハートリーフォック軌道を占有する電子を表し，赤と青はスピンの向きに対応しています。

[†] 仮説（仮設と訳す場合もあります[8]）。

カップルドクラスター法では，このようなハートリーフォック軌道間の電子励起を考慮することにより，平均場近似を越えた多体効果を取り入れることができます。

カップルドクラスター法の利点は，(3)の形の多体波動関数が，size-extensivity を満足することにあります。Size-extensivity とは，系のサイズを大きくしていったときに，近似の程度が変わらない性質のことです。例えば，相互作用のない N 個の系を一つとみなして計算して得られるエネルギーが，個別に計算して得られるエネルギーの和(つまり 1 個のエネルギーの N 倍)となっていれば，size-extensivity を満たしていると言います。この性質は固体の計算などの電子数が多い系の計算では重要です。

さて，$t_i^a, t_{ij}^{ab}, t_{ijk}^{abc}, \cdots$ は，以下の時間に依存しないシュレディンガー方程式を解くことによって決定されます。

$$\hat{H}\exp(\hat{T})\left|\Psi_0\right\rangle = E\exp(\hat{T})\left|\Psi_0\right\rangle \qquad (5)$$

ここで $\hat{H}$ は第二量子化で表された電子系のハミルトニアン，E はエネルギー固有値です。この方程式の左から $\exp(-\hat{T})$ を掛け，さらに一電子励起関数 $\left|\Psi_i^a\right\rangle$，二電子励起関数 $\left|\Psi_{ij}^{ab}\right\rangle$，三電子励起関数 $\left|\Psi_{ijk}^{abc}\right\rangle$，…を射影すると次の連立方程式(カップルドクラスター方程式)が得られます[9]。

$$\left\langle\Psi_i^a\right|\exp(-\hat{T})\hat{H}\exp(\hat{T})\left|\Psi_0\right\rangle = 0, \qquad (6)$$

$$\left\langle\Psi_{ij}^{ab}\right|\exp(-\hat{T})\hat{H}\exp(\hat{T})\left|\Psi_0\right\rangle = 0, \qquad (7)$$

$$\left\langle\Psi_{ijk}^{abc}\right|\exp(-\hat{T})\hat{H}\exp(\hat{T})\left|\Psi_0\right\rangle = 0, \qquad (8)$$

$$\cdots$$

これらの方程式の左辺を展開すると，係数 $t_i^a, t_{ij}^{ab}, t_{ijk}^{abc}, \cdots$ (t と略記) とハートリーフォック軌道を基底にとって表した**電子間反発積分（二電子積分）**の積で表される沢山の項の和が出てきて，係数 t に対する連立非線形方程式になっていることを示すことができます。このことについては後でもう一度触れます。この方程式を解く方法としては，得られた式をさらに t に対する漸化式に変形し，逐次法を用いて t を求める方法が用いられます。このようにカップルドクラスター法は，**変分原理**に基づいてシュレディンガー方程式を解く他の多くの電子状態計算手法とは大きく異なっています。

係数 t，すなわち $\hat{T}$ が決まれば，系の基底状態のエネルギーを次式で求めることができます。

$$\left\langle\Psi_0\right|\exp(-\hat{T})\hat{H}\exp(\hat{T})\left|\Psi_0\right\rangle = E \qquad (9)$$

さて，励起演算子 $\hat{T}$ は，一電子励起，二電子励起，三電子励起，…というように，系の電子の数に応じて無限項の和になります。もし，可能な全ての励起演算子を考慮できれば，厳密解に到達することができると考えられる訳ですが，電子が数個程度の場合ならばともかく，一般の場合はそうはいきません。幸い二電子励起まで考慮すれば，かなり精度の高い波動関数が得られることがわかっているので，$\hat{T}_3$ 以降を打ち切る近似（**CCSD 近似**と呼ぶ）あるいはその派生形がよく

用いられます。その場合でも，求めなければいけない係数 t の数は相当の数になります。例えば，被占軌道及び空軌道の数をそれぞれ 100 個とした場合， t_{ij}^{ab} の数は 10e+8 個となり， t_{ij}^{ab} に対する連立方程式の数も同じだけ必要になります。そんなにたくさんの未知数があって，しかも非線形になっているこの方程式の解が無事に求まるのかどうか心配になりますが，多くの場合，逐次法で解くときちんと収束します。ただしそのためには，ハートリーフォックの波動関数から作られたスレーター行列式が，多体波動関数の十分によい近似になっている必要があります。

さて，カップルドクラスター法では励起状態を求めることもできます。いくつかの方法が提案されていますが，ここではそのうちの一つである equation-of-motion (EOM) という方法[9]に従って説明します。まず，(6)式の左辺のブラケットの中にある演算子 $\exp(-\hat{T})\hat{H}\exp(\hat{T})$ を一電子励起関数，二電子励起関数，三電子励起関数，…等々を基底にとることによって行列表示してやると，係数 t の要素の数と同じ次数をもつ巨大な非対称行列が得られます。励起状態はこの行列の固有値問題を解くことによって求めることができます。この行列の各々の要素は，カップルドクラスター方程式と同様に，係数 t と二電子積分の間の積和で表されるたくさんの項の和で表されるので，非常に多くの計算量が必要になります。

ここまではカップルドクラスター法の一般論を述べてきました。次節では，カップルドクラスター法を一次元系（高分子）に適用する場合について，具体的な計算手順を述べます。

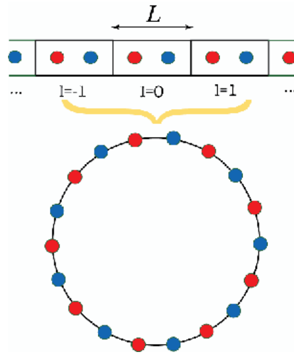
３． カップルドクラスター法の一次元系への適用

カップルドクラスター法を一次元系に適用するには，もちろんまず最初に，一次元系のハートリーフォック軌道（周期系なので**結晶軌道**とも呼ぶ）を求める必要があります。周期的境界条件を満足するハートリーフォック軌道の波動関数は，**波数**を k，軌道の番号を i とすると次の形で書けます[10]。

$$\varphi_i^k(\mathbf{r}) = \frac{1}{\sqrt{N_C}} \sum_{l=-l\max}^{l\max} e^{ikl} \sum_p C_{pi}^k \chi_p^l \qquad (10)$$

ここで χ_p^l は l 番目のセルに置かれた p 番目のガウス型基底関数を表し，また，C_{pi}^k は結晶軌道の展開係数（複素数）を表します。上式が普通の原子や分子の計算と異なっている点は，波数 k が新たに加わり，それに依存する e^{ikl} という位相因子がかかること，それから，格子の番号 l に対する和が新たに加わることです。

式(10)の N_C はセルの数を表しており，これは第２図に示すように，波動関数を周期的に並んだ有限個のセルで表現していることによるもので，式(10)の場合では $N_C = 2l_{\max} + 1$ となります。またこのことに対応して，波数 k は離散的な値，すなわち $k = 2\pi j / L / N_C$ として，$j = 0,1,2,\ldots,N_C - 1$ の値しかとることができません（L は単位セルの長さ）。



第２図：一次元周期的境界条件の模式図。

ガウス型基底関数を用いて一次元周期系の計算を行なう場合には，ある有限の個数（N_C）の単位セル（長さL）が仮想的に円周上に並んでいると見立てて周期的境界条件を表現します。上の図は$N_C = 9$ （$l_{\max} = 4$）の場合です。赤と青の丸は原子を表しています。

結晶軌道を求めるには，フォック演算子をガウス型基底関数で行列表示したもの（**フォック行列**）と基底関数の**重なり積分行列**を，波数k毎に求める必要があります。それらを以下に示します。

$$F_{rs}^{k} = \sum_{l} e^{ikl} \left(h_{rs}^{0l} + J_{rs}^{0l} - K_{rs}^{0l} \right) \qquad (11)$$

$$S_{rs}^{k} = \sum_{l} e^{ikl} S_{rs}^{0l} \qquad (12)$$

$h_{rs}^{0l}, J_{rs}^{0l}, K_{rs}^{0l}$は，フォック演算子の一電子項，クーロン項，交換項を，それぞれガウス型基底関数χ_r^0とχ_s^lの行列要素として表したもので，S_{rs}^{0l}は重なり積分$\int \chi_r^0(\mathbf{r}) \chi_s^l(\mathbf{r}) d\mathbf{r}$です。

これらの中で計算量が最も多いのは，J_{rs}^{0l}およびK_{rs}^{0l}の部分です。これらは密度行列$P_{tu}^{l} = \frac{2}{N_C} \sum_{i}^{occ} \sum_{k}^{BZ} C_{it}^{k*} C_{iu}^{k} e^{ikl}$と二電子積分の積になっており，次式のように計算されます。

$$J_{rs}^{0l} = \sum_{mn} \sum_{tu} P_{tu}^{n} (r^0 s^l | t^m u^n) \qquad (13)$$

$$K_{rs}^{0l} = \frac{1}{2} \sum_{mn} \sum_{tu} P_{tu}^{n} (r^0 t^m | s^l u^n) \qquad (14)$$

ここで二電子積分は以下のように定義されます。

$$(r^0 s^l | t^m u^n) \equiv \int \chi_r^0(\mathbf{r}_1) \chi_s^l(\mathbf{r}_1) \frac{1}{r_{12}} \chi_t^m(\mathbf{r}_2) \chi_u^n(\mathbf{r}_2) d\mathbf{r}_1 d\mathbf{r}_2 \qquad (15)$$

式(13)，(14)はr,s,lの各組に対してm,n,t,uの四重ループを回す**積和演算**になっています。全部で７重のループになるので一見大変そうですが，並列プログラミングを用いて容易に並列化することができます。例えば，（15）式の二電子積分を(l,m,n)のラベルで分割して複数のプロセッサに割当て，式(13)，(14)の計算をそれぞれのプロセッサで独立に行ない，最後に，各プロセッサのJ_{rs}^{0l}およびK_{rs}^{0l}を足し合わせれば計算が完了します。

フォック行列が求まれば，あとはフォック行列と重なり積分行列の一般化固有値問題を解くことによって，ハートリーフォック結晶軌道を求めることができます。この固有値問題の規模は，計算の対象となる系のサイズを N とする時，N^3 に比例して大きくなりますが，カップルドクラスター方程式を解く部分に比べればずっと小さい計算量なので，大した問題ではありません。

カップルドクラスター法の計算に進むためには，二電子積分をハートリーフォック結晶軌道で表したものに変換する必要があります。このプロセスのことを**積分変換**と呼びます。以下にその二電子積分の定義と変換式を示します[4, 11-13]。

$$\left\langle i(k_1)j(k_2)\middle|k(k_3)l(k_4)\right\rangle \equiv \int \phi_i^{k_1*}(\mathbf{r}_1)\phi_j^{k_2*}(\mathbf{r}_2)\frac{1}{r_{12}}\phi_k^{k_3}(\mathbf{r}_1)\phi_l^{k_4}(\mathbf{r}_2)d\mathbf{r}_1 d\mathbf{r}_2$$

$$=\frac{1}{N_C}\delta_{T(k_1+k_2),T(k_3+k_4)}\sum_{l_2 l_3 l_4} e^{-ik_2 l_2}e^{ik_3 l_3}e^{ik_4 l_4}\times\sum_{rstu} C_{ir}^{k_1*}C_{jt}^{k_2*}C_{ks}^{k_3}C_{lu}^{k_4}(r^0 s^{l_3}|t^{l_2}u^{l_4}) \quad (16)$$

上の式の中に現れる $T(k_1+k_2)$ は，k_1+k_2 を**第一ブリルアンゾーン**に還元するための関数です。上の式は，波数 k_1+k_2 と k_3+k_4 とがそれぞれ第一ブリルアンゾーンに還元したときに，等しくなっていない場合には0になります。

ここで，二電子積分の略記法について触れます。式(15)と(16)は，どちらも二電子積分を表す略記法ですが，よく見ると $\mathbf{r}_1$ と $\mathbf{r}_2$ の順番が違っていることがわかると思います。一般に，(15)のような二電子積分の記法を Mulliken's notation，(16)のような記法を Dirac's notation と呼び，括弧の形を変えることで区別しています。なお，(16)式では，ブラとケットの二つの ϕ_i^k の積は反対称化されていませんが，反対称化された積（スレーター行列式）を使いたい場合があります。その場合は式(16)の略記法に代えて，$< i(k_1)j(k_2)\| k(k_3)l(k_4) >$ のように表します。これはカップルドクラスター方程式等を記述するときに使います。

さて，式(16)の計算もフォック行列の計算と同様，かなりループのネストが深い計算となっており，i,j,k,l,k_1,k_2,k_3,k_4 の組み合わせに対して，r,s,t,u,l_2,l_3,l_4 の7重ループの積和計算が必要です。しかし，こちらもフォック行列の計算と同様，簡単に演算を並列化することができます。式(16)は全体にδ関数がかかっているため，全ての波数 k の組み合わせの項を計算する必要はなく，$(N_C)^3$ 個の項しか値を持ちません。そこで，値を持つ波数 k の組み合わせをテーブル化しておき，それをいくつものプロセッサに分割して割り当てて式(16)を分担して計算させれば，簡単に並列計算を実現できます。なお，フォック行列と積分変換の計算は，どちらも並列化効率は非常に良好です。

ハートリーフォック結晶軌道と，それによって表した二電子積分が求まれば，カップルドクラスター法の計算に進むことができます。ところで，筆者が最初に作ったカップルドクラスター法のコードは，分子の計算のために書いたものでした。分子の場合，軌道の展開係数と二電子積分は実数となり，複素数である式(16)をそのまま適用することはできません。幸い，波数 k と $-k$ の結晶軌道の展開係数は互いに複素共役の関係にありますので，それら結晶軌道のユニタリー変換（和と差）を作って結晶軌道を再定義し，これらを基底にとって式(16)の二電子積分を再度変換すれば，二電子積分は実数となります[4]。この方法は，波数 k がもつ対称性を計算に利用することができなくなる欠点がありますが，分子用に作られたコードをそのまま用いることができるという便利さがあるために，筆者はこの方法を用いています。

以下の説明では，式(16)の二電子積分に出てくる$i(k)$の波数kの表示を省略し，一つのシンボル（i,j,a,bなど）で軌道番号と波数のペアを表すことにします。

さて，それではカップルドクラスター方程式の計算に進みます。この方程式は CCSD 近似の場合，式 (6),(7)だけで表されます。ハミルトニアン$\hat{H}$を（反対称化された）二電子積分を用いて**第二量子化表示**で表した式：

$$\hat{H} = \sum_{pq} h_{pq} p^{\dagger} q + \frac{1}{4} \sum_{pqrs} \langle pq \| rs \rangle p^{\dagger} q^{\dagger} sr \tag{17}$$

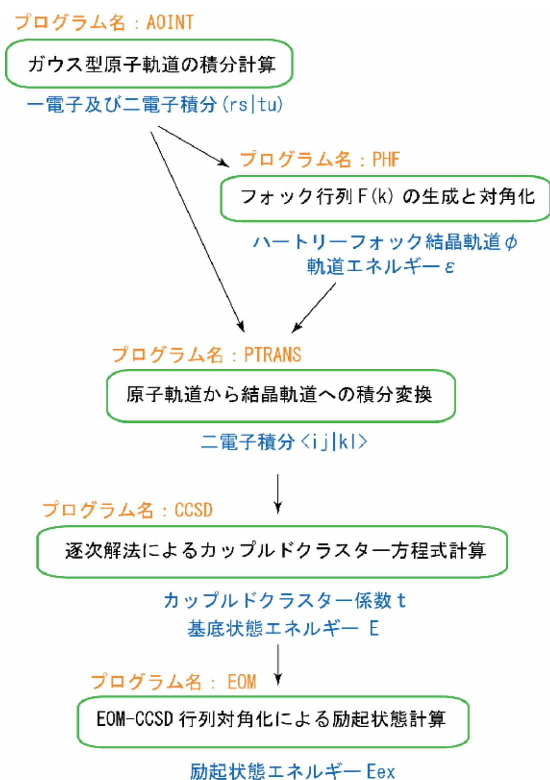
を式(6),(7)に代入して展開すると，二電子積分$< pq \| rs >$，係数t_i^a，t_{ij}^{ab}の積でできた多数の項の和で表された非線形連立方程式が得られます。この方程式を逐次法による繰り返し計算によって解くのが，カップルドクラスター計算の主要部分になります。これらの方程式は非常に長い式になるので，そのうちの一つの項だけを挙げるとこんな風です。

$$\sum_{mef} \langle am \| ef \rangle t_i^f t_{mj}^{eb} \tag{18}$$

これは,(6) 式から出てくる項のほんの一部です。この式も実際には，i,j,a,bとm,e,fが軌道の番号のほかにスピンの自由度を持っていますので，スピンをあらわに表示すると，さらに複雑な式になります。カップルドクラスター方程式に現れるほとんどの項の基本的な構造は，上と同様の形，すなわち，二電子積分と係数 t の積という形になっていて，異なっているのは，積和を取る添字の位置や積和をとる係数 t の種類や数です。これらは多体論で使われるダイアグラムを用いて表すこともできます。

さて，式(18)の計算は，m,e,fに関する積和計算を全てのi,j,a,bの組み合せに対して行なう必要があります。計算量は，結晶軌道と波数の数の積をNとすると，だいたいN^6に比例します。また，二電子積分やt_{ij}^{ab}には，N^4のオーダーの記憶量が必要になります。式(18)の計算は，i,j,a,bの組それぞれにおいて独立となっているので，一見，並列化が容易なように見えるのですが，二電子積分やt_{ij}^{ab}へのアクセスのパターンが複雑で，しかもそれが項によって大きく異なること，これらのパーツ及び積和の結果を格納しておく一時的な領域が，非常に大きなメモリを消費することの二つの相乗効果で，実際にはそれほど単純ではありません。次節では式(18)の実際のコーディング例を示し，それらの性能を比較することにします。

この節を終わる前に，第 3 図に筆者が作成したカップルドクラスター法のプログラムの全体図を示します。プログラムは全て FORTRAN で記述し，並列化は OpenMP を用いて行っています。OpenMP は**共有メモリ型の並列コンピュータ**上で利用することができますが，一般には，PC を集めたいわゆる**クラスター型の並列コンピュータ**では利用することが出来ないので注意が必要です。OpenMP は，プログラムに数行の指示行を加えるだけで並列化が出来るため，作業効率が良く，さらに，指示行の挿入でプログラムの意味が変わることがないという利点があり，筆者は気に入っています。OpenMP の使用法の詳細については，本特集号の記事をご覧ください。また，SGI 社のチュートリアル等の資料[14]も分かり易くてお勧めです。



第３図：　カップルドクラスター法プログラムの全体図。

４．カップルドクラスター法における積和計算のプログラミング

この節では，カップルドクラスター法で必要となる積和計算について，式(18)の計算（正確にはその一部）に焦点を当てて，プログラミング例を紹介することにします。（なお，以下に挙げるプログラムは本稿のために，元のプログラムに手を加えて修正したものです。）

式(18)をスピンの種類をあらわに表示した書き方に改め，その一部を取り出したものを以下に示します。

$$w_{i\bar{j}}^{a\bar{b}} = w_{i\bar{j}}^{a\bar{b}} + 2\sum_{mef}(af|em)t_i^f\left(t_{m\bar{j}}^{e\bar{b}} + t_{mj}^{eb}\right) - 2\sum_{mef}(ae|fm)t_i^f t_{m\bar{j}}^{e\bar{b}} \qquad (19)$$

$$w_{\bar{i}\bar{j}}^{b\bar{a}} = w_{\bar{i}\bar{j}}^{b\bar{a}} - 2\sum_{mef}(ae|fm)t_i^f t_{m\bar{j}}^{b\bar{e}} \qquad (20)$$

これらの式はカップルドクラスター係数 t を求める収束計算のサイクルで，毎回一度実行されます。各イタレーションで，t に初期値を入れて上の式を計算し，さらに他の項も同様の計算をして，最終的に得られた $w_{i\bar{j}}^{a\bar{b}}$ が新しい $t_{i\bar{j}}^{a\bar{b}}$ の初期値となります。上の式でバーの無し，有りはスピンの上向きと下向きに対応しています。t_{ij}^{ab} と $t_{i\bar{j}}^{a\bar{b}}$ はスピンをあらわに表示したために出てきたもので，両者は全く異なるものであることに注意してください。$(pq|rs)$ の形をしているものは，Mulliken の記法による二電子積分です。また，i,j,m,n は占有軌道，a,b,e,f は空軌道に対する添字を表しています。以下ではそれぞれの軌道の数を NOCC,NVIR とします。

式(19), (20)は，二電子積分$(pq|rs)$とt_i^aとt_{ij}^{ab}（または$t_{i\bar{j}}^{a\bar{b}}$）の積和をとり，$w_{i\bar{j}}^{a\bar{b}}$に加算または減算する形になっています。ここで注意しなければならないことは，この計算は２段階に分けて実行する必要があるという点です。例えば式(20)を計算する場合，以下のようにしてはいけません。

do i=1,NOCC; do j=1,NOCC; do a=1,NVIR; do b=1,NVIR;

$$w_{i\bar{j}}^{b\bar{a}} = w_{i\bar{j}}^{b\bar{a}} - 2\sum_{mef}(ae|fm) \times t_i^f \times t_{m\bar{j}}^{b\bar{e}}$$

enddo; enddo; enddo; enddo;

プログラム１：ダメなループの回し方

この方法では全体として７重のループとなり，計算量はNOCC^3*NVIR^4 となってしまいます。これは以下のように計算する方が得です。

do a=1,NVIR; do e=1,NVIR; do i=1,NOCC; do m=1,NOCC;

$$A(a,e,i,m) = 2\sum_f (ae|fm) \times t_i^f$$

enddo; enddo; enddo; enddo;

do i=1,NOCC; do j=1,NOCC, do a=1,NVIR;, do b=1,NVIR;

$$w_{i\bar{j}}^{b\bar{a}} = w_{i\bar{j}}^{b\bar{a}} - \sum_{me} A(a,e,i,m) \times t_{m\bar{j}}^{b\bar{e}}$$

enddo; enddo; enddo; enddo;

プログラム２：　正しいループの回し方

この方法だと前半のループは５重，後半が６重となり，後半部分の計算量は NOCC^3*NVIR^3 となります。NOCC や NVIR は計算する系の大きさによっては数百に達しますので，計算量は最初のやり方に比べて圧倒的に少なくてすみます。掛け算の順序は，t_i^aを先に掛ける方法のほかに後で掛ける方法もあり，その場合の計算量は NOCC^2*NVIR^4 となります。どちらが得かは NOCC と NVIR の大小関係に依存します。一般には NOCC<NVIR となっていることが多いため，上で示したようにt_i^aを先に掛ける方が計算量が少なくなります。

さて，上の計算は前半・後半ともに単なる行列の積で表すことができますので，線形演算の標準パッケージである BLAS ライブラリを使って計算はおしまい，と言いたいところですが，ワーク配列である A は非常に大きな記憶領域(例えば NOCC=NVIR=256 の時で 32GB)を占めるので，少しでもメモリを節約するためには何らかの工夫が必要です。そこで以下では５つのプログラム例を示し，その性能を比較することにします。

４．１　ワーク配列を全く使わないプログラム(SIMPLE.f)

まず，ワーク配列を全く使わない例をお見せします。

プログラム例1：　SIMPLE.f

```
 1      do mm=1,NOCC
 2        do me=1,NVIR
 3 !$OMP PARALLEL DO DEFAULT(PRIVATE),SHARED(ia,idx,t1,t2s,t2a,w2a,v,
 4 !$OMP& me,mm)
 5          do mi=1,NOCC
 6            do ma=1,NVIR
 7              mea=idx(ma+NOCC,me+NOCC)
 8              sum1=0d0
 9              sum2=0d0
10              do mf=1,NVIR
11                mem=ia(me+NOCC)+mm
12                mfm=ia(mf+NOCC)+mm
13                mfa=idx(mf+NOCC,ma+NOCC)
14                sum1=sum1+v(mfa,mem)*t1(mf,mi)
15                sum2=sum2+v(mfk,mem)*t1(mf,mi)
16              enddo
17              do mj=1,NOCC
18                do mb=1,NVIR
19                  w2a(mb,ma,mj,mi)=w2a(mb,ma,mj,mi)
20     &                          +2.0d0*(sum1*(t2s(mb,me,mj,mm)
21     &                                        +t2a(mb,me,mj,mm))
22     &                                  -sum2*t2a(mb,me,mj,mm))
23                  w2a(ma,mb,mj,mi)=w2a(ma,mb,mj,mi)
24     &                          -2.0d0*sum2*t2a(me,mb,mj,mm)
25                enddo
26              enddo
27            enddo
28          enddo
29        enddo
30      enddo
```

このプログラムでは二電子積分 $(pq|rs)$ を２次元配列 v(pq,rs)に，$t_i^a,t_{ij}^{ab},t_{\bar{i}\bar{j}}^{a\bar{b}},w_{\bar{i}\bar{j}}^{a\bar{b}}$ を２次元及び４次元配列 t1(mi,ma),t2s(mb,ma,mj,mi),t2a(mb,ma,mj,mi),w2a(mb,ma,mj,mi)に，それぞれ割り当てています。このプログラムでは v と t1 の積を sum1,sum2 にキープすることで，ワーク配列を使わずに済ませています。3,4 行目は OpenMP の指示制御文で，ここではループ変数 mi について並列化しています。外側の me または mm の do 文の前に同様のやり方で制御文を入れて並列化するのは正しくないことに注意してください。これは，異なるスレッドが w2a の同じ要素をアクセスして更新する可能性が出てくるからです。

二電子積分を格納している二次元配列 v(pq,rs)については少し説明が必要です。ここで p,q,r,sは軌道全体に対する添字を表し,最初の1,2,..,NOCCが占有軌道を,そしてNOCC+1, NOCC+2,..,NOCC+NVIRが空軌道を指すとします。ところで,Mullikenの二電子積分$(pq|rs)$は,次のような添字に関する対称性：$(pq|rs)=(qp|rs)=(pq|sr)=(qp|sr)$，$(pq|rs)=(rs|pq)$を持っています。前者の性質を利用し，$p \geq q$および$r \geq s$の組（要するに下三角行列の部分）を，一次元化した添字 pq と rs を用いて，二次元配列 v(pq,rs)に格納することにすれば，記憶容量を１／４に節約することができます。7,11-13 行目は，このルールに基づいて二電子積分 v の添字を計算している部分です。配列 ia と idx にはそれぞれ,ia(i)=i*(i-1)/2,idx(p,q)= ia(max(p,q))+ min(p,q)が格納されています。$p \geq q$の場合は pq=ia(p)+q で，また，p と q の大小関係が不定の場合は pq=idx(p,q)によって配列要素のアドレスを計算しています。

４．２　ワーク配列を使うプログラム(ARRAY.f)

さて，前のプログラム(SIMPLE.f)はコンパクトで見た目にも単純でいいのですが，配列 w2a に書き込むループが mi,ma,mj,mb の順番に入れ子になっていて,メモリアクセスの順番が連続的ではありませんでした。FORTRAN の場合，配列の最初の添字が最内側のループ変数になるようにして，２番目,３番目,…の添字を順に外側にすれば，効率の良いメモリアクセスになります。一般にコンピュータでは,

今参照している場所から離れた記憶要素をアクセスするのは時間がかかる

という性質があります。これは,コンピュータは演算装置の下にレジスタ,キャッシュ,メモリ,ディスクといった記憶装置がぶら下がっていて階層構造を作っており,下の階層に行くほどデータの入出力に時間がかかり,それぞれの階層に入りきらない部分はその下の階層に溢れてしまうためです。さらに並列コンピュータの場合は，これらの他にネットワークの階層が付け加わります。プログラムを書く場合は，コンピュータが階層構造の塊であることを常に念頭に置く必要があります。

それでは速いプログラムを書くにはどうしたら良いでしょうか？そのためにはまず,階層構造を越えるようなデータのやり取りを極力しないことです。しかし大きなデータを扱う時は，階層構造を越えるようなことが必ず起こります。その場合でもコンピュータにたいへん都合の良い仕掛けがあって，階層構造の間のギャップを隠蔽するための何らかの緩衝機構（バッファ）が用意されているものです。速いプログラムは，そのような緩衝機構を最大限利用し，階層構造の間のやり取りが実行速度のボトルネックにならないように工夫されています。そのために必要なことは何かと言うと，それは実にとても簡単なことで,

同じ要素，または近くにある記憶要素を可能な限り何度も再利用すること

というたった一つのことだけです。

さて，SIMPLE.f を次のように書き直します。

プログラム例２：　ARRAY.f

```
 1       dimension wk1(NVIR,NOCC),wk2(NVIR,NOCC)
 2
 3       do mm=1,NOCC
 4         do me=1,NVIR
 5 !$OMP PARALLEL DO DEFAULT(PRIVATE),SHARED(ia,idx,t1,t2s,t2a,w2a,v,
 6 !$OMP& wk1,wk2,me,mm)
 7           do mi=1,NOCC
 8             do ma=1,NVIR
 9               mea=idx(ma+NOCC,me+NOCC)
10               sum1=0d0
11               sum2=0d0
12               do mf=1,NVIR
13                 mem=ia(me+NOCC)+mm
14                 mfm=ia(mf+NOCC)+mm
15                 mfa=idx(mf+NOCC,ma+NOCC)
16                 sum1=sum1+v(mfa,mem)*t1(mf,mi)
17                 sum2=sum2+v(mfm,mea)*t1(mf,mi)
18               enddo
19               wk1(ma,mi)=sum1
20               wk2(ma,mi)=sum2
21             enddo
22           enddo
23 !$OMP PARALLEL DO DEFAULT(PRIVATE),SHARED(ia,idx,t1,t2s,t2a,w2a,v,
24 !$OMP& wk1,wk2,me,mm)
25           do mi=1,NOCC
26             do mj=1,NOCC
27               do ma=1,NVIR
28                 do mb=1,NVIR
29                   sum=wk1(mb,mj)*(t2s(me,ma,mm,mi)+t2a(me,ma,mm,mi))
30      &               -wk2(mb,mj)*t2a(me,ma,mm,mi)
31      &               -wk2(ma,mj)*t2a(mb,me,mm,mi)
32                   w2a(mb,ma,mj,mi)=w2a(mb,ma,mj,mi)+2.0d0*sum
33                 enddo
34               enddo
35             enddo
36           enddo
37         enddo
38       enddo
```

このプログラムは，SIMPLE.f の一時変数 sum1,sum2 の代わりに二次元配列 wk1,wk2 を使っていて，mi に関するループを二つに分割して後半のループの順番が w2a 配列のメモリ配置の順番と一致するようにしたものです。ループを二つに分割したことに伴って，OpenMP の指示制御文も一つ増えています。

このプログラムにはもう一つ，SIMPLE.f と異なる点があります。それは 29-31 行目の t2s,t2a の添字の並び方です。SIMPLE.f では添字 mm の位置が最後であったのに対して，上のプログラム(ARRAY.f)では３番目になっていて，その他の添字の順番も入れ替わっています。これは，$t_{ij}^{ab}, t_{i\bar{j}}^{a\bar{b}}$ が添字 i,j と a,b の同時の交換に関して対称であるというカップルドクラスター係数の性質を利用して，添字の順番を入れ替えたものです。このプログラムの後半部分では，mi の添字でループを並列化しているので，共有している配列に複数の**スレッド**（並列動作の単位）から同時にアクセスが起きた時に，参照位置が互いに近くなって競合が起きることがないように配慮する必要があります。ここでは t2s,t2a 配列の添字を入れ替えて mi を最後に持ってくることによって，そのようなアクセスの競合が起きることを抑制しています。

共有メモリ型の並列コンピュータではメモリアクセスの競合に注意が必要

４．３　ブロッキング手法を使うプログラム(BLOCK.f)

前のプログラム(ARRAY.f)では，w2a に対するアクセスは理想的になったのですが，t2s,t2a および v に対するアクセスは連続的ではありません。また，ループの全体の外側には mm と me のループがあって，w2a 配列全体を NOCC*NVIR の回数だけスキャン（舐めるともいう）していて，**「同じ要素，または近くにある記憶要素を可能な限り何度も再利用すること」**という原則に反しています。このような時に使われるのが，**ブロッキング**（または**ブロック化**）という手法です。ここでは，mm と me のループをブロック化した例を示します。

プログラム例３：　BLOCK.f

```
 1      parameter(NBLK=255)
 2      dimension wk1(NBLK,NVIR,NOCC),wk2(NBLK,NVIR,NOCC)
 3
 4      nme=0
 5      do mm=1,NOCC
 6        do me=1,NVIR
 7          nme=nme+1
 8          mne(nme)=me
 9          mnm(nme)=mm
10        enddo
11      enddo
```

```
12
13      nme_total=nme
14      nme_blk=(nme_total-1)/NBLK+1
15
16      do iblk=1,nme_blk
17        nme_start=NBLK*(iblk-1)+1
18        nme_end=min(nme_start+NBLK-1,nme_total)
19
20 !$OMP PARALLEL DO DEFAULT(PRIVATE),SHARED(ia,idx,t1,t2s,t2a,w2a,v,
21 !$OMP& wk1,wk2,mne,mnm,nme_start,nme_end)
22        do mi=1,NOCC
23          do ma=1,NVIR
24            nme=0
25            do nme_ptr=nme_start,nme_end
26              nme=nme+1
27              mm=mnm(nme_ptr)
28              me=mne(nme_ptr)
29              mea=idx(ma+NOCC,me+NOCC)
30              sum1=0d0
31              sum2=0d0
32              do mf=1,NVIR
33                mem=ia(me+NOCC)+mm
34                mfm=ia(mf+NOCC)+mm
35                mfa=idx(mf+NOCC,ma+NOCC)
36                sum1=sum1+v(mfa,mem)*t1(mf,mi)
37                sum2=sum2+v(mfk,mea)*t1(mf,mi)
38              enddo
39              wk1(nme,ma,mi)=sum1
40              wk2(nme,ma,mi)=sum2
41            enddo
42          enddo
43        enddo
44
45 !$OMP PARALLEL DO DEFAULT(PRIVATE),SHARED(ia,idx,t1,t2s,t2a,w2a,v,
46 !$OMP& wk1,wk2,mne,mnm,nme_start,nme_end)
47        do mi=1,NOCC
48          do mj=1,NOCC
49            do ma=1,NVIR
50              do mb=1,NVIR
51                nme=0
```

```
52              sum=0d0
53              do nme_ptr=nme_start,nme_end
54                nme=nme+1
55                mm=mnm(nme_ptr)
56                me=mne(nme_ptr)
57                sum=sum+wk1(nme,mb,mj)*(t2s(me,ma,mm,mi)
58     &                                +t2a(me,ma,mm,mi))
59     &                    -wk2(nme,mb,mj)*t2a(me,ma,mm,mi)
60     &                    -wk2(nme,ma,mj)*t2a(mb,me,mm,mi)
61              enddo
62              w2a(mb,ma,mj,mi)=w2a(mb,ma,mj,mi)+2.0d0*sum
63            enddo
64          enddo
65        enddo
66      enddo
67    enddo
```

このプログラムでは，mm,me の組を一次元化して，それを NBLK の単位のかたまりに分割して処理しています。（この例では NBLK は 255 に固定しています。）プログラムの最初で me と mm の値をテーブル配列 mne,mnm にストアしています。こうすることで，今処理している me と mm の値がテーブルを引けばすぐにわかるようになっています。

このようなプログラムの書き換えによって mm,me のループを前の例よりも内側に持ってくることができます。この利点は三つあります。一つは w2a 全体をスキャンする回数を減らせること。もう一つはスレッドの起動回数を少なくできること。そしてもう一つは mm,me の組をブロックとしてループの内側で扱うことで，**「メモリアクセスの局所化と同じ要素の再利用」**が容易になることです。これは，このプログラムの後半部分を BLAS ライブラリなどの線形計算パッケージに置き換えることで簡単に取り入れることができます。それを次節に示します。

４．４　ブロッキングとライブラリを併用するプログラム(BLAS.f,MATMPP.f)

前節に書いた通り，BLOCK.f の後半部分は BLAS ライブラリの dgemm 等の行列積サブルーチンを使って書き換えることができます。dgemm 等の最適化された行列積のサブルーチンでは，ブロッキングの手法や**ループアンローリング**といった手法を駆使して，一度アクセスした配列要素が出来る限り再利用されるようなアルゴリズムで計算を行なうため，場合によっては，ほぼ理論ピーク性能に近い速度を出すことができます。(行列積計算の高速化手法については，C や FORTRAN によるコードの事例解説[15,16]があります。）ちなみに，単純なベクトル a と b のスカラー積：

```
sum=0d0; do i=1,N; sum=sum+a(i)*b(i); enddo
```

の計算を単一プロセッサで行なう場合，スピードアップの意味で BLAS ライブラリなどを呼び出す意味は全くありません。これは，上のような演算では a(i) と b(i) の各要素は計算中に一度

しか参照されず，メモリのバンド幅だけで処理速度が決まるため，工夫の余地がないからです。

以下ではBLASのdgemmを用いてBLOCK.fを書き換えた例について，追加配列の宣言部と後半（BLOCK.fの45行目以降）の置き換え部分だけを示します。

プログラム例４：　BLAS.f（一部）

```
                        ...
         dimension wk3(NVIR,NBLK),wk4(NVIR,NBLK),wk5(NVIR,NBLK)
         dimension wk6(NVIR,NVIR),wk7(NVIR,NVIR),wk8(NVIR,NVIR)
                        ...
 1 !$OMP PARALLEL DO DEFAULT(PRIVATE),SHARED(ia,idx,t2s,t2a,w2a,v,
 2 !$OMP& wk1,wk2,mne,mnm,nme_start,nme_end)
 3        do mi=1,NOCC
 4          nme=0
 5          do nme_ptr=nme_start,nme_end
 6            mm=mnm(nme_ptr)
 7            me=mne(nme_ptr)
 8            nme=nme+1
 9            do ma=1,NVIR
10              wk3(ma,nme)=t2s(me,ma,mm,mi)+t2a(me,ma,mm,mi)
11              wk4(ma,nme)=t2a(me,ma,mm,mi)
12              wk5(ma,nme)=t2a(me,ma,mm,mi)
13            enddo
14          enddo
15          do mj=1,NOCC
16            call dgemm('N','N',NVIR,NVIR,nke,1d0,wk3,NVIR,wk1(1,1,mj),
17     &        NBLK,0d0,wk6,NVIR)
18            call dgemm('N','N',NVIR,NVIR,nke,1d0,wk4,NVIR,wk2(1,1,mj),
19     &        NBLK,0d0,wk7,NVIR)
20            call dgemm('N','N',NVIR,NVIR,nke,1d0,wk5,NVIR,wk2(1,1,mj),
21     &        NBLK,0d0,wk8,NVIR)
22            do ma=1,NVIR
23              do mb=1,NVIR
24                w2a(mb,ma,mj,mi)=w2a(mb,ma,mj,mi)
25     &                        +2.0d0*(wk6(ma,mb)-wk7(ma,mb)
26     &                        -wk8(mb,ma))
27              enddo
28            enddo
29          enddo
30        enddo
```

このプログラムでは，5 行目の nme_ptr のループが BLOCK.f の 53 行目のループに対応しています。t2s,t2a の参照部分は，mi,nme_ptr,ma の 3 重ループの内側になっており，BLOCK.f よりもはるかに少ない参照回数で済むようになったことがこの書き換えの大きなメリットです。

日立の SR11000 では，BLAS ライブラリのほかに MATRIX/MPP という日立製の行列計算ライブラリが用意されています。BLAS の dgemm に対応するルーチンは hdmffm です。インタフェースが dgemm と異なっており，転置行列の積を計算できないという欠点がありますが，上のプログラムはほとんど変更することなく，hdmffm を使ったバージョン(MATMPP.f，リストは省略)にも書き換えることができます。

次節ではこれまでに挙げた５つのプログラムの性能を比較することにします。

４．５　実行テスト結果

プログラムの実行テストは，産業技術総合研究所の SR11000 のほか，IBM p5-560Q，Apple Power Mac G5 Quad を用いました。各コンピュータの性能諸元と使用 OS，コンパイラと使用ライブラリを表１に示します。Power Mac G5 では，用いた IBM XL Fortran for Linux に BLAS ライブラリのサブセット（dgemm を含む）が付属していたので，それを利用しました。

	日立 SR11000 (モデル H1)	IBM p5-560Q	Apple Power Mac G5 Quad
CPU memory	Power4+ 1.7GHz 16proc，64GB(1 node)	Power5+ 1.5GHz 8proc，32GB	PowerPC 970 2.5GHz 4proc，16GB
理論性能	108.8 GFLOPS	48.0 GFLOPS	40.0 GFLOPS
OS	AIX 5L	AIX 5L	Fedora Core 5（PPC 版）
コンパイラ & ライブラリ	Optimizing FORTRAN90 V01-05-/B MATRIX/MPP V01-04	IBM XL Fortran V9.1 ESSL V4.2	IBM XL Fortran V10.1 for Linux (ライブラリ付属)

表 1:　テストに使用したコンピュータの性能緒元

テスト計算の対象としては，一次元に並んだ Be 原子を選びました。基底関数は最小基底(STO-3G)を用いています。計算規模が増加した場合の性能を評価するために，k 点の数を 15 と 31 に変えた二通りの計算を行ないました。軌道の数:(NOCC,NVIR)に換算すると，それぞれ(15,45)，(31,95)となり，後者の場合の計算量は前者のほぼ 83 倍になります。

テストは 4.1 から 4.4 節までに取り上げた５つのコード（SIMPLE，ARRAY，BLOCK，BLAS，MATMPP)を実際のカップルドクラスターのプログラムに組み込み，カップルドクラスター方程式を解く収束計算を 10 回繰り返すことで行ない，各々のコードの実行に要した時間を測定して積算しました。時間の測定は，SR11000 では xclock，p5-560Q と Power Mac G5 では IBM XL Fortran の system_clock サブルーチンを使いました。今回取り上げた式(19)と(20)は演算量があらかじめ決まっていますので，演算量を経過時間で割り算して GFLOPS 値を算出して比較しています。計算に要した時間は k 点が 15 点の場合で 0.7-17 秒，31 点の場合で 53-7000 秒程度でした。

なお，コンパイルオプションは，SR11000 では xf90 -Os -cyclic -64 -omp -save -lmatmpp_sc

-llapack_sc -lblas_sc -lf90c を用い，また，-omp の代わりに-noomp オプションを使って，OpenMP を利用せずに自動並列化機能を使った場合についても調べました。IBM XL Fortran では xlf_r -O3 -w -qsclk=micro -qtune -qhot -qcache -qunroll=yes -qextname -qsmp=omp -q64 を用い，p5-560Q の場合は-lessl，G5 の場合は-qarch=ppc970 を付け加えています。

それでは，k 点を変えた二つのケースに関する実行結果を以下（表２，３）に示します。

	SR11000 OpenMP	SR11000 自動並列化	p5-560Q	Power Mac G5 Quad
SIMPLE	7.6	7.7	4.3	1.5
ARRAY	21.8	23.1	12.5	2.6
BLOCK	14.0	14.2	6.7	3.2
BLAS	21.7	1.9	22.7	3.5
MATMPP	38.6	5.1	-	-

表２：　k 点が 15 点の場合の各プログラムの実行速度（単位は GFLOPS）

	SR11000 OpenMP	SR11000 自動並列化	p5-560Q	Power Mac G5 Quad
SIMPLE	1.4	1.3	1.7	0.3
ARRAY	10.8	5.2	14.1	2.4
BLOCK	12.9	13.1	5.6	3.0
BLAS	33.2	4.1	24.2	4.5
MATMPP	36.9	5.2	-	-

表３：　k 点が 31 点の場合の各プログラムの実行速度（単位は GFLOPS）

まず SR11000 で OpenMP を利用した場合の結果を見ると，SIMPLE の性能が非常に悪いことがわかります。システムのサイズを大きくするとその傾向は特に顕著になります。また，ARRAY の性能は，システムのサイズが小さいときはまあまあですが，サイズを大きくすると性能は半分近くになります。BLOCK，MATMPP ではシステムのサイズを大きくしてもさほど性能が変わりません。これは SIMPLE や ARRAY の性能がキャッシュの大きさに強く依存しており,扱うデータが大きくなるとキャッシュに入りきらずに性能が低下するのに対して，BLOCK や MATMPP ではブロッキング手法を使っているため，そのようなことがないのが理由です。MATMPP は今回テストした中で最も性能がよく，SR11000 の理論性能の 30%以上を出しています。BLAS の方は MATMPP に比べて性能が落ちています。詳しく調べてみたところ，SR11000 の BLAS ライブラリの dgemm は MATRIX/MPP ライブラリの hdmffm に比べて性能が悪く，特に計算サイズが小さい時の性能が良くないことがわかりました。SR11000 では BLAS の利用を避けた方が賢明のようです。（現在 BLAS ライブラリが業界標準になっていることを考えると，これはちょっと困ったことですが。）

SR11000 で自動並列化を使った場合を見ると，SIMPLE，ARRAY，BLOCK では OpenMP を使った場合と傾向が似ています。特に，計算サイズが小さい場合は，OpenMP を使った場合の性能と

大差がありません。ところがBLAS，MATMPPでは，自動並列化を使うと大きく性能が低下しています。この理由は良くわかりませんが，自動並列化はまだ万能ではないということが言えそうです。

p5-560Qの場合でもSIMPLEの性能が悪い点はSR11000と同じです。BLASの性能は非常に良好で，p5-560Qの理論性能のほぼ半分出ています。ARRAYもまあまあの性能を出していて，BLASの場合もそうですが，サイズを大きくした時に速度が向上している点が目を引きます。一方，BLOCKの性能はSR11000と大きく異なり，全く良くありません。

最後にPower Mac G5の場合です。系のサイズを大きくした時にSIMPLEの性能が大きく低下している点，ARRAYでは性能が向上している点はp5-560Qと共通です。ところがBLOCKでは，p5-560Qと同じIBMコンパイラを使っているにもかかわらず，ARRAYよりも良い性能が出ています。また，BLASの性能は他の三つのプログラムよりも良いのですが，他の機種で見られたほどの性能の向上はありませんでした。

さて，これらの結果を俯瞰すると，今回扱った計算はプログラムの書き方で大きく速度が異なること，特にブロック化と行列計算ライブラリを併用する方法が非常に有効であること，また，プログラムの書き換えによる効果の度合いは，使用するコンピュータやコンパイラ，系のサイズに強く依存するということがお分かりいただけると思います。ブロッキング手法等を持ち込むとプログラムは非常に複雑になり，労力もそれなりにかかりますので，どの程度のチューニングが必要かの判断は，プログラムの目的を十分考慮して考える必要があるでしょう。ここでは，プログラムが実際に何GFLOPS出ているかを知ることが，チューニングの必要性を客観的かつ適切に判断する上で非常に大切である，ということを強調しておきたいと思います。

それにしても，現在のコンピュータ（スパコンも含む）は，洗練された道具と言うには，いまだほど遠いものだと言えるでしょう。性能を引き出すためには，かなり細かいことをコンピュータに指示しなければなりません。プロセッサがベクトル型から現在のプロセッサに変遷してから，その傾向はさらに顕著になっている気がしています。スーパーコンピューティングがオリンピックならば（実際そういう側面もあると思いますが），性能を引き出すための細かい工夫は意味のあることでしょうが，できればそういったことはしないで済むに越したことはありません。もっとスマートなコンピュータの登場を望みたいものです。

なお，ライブラリを利用してプログラムを書く場合には，4.4のBLAS.fのように，特定のライブラリのサブルーチン呼び出しをコードの中に埋め込む方法を取らず，間にプログラマーが決めた別の名前のサブルーチン（ラッパーと言う）を入れて間接的に呼び出すような方法を取ることをお勧めします。この方法を使えば，ライブラリを変更しても（例えばBLAS->MATRIX/MPP），ソースコードの変更を最小限で済ますことができます。

5. おわりに

本稿では，カップルドクラスター法と高分子の計算への適用について概要を紹介し，カップルドクラスター係数の積和計算を取り上げて，そのプログラミング手法について紹介しました。このような積和計算は，カップルドクラスター法のような多体問題のダイアグラム計算だけではなく，様々な数値計算のアプリケーションに出てくると思います。拙稿で紹介した事例が，これを読まれた方の参考になれば幸いです。なお本稿ではカップルドクラスター法を高分子に応用した場合の成果については触れませんでした。それについては文献[4,13]をご覧ください。

今，次世代のスパコンでは数万以上のプロセッサが並列に動作することが想定されています。カップルドクラスター法の計算をそのようなシステムで効率よく行なうには，いろいろな問題を解決する必要があります。例えば，カップルドクラスター法で扱うカップルドクラスター係数や二電子積分は，系のサイズの４乗のオーダーで大きくなりますので，１ノード内のメモリに収まらない場合にどうするかを考慮しなければなりません。その場合，実際に１ノードが持つメモリ量によって取る戦略が変わってきます。またこのほかに，数万以上のプロセッサで並列化するためには，どういった並列要素を利用するかということも考える必要が出てくるでしょう。こういった問題は他の多くの電子状態計算法に共通のもので，今後の課題であると言えます。

なお，本稿の成果の一部は，日本学術振興会の科学研究費補助金特定領域研究「次世代共役ポリマーの超階層制御と革新機能」の援助を受けて得られました。また長嶋雲兵氏からは，本稿について有難いコメントを頂戴しました。この場を借りて感謝したいと思います。

参 考 文 献

[1] A・ザボ／N・S・オストランド：大野公男／阪井健男／望月祐志訳『新しい量子化学（上・下)』，東京大学出版会，1988.

[2] R・G・パール／W・ヤング：狩野覚／関元／吉田元二訳『原子・分子の密度汎関数法』，シュプリンガー・フェアラーク東京，1996.

[3] B.L. Hammond, W.A. Lester, P.J. Reynolds, *Monte Carlo Methods in Ab initio Quantum Chemistry* (World Scientific, 1994).

[4] H. Katagiri, J. Chem. Phys. **122**, 224901 (2005).

[5]F. Coester, Nucl. Phys. **7**, 421 (1957); F. Coester and H. Kümmel, Nucl. Phys. **17**, 477 (1960).

[6] J. Cizek, J. Chem. Phys. **45**, 4256 (1966); J. Cizek, Adv. Chem. Phys. **14**, 35(1969).

[7] http://www.emsl.pnl.gov/forms/basisform.html

[8] M･T･バチェラー:田崎晴明訳『ベーテ仮設の７５年』パリティ,Vol.22, No.09, 丸善,(2007).

[9] R.J. Bartlett, in *Modern Electronic Structure Theory Part II*, ed. D.R. Yarkony, (World Scientific, 1995), pp.1047-1131.

[10] C. Pisani, R. Dovesi, and C. Roetti, *Hartree-Fock Ab Initio Treatment of Crystalline Systems*, Lecture Notes in Chemistry Vol. 48 (Springer, Heidelberg, 1988).

[11] S. Suhai, Phys. Rev. B **27**, 3506 (1983).

[12] J.-Q. Sun and R. J. Bartlett, J. Chem. Phys. **104**, 8553 (1996).

[13] S. Hirata, R. Podeszwa, M. Tobita, R.J. Bartlett, J. Chem. Phys. **120**, 2581 (2004).

[14] http://www.sgi.co.jp/origin/ODP/documents/programming/openmp/index.html

[15] 前野，太田：情報処理学会研究報告(93-HPC-49)，Vol.93, No. 89, pp.1-8 (1993).

[16] 山本，小畑，村上，片桐，秦野：情報処理学会研究報告(95-HPC-55)，Vol.95, No.55, pp.57-64 (1995); 山本，秦野：名古屋大学大型計算機センターニュース Vol.26, No.3, pp.182-193 (1995).

第２部
T2K オープンスパコンの高速化技術

オープンスパコンのＯＳとアーキテクチャの基礎

松 葉 浩 也

東京大学情報基盤センター

1. はじめに

東京大学の情報基盤センターには 2007 年現在 SR11000/J2 が導入されており，2008 年度には「オープンスーパーコンピュータ」と呼ばれる新たなスーパーコンピュータが導入される。オープンスーパーコンピュータはその名の通り，一般的に仕様が公開されているハードウェアおよびオープンソースのソフトウェアを中心に構成されたスーパーコンピュータである。このオープンな性質により，ユーザーはより深くスーパーコンピュータを理解しその知識を最適化に役立てることができる。

本稿はスーパーコンピュータのアーキテクチャを理解する助けとなるよう，プロセッサ単体から並列計算機，通信の仕組みに至るまで，スーパーコンピュータに欠かせない技術を広く紹介するものである。本稿で扱える範囲は非常に限られているが，それでもスーパーコンピュータの動作原理を少しでも理解していただき，プログラムの最適化などに役立てていただければ幸いである。

2. プロセッサアーキテクチャの基礎

最初にスーパーコンピュータの最も基本的な構成要素であるプロセッサについて述べる。近年，スーパーコンピュータと一般のコンピュータとの違いは主にプロセッサの数であり，単体のプロセッサはスーパーコンピュータのものであってもパーソナルコンピュータのものであっても大きな差はない。したがってここで述べる内容はスーパーコンピュータのみならず，パーソナルコンピュータも含め，現在入手できるほとんどのコンピュータに当てはまるものである。

2.1 計算機の構成と動作原理

計算機（コンピュータ）はプロセッサ，メモリ，入出力機器，およびそれらを接続するためのチップセットから成る。これらは図 1 のように接続されている。

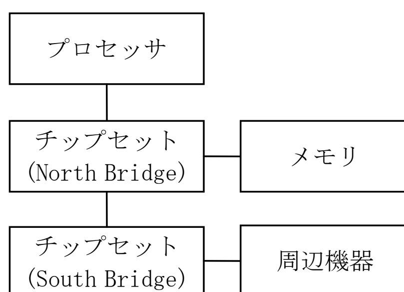


図 1　計算機の基本構成

まずプロセッサと直接接続されているのが North Bridge と呼ばれるチップセットであり，ここにはメモリコントローラと呼ばれるメモリアクセスのための機構が実装されている。North Bridge は伝統的に

はプロセッサとは別のチップであったが，近年では高速化のためにこの機能をプロセッサの中に内蔵しているものもある。North Bridge の先には South Bridge と呼ばれるチップセットが接続されている。South Bridge の主な役割は周辺機器との通信の窓口となることである。ディスク，ネットワーク，USB，拡張スロットに搭載する様々なボードはすべて South Bridge に接続されている。このように計算機は高速なプロセッサとメモリが一番近くに配置され，周りに低速な周辺機器が接続される形で構成されている。

計算機の動作をごく簡単に述べると，プロセッサは周辺機器であるディスクから読み込んだプログラムをメモリに書き，そのプログラムを順番に読み込み，実行することで動作している。また，計算対象のデータもディスクに書かれており，それをプロセッサがメモリにコピーし，さらにプロセッサ内部に読み込み，計算し，結果をまたメモリに書き出すことでデータが処理されている。つまり計算機の動作は大部分が「プロセッサ内での演算」または「メモリおよび周辺機器とのデータ交換」のどちらかである。したがってプロセッサの性能を最大限引き出すためには，演算効率とメモリアクセスの効率を上げることが重要となる。次節以降では演算性能に関わる事項として「パイプライン」，メモリ性能に関わる事項として「キャッシュメモリ」および「TLB」について解説する。

2.2 パイプライン

前述のように計算機はプロセッサがメモリに書かれた命令列を読み込み，指示された操作（演算やメモリ入出力）を繰り返すことで動作している。これをもう少し細かく見るとプロセッサは主に以下の手順で命令を実行している。

1. フェッチ（F）
   メモリから命令を読み込む
2. デコード（D）
   読み込まれた命令を解釈する
3. 読み込み（R）
   命令実行に必要な被演算数を読み込む
4. 実行（X）
   命令を実行する
5. 書き込み（W）
   実行結果をレジスタに書き込む

これらのステップはそれぞれ独立した回路で行われるため，ある命令がこれらのステップを完全に完了しなくても，次の命令の実行を始めることができる。したがって，プロセッサは下の図 2 のように複数の命令を同時に処理している。

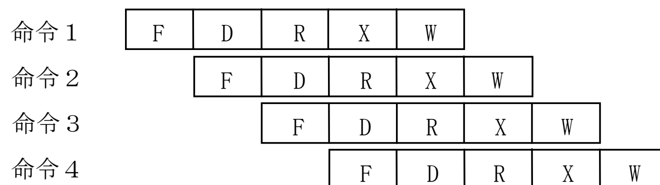


図 2 パイプライン

各命令が完全に独立ならばパイプライン実行は効率的であるが，実際にはプログラムは前の命令の実

行結果を使って次の計算を行うことが多い。このような場合，前の命令の実行が終わるまで次の命令を実行できないため，下図 3 のような待ちが発生する。このような実行待ちを「パイプラインハザード」と呼ぶ。

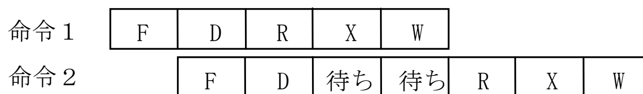


図 3　パイプラインハザード

パイプラインハザードは一般的に以下の場合に生じる。

1. 演算の被演算数（オペランド）に依存関係がある場合
   前の命令の結果がないと次の命令が実行できない
2. 条件分岐の場合
   前の命令が終わらないと，次にどの命令をフェッチするのかがわからない
3. 資源に競合がある場合
   例えば，メモリアクセス命令を連続して発行しても，メモリ操作には時間がかかるため待ちが発生してしまう

ここでは解説のため命令実行は 5 ステップとしたが，近年実際に使用されているプロセッサはこれよりはるかに複雑であり，20ステージ近くまで細かく分割しているプロセッサもある。このようにパイプラインの深いプロセッサではハザードが起きたときの性能ロスも大きくなる。このようなロスを避けるため，プロセッサには依存関係のない命令を探し出して順番を入れ替えて実行するなどの仕組みが実装されている。しかし効率的な実行のためにはソフトウェアの最適化も必要不可欠である。

スパコンプログラミングのためのヒント

プロセッサ性能を極限まで引き出すにはプロセッサメーカーが提供する最適化マニュアルなどを参照しながらアセンブリレベルでプログラムをチューニングする。しかしこれには専門的知識と長い時間を要するため，多くのスパコンユーザーにとって現実的な作業ではない。したがって，最適化はコンパイラとチューニングされた専用ライブラリに任せ，自ら記述する部分に関してはごく基本的なことのみ注意するのが現実的である。具体的な留意点としては以下の3点が挙げられる。

- 条件分岐やジャンプは高コストであることを意識する
  ループの最内周など，実行回数が非常に多い部分に細かい条件分岐や短い関数の呼び出しを入れるのはなるだけ避ける
- 基本的な計算は最適化されたライブラリに任せる
  後述のキャッシュ効率の観点からも重要である
- コンパイルの際には適切な最適化オプションを付けることを忘れない
  多くのシステムではデフォルトでは最適化オプションは付いていない

2.3 キャッシュメモリ

メモリに代表される記憶装置には，性能と容量にトレードオフの関係がある。つまり，大容量の記憶装置は低速で，高速な装置は容量が小さくなる。一方，一般的にアプリケーションプログラムのメモリアクセスには局所性があり，一度アクセスされたメモリは近い将来再びアクセスされるか付近をアクセスされ

る可能性が高いとされている。キャッシュメモリはこの性質を利用して，一度アクセスされたメモリ領域（通常 64 バイト程度のブロック単位で管理される）を高速メモリに格納することにより，続くアクセスを高速化する技術である。計算機の記憶装置は以下のような階層を持つ。

1．L1 キャッシュメモリ
　2,3 クロックでアクセスできる超高速メモリであり，プロセッサ内部に 64KB 程度存在する。
2．L2 キャッシュメモリ
　L1 よりも低速で 10 クロック程度の遅延があるが，プロセッサ内部に 512KB 程度存在する。
3．L3 キャッシュメモリ
　プロセッサの種類によって，存在しないもの，プロセッサ内部に持つもの，外部に持つもの様々であり，存在する場合も 1MB から 32MB 程度と幅がある。50 クロック程度の遅延のものが一般的だが，状況によって異なるものもあり，プロセッサによるばらつきも大きい。
4．メインメモリ
　プロセッサあたり数 GB の容量を持つ主記憶である。数百から千クロック程度のレイテンシがある。
5．スワップ
　メインメモリが足りないときにハードディスクなどにデータを書き出す。これはプロセッサの機能ではなくオペレーティングシステムの機能であるが，あまりにも低速でスーパーコンピュータでの使用には耐えないので，スーパーコンピュータではこの機能は OFF にされることが多い。

このように，どのメモリを使用するかによって性能は大きく異なる。効率的なプログラム実行のためには，なるだけキャッシュ上のメモリを使って計算を行うことが重要である。

スパコンプログラミングのためのヒント

　キャッシュメモリに格納するデータの取捨選択やキャッシュメモリとメインメモリの一貫性の維持など，キャッシュメモリの管理はすべてハードウェアによって制御されている。そのため，プログラム上はキャッシュの存在を無視しても動作の正しさには影響しない。これは便利な性質ではあるが，プログラムによる制御ができないため最適化にはキャッシュの振る舞いを予測しながらの高度なプログラミングが求められることにもなる。キャッシュを効率的に使用する手法には様々なものがあり，本稿が扱う範囲を超えてしまうので本特集号の別記事を参照いただきたい。ただし，少なくとも以下の 2 点は意識してプログラミングを行うべきである。

- メモリはなるだけ連続アクセスする
　ランダムにアクセスすると，キャッシュされていないメモリをアクセスする可能性が高いので性能が低下する。計算アルゴリズムの工夫が重要であるが，それ以前の問題として有名なミスは多次元配列のアクセス順序を間違えることである。特に FORTRAN と C 言語では多次元配列のメモリの配置が異なるので注意が必要である。またプログラム中のジャンプ命令（特に関数ポインタを用いたジャンプ）は，パイプラインの効率にも影響を与えるばかりでなく，命令メモリのキャッシュミスにつながることがあるという点でも注意が必要である。
- 基本的な計算はなるだけライブラリに任せる
　数値演算ライブラリはキャッシュ効率も考慮した上で最高の性能で計算できるよう最適化されている。これらは高級言語で記述した数学的に等価なプログラムよりも高速であることが多い。

2.4 TLB

アプリケーションプログラムがメモリアクセスの際に使用するメモリアドレスは，アプリケーション固有のメモリ空間におけるアドレス（仮想アドレス）であり，物理的なメモリ内の位置を示すアドレス（物理アドレス）とは異なる。この仮想アドレスと物理アドレスの対応表はオペレーティングシステムがメインメモリ上に作成する。アプリケーションがメモリアクセス命令を発行した際には，プロセッサが自動的にその変換表を参照して物理アドレスに変換し実際のメモリにアクセスする。このアドレス変換は「ページ」と呼ばれる一定量の連続領域を単位にして行われる。例えば x86 系のアーキテクチャでは通常，ページサイズは 4KB である。キャッシュの項でも述べたようにメインメモリへのアクセスには時間がかかるため，メインメモリ上の変換表をメモリアクセスの度に参照するのは非効率である。そこでこの変換表にも TLB と呼ばれるキャッシュに似た機構が準備されている。メモリアクセスの際にアドレスが TLB に存在していれば遅延の短いアクセスが可能となる。

スパコンプログラミングのためのヒント

キャッシュとは異なり TLB は1エントリで 4KB もの領域をカバーできるため，普通にプログラムを書けば TLB はほぼヒットすると考えてよい。ただしあまりにもアドレスの離れたランダムアクセスや，多次元配列のアクセス順序の間違いを犯すとTLBミスが多発する可能性がある。また，プログラム開始直後はすべてのページで TLB ミスが起きる。ただしプログラム開始直後に関しては TLB ミスのペナルティーよりもオペレーティングシステムのデマンドページングの処理に要する時間の方が問題であるため，詳細はオペレーティングシステムの章で述べる。

3. スーパーコンピュータの構成方式

並列計算機は多数のプロセッサを搭載した計算機であり，近年のスーパーコンピュータは例外なく並列計算機である。並列計算機はプロセッサを接続する方式の違いにより「共有メモリ型並列計算機」と「分散メモリ型並列計算機」に分類される。本章では両方式を紹介した上で，近年のスーパーコンピュータの構成方式について述べる。

3.1 共有メモリ型並列計算機

共有メモリ型並列計算機とは，その名の通りすべてのプロセッサが同一のメモリを共有する並列計算機である。この構成の並列計算機には主に SMP と ccNUMA が存在する。

3.1.1 SMP 方式

SMP は Symmetric Multi Processing の頭文字である。これはすべてのプロセッサが対等な並列計算機であり，理想的な SMP ではどのプロセッサからどのメモリをアクセスしても同じ性能であり，周辺機器との通信もすべてのプロセッサが均等に処理を行う。図 4 に SMP 構成の概略を示す。

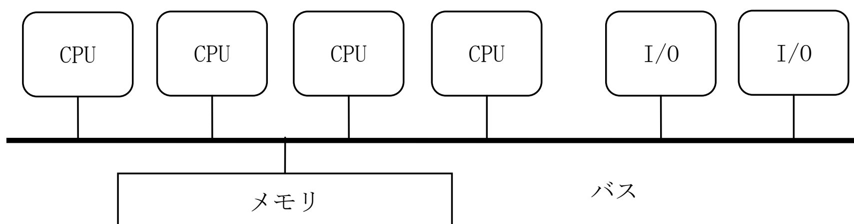


図 4 SMP

すべてのプロセッサや I/O(周辺機器)は1本のバスに接続されており，メモリアクセスはこのバスを通じて行われる。すべてのプロセッサとI/O が1本のバスにつながっているため，どのプロセッサからどのメモリをアクセスしても性能は均一であり，すべてのプロセッサがI/O に対しても対等の関係にある。

キャッシュメモリの存在は共有メモリ型並列計算機にとっては大きな問題である。各プロセッサは個別にキャッシュメモリを持つため，複数のプロセッサが同じ領域をキャッシュする場合があり，これらが矛盾しないようにしなくてはならない(キャッシュの一貫性保持)。SMP 方式の場合，各プロセッサはバスに流れる制御信号を常に監視し，他のプロセッサや I/O によって自身がキャッシュしているメモリ領域が更新された際にはキャッシュの内容を破棄する処理を行っている。

SMP 方式はメモリの物理的な場所をソフトウェアが意識する必要がないため扱いやすい。しかし，バス性能の限界やキャッシュの一貫性保持のためのコストが高く，大規模な SMP 型計算機を作るのは困難である。

3.1.2 ccNUMA 方式

NUMA とは Non-Uniform Memory Access であり，cc は cache coherent を意味する。つまりプロセッサとメモリの位置関係によって性能や振る舞いが異なるが，キャッシュの一貫性は保たれている共有メモリ型並列計算機である。図 5 に ccNUMA 方式の概略図を示す。

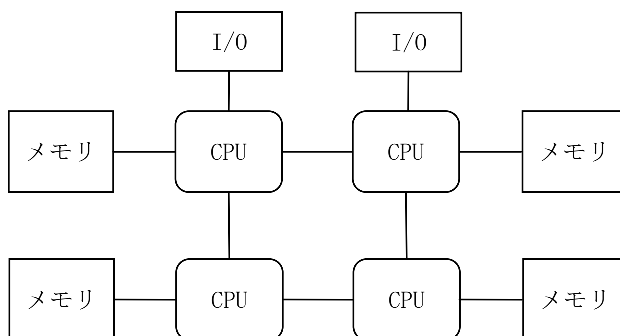


図 5 NUMA

図のような構成の場合，直結されているメモリには高速にアクセスできるが，他の CPU に接続されているメモリにアクセスする場合は遅延が大きくなる。また，周辺機器から見たメモリ性能も均一ではない。ソフトウェアの立場から見るとccNUMA構成の並列計算機はSMPよりは扱いづらい。しかしハードウェア作成における制限が SMP よりは緩和されるため，大規模な計算機の作成が可能であり，最適化されたプログラムでは SMP よりも高性能であることが多い。

ccNUMA は機能的には SMP と同じである。つまり，すべてのプロセッサからすべてのメモリのアクセスが可能でありキャッシュの一貫性も保持される。したがって基本的にはSMP 用のソフトウェアがそのまま使用できる。ただし，最高の性能を得るためにはなるだけ近くのメモリにアクセスすることが必要である。近年では多くのオペレーティングシステムが NUMA の特性を考慮したメモリ配置を行っている。

3.2 分散メモリ型並列計算機

分散メモリ型並列計算機は複数の計算機をネットワークで接続した並列計算機である。構成要素となる個々の計算機は「ノード」と呼ばれ，非常に限定された機能のみを持つ小型計算機である場合や，前述したような共有メモリ型の並列計算機である場合などがある。また，ノード間のネットワークはインタ

ーネットなどで一般的に使用されているネットワークとは違い，高速な専用ネットワークであることが多い。図 6 に分散メモリ型並列計算機の概略を示す。

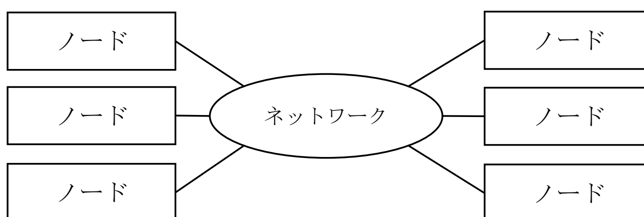


図 6　分散メモリ型並列計算機

分散メモリ型並列計算機はそれぞれのノードのメモリが独立しているため，プロセッサから他のノードのメモリを直接アクセスすることはできない。したがってノードにまたがる協調動作のためにはアプリケーションプログラムが明示的な通信を行う必要があり，共有メモリ型並列計算機のプログラムをそのまま使用することはできない。このように分散メモリ型並列計算機はアプリケーションに通信を記述しなくてはならないという欠点を持つが，共有メモリ型よりも安価であり，また共有メモリ型では不可能な超大型計算機が作成可能であるという利点を持つ。

3.3 近年の並列計算機

共有メモリ型並列計算機はメモリアクセスの度にキャッシュの一貫性保持のための通信が必要である。プロセッサ数が増加するにつれて通信量も増加するため，共有メモリ型並列計算機では規模に比例した性能向上は見込めない。そのため大規模化には限界があり，実際，市場に存在する最大の共有メモリ型並列計算機は，SMP で 100 プロセッサ，ccNUMA で 1000 プロセッサ程度である。このような共有メモリ型の限界のため，現在の大型計算機の主流は分散メモリ型である。

一方，分散メモリ型並列計算機を構成するノードに目を向けると，プロセッサのマルチコア化の流れを受けて，ノードが小規模な共有メモリ型並列計算機を構成していることが多くなってきている。現在のスーパーコンピュータでは，マルチコアのプロセッサを数個搭載した 4 から16プロセッサ程度の共有メモリ型並列計算機をノードとして使用するのが主流である。

スパコンプログラミングのためのヒント

現在の主流である分散メモリ型並列計算機を使用するためには，必要な通信をアプリケーション内に明示的に記述する必要がある。この通信の記述には MPI と呼ばれる通信ライブラリを使用するのが一般的である。MPI でアプリケーションを記述すれば，並列計算機が使用するネットワークに依存しないインタフェースで通信が記述できるため，構成の異なる並列計算機に移行する際も，再コンパイルするのみで動く可能性が高い。また，MPI は分散メモリ型並列計算機向けの通信規格であるが，共有メモリ型並列計算機でもメモリを介して通信を行うことで MPI を使用できる。

一方，各ノードは共有メモリ型並列計算機であるので，ノード内では共有メモリを使用した並列化が可能である。ノード内での並列化には自動並列化または OpenMP が使用されることが多い。自動並列化はコンパイラが自動的に依存関係のない複数の計算を検知して，それぞれの計算を個別のプロセッサに割り付ける方法である。プログラマの負担は一切ないため最も簡単な方法であるが，コンパイラによる並列性の自動検出には限界があり，性能がよくないことがある。OpenMP は並列化して計算できる部分をユーザーが明示的に指定するための規格である。並列化可能なループに OpenMP 指示文を

追加すると，コンパイラはそれにしたがって並列化したコードを生成する。自動並列化や OpenMP は共有メモリ型並列計算機向けの仕組みであり，原則として分散メモリ環境では使用できない。

以上をまとめると，並列アプリケーションの作成方法は以下の2通りとなる。

1．ノード内を自動並列化または OpenMP，ノード間を MPI で並列化する
   自動並列化の場合は MPI プログラムを自動並列化オプション付きでコンパイルするのみである。OpenMP の場合は MPI による通信を書いてはならない場所があるので注意が必要である。
2．全体を MPI で並列化する
   共有メモリ型並列計算機内でも MPI は使用できるので，全体を MPI で並列化してもよい。

残念ながら，いずれにしても MPI による通信の記述は避けられないのが現実である。したがって新たにアプリケーションを作成する際は，まず MPI で通信を記述することによる並列化を行うべきである。それをそのまま実行すれば2の方式となり，OpenMP 指示文などを追加してノード内並列化を最適化すれば1の方式となる。一般的に1の方が高速だとされているが，アプリケーションによっては2の方が高速なこともある。

3.4 メモリバンド幅

ここまでスーパーコンピュータの構成方式を述べてきたが，最後にまとめとして，スーパーコンピュータの構成を考える上で非常に重要なメモリバンド幅について考えてみる。

スーパーコンピュータを数値演算に使用する場合，メモリの性能（メモリバンド幅）はアプリケーション性能を左右する重要な要素である。例えば代表的な数値計算ライブラリである BLAS の中には $\vec{y} = \vec{y} + a * \vec{x}$ を行う DAXPY と呼ばれるサブルーチンがある。この演算では2回の浮動小数点演算に対して3回のメモリアクセスが発生する。DAXPY では浮動小数点数は 8 byte なので，一回の浮動小数点演算で発生するメモリアクセスは 3 x 8 / 2 = 12 Byte である。これを 12 Byte/Flop と言う。一方，実際のプロセッサの例として最新の Quad Core Opteron プロセッサを例に挙げると，このプロセッサは1クロックで 4 回の浮動小数点演算ができるため，2.3GHz 動作の Quad Core プロセッサでは 1 秒あたり 2.3G x 4 x 4 = 36.8G 回の計算ができる。このプロセッサと共に一般的に使われるメモリの性能は 10.6GB/s なので 10.6 / 36.8 = 0.28 Byte/Flop ということになる。これは DAXPY で求められる 2.3% でしかない。つまり，キャッシュの効かない巨大なベクトルでこの演算を行った場合，メモリバンド幅が律速となって CPU の理論性能の 2.3%の性能しか得られないこととなる。これは非常に極端な例であり，行列積計算のようにキャッシュの利用効率が高く，理論性能の 90% 近くが得られる計算も存在するが，DAXPY はメモリバンド幅の重要性，プロセッサの周波数やコアの数をのみ求める無意味さを示す好例である。

本章で述べてきた並列計算機の構成方式もメモリバンド幅と深い関係がある。各方式の説明でも触れたが，SMP 方式はキャッシュの一貫性保持のためのコストやバスのバンド幅が律速となりやすくメモリバンド幅を確保するのが困難である。ccNUMA であればバスが律速となることがなくなるため，SMP よりはメモリバンド幅は確保しやすくなるが，キャッシュ一貫性保持のためのコストは依然として残るため，プロセッサ数に比例してメモリバンド幅が増えるわけではない。その点，分散メモリ方式であれば計算機の数を増やせばそれに比例して総合的なメモリバンド幅も増加する。なお本稿では触れないが，高いメモリバンド幅を持つ計算機の代表はベクトル型スーパーコンピュータである。近年では並列化された演算器を作るのは難しいことではないため，ベクトル型の最大の利点はベクトルであることよりもスカラー型を圧倒的に上回るメモリバンド幅であると言える。

4. オペレーティングシステム

本章ではアプリケーションの実行をサポートするオペレーティングシステムについて述べる。オペレーティングシステムとは，アプリケーションに対する資源の割り当てを決定したり，ディスク入出力をサポートするためのソフトウェアであり，近年の計算機ではほぼ例外なく使用される基本ソフトウェアである。共有メモリ型並列計算機では計算機一台につき一個のオペレーティングシステムが，分散メモリ型並列計算機では各ノードで一個のオペレーティングシステムが動作している。

本章では動作中のアプリケーションを表現するための概念であるプロセスとスレッドについて紹介した後，アプリケーションがオペレーティングシステムの機能を使用する際に使用するシステムコールについて述べる。そして最後にメモリ管理の仕組みとして重要な概念であるページングについて紹介する。

4.1 プロセス，スレッド

プロセスとは実行中のプログラムのことである。同一のプログラムであっても複数走らせれば複数プロセスになる。プロセスは独立したメモリ空間を持つため，他のプロセスのメモリを直接アクセスすることはできない。スレッドはプログラム実行の最小単位であり，プロセッサの割り当てはスレッド単位で行われる。一個のプロセスは複数のスレッドを持つことができ，同じプロセスに属すスレッドはすべて同一のメモリ空間で動く。下図 7は同一のプログラムを2個走らせた状況を示しており，それぞれのプロセスが3個のスレッドを持つ。図ではそれぞれのスレッドが別の関数を実行しているが，これは同じであっても構わない。

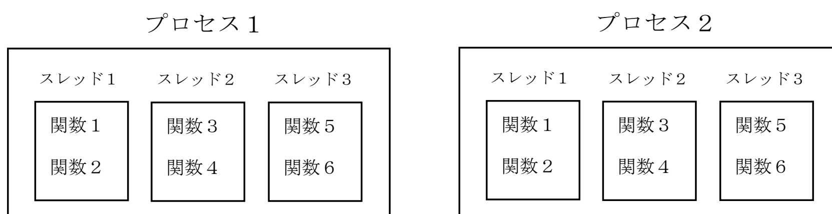


図 7　プロセスとスレッド

共有メモリ型並列計算機上では，複数のスレッドを同時実行することでも複数のプロセスを同時実行することでも並列計算は可能である。ただし，複数プロセスによる並列化の場合は，明示的にデータ共有の仕組みを作らなくてはならない。これはほとんど分散メモリ型並列計算機を使用しているのと同じ状態であり，実際，一台の共有メモリ型並列計算機の中で複数のプロセスを使った並列化を行う際には，MPI で通信を記述するのが一般的である。共有メモリ型並列計算機の利点は，メモリを共有しながら複数のプロセッサを同時に使用できる点であり，この利点を生かすためにはスレッドを用いるのが最適である。前述した OpenMP による並列化を行った場合，または自動並列化を使用した場合は自動的にスレッドが使用される。

スパコンプログラミングのためのヒント

スレッドとプロセスの使い分けに関しては，メモリ共有の有無の他，スレッドはプロセスよりも生成，消滅に関わるオーバヘッドが低いというような性質も考慮して適切な方を選択する。これはサーバアプリケーションなどを作成するときには重要なポイントであるが，並列計算にあいては，MPI を使用すればプロセス，OpenMP や自動並列化を使えばスレッドが自動的に選択される。このためスパコンユーザー

がこれらの性質を考慮する必要はない。

スパコンユーザーが注意するべき点は，スレッドを使った際の資源競合である。スレッドの場合はすべてのスレッドがメモリ空間を共有するため，グローバル変数や動的に確保したメモリ領域は複数のスレッドから同時にアクセスされる可能性がある。複数スレッドが同じメモリ領域を同時にアクセスすると，普通では起こりえない矛盾した状態のデータが書き込まれる危険性がある。例えばグローバル変数の値に1を加える処理を2スレッドが同時に行う場合を考えると，それぞれのスレッドが同じ値をメモリから読み込み，1を加算してメモリに書き戻すため，結果として1しか加算されない（同時実行でなければ2増加したはずである）。OpenMP による並列化を行う場合，このような矛盾した状況を引き起こさずに並列化できるかどうかを判断するのはユーザーの責任である。例えばグローバル変数への書き込みがあるようなループを並列化する際には安全性の検討を十分に行う必要がある。

4.2 システムコール

プロセスやスレッドの生成やディスク入出力など，アプリケーションがオペレーティングシステムの機能を使用する場合はシステムコールと呼ばれる方法を使ってオペレーティングシステムに処理の要求を出す。システムコールはソフトウェアによって明示的に発生させる例外であり，例外が発生するとプロセッサは特権モードと呼ばれる権限の高いモードに移行した上で，オペレーティングシステムの決められた部分（例外ハンドラ）へと処理を移行する。この例外ハンドラにおいてオペレーティングシステムはユーザープログラムの要求を受け取り，必要な処理を行った上で例外ハンドラの終了をプロセッサに伝える。例外ハンドラが終了するとプロセッサは権限をユーザープログラムの権限に戻した上で，例外を発生させた次の命令より実行を再開する。

スパコンにおけるプログラムはほとんどが数値計算であるため，このようなシステムコールが発行される頻度は低い。しかし，ファイル入出力など，一部のシステムコールはスパコンでも頻繁に使用されるので，システムコールの正しい使用方法を知ることは重要である。

スパコンプログラミングのためのヒント

C 言語から使用する場合，システムコールは通常のライブラリ関数同じように使用できる。例えばファイルにデータを書き出すシステムコールは write 関数であり，この関数は特別な準備をすることなく C 言語のプログラム中で使用することができる。しかし，見た目は単なるライブラリ関数であっても，前述のようにシステムコールは例外を発生させる。例外が発生するとプロセッサはプログラム実行の強制中断に伴う状態の保存や特権モードの切り替えなど，様々な作業を行わなければならない。したがってこれには単なる関数呼び出しとは比較にならないほどの長い時間を要する。

一般にスパコンプログラムの中でシステムコールを直接呼び出すことはなるだけ避けた方がよい。例えばファイル入出力の場合，システムコールを効率的に使用するライブラリ群が準備されているので（通常使用する fread，fwrite がそのライブラリである）そちらを使う方がよい。特殊なファイルシステムなどでライブラリ関数が効率的に動かない場合はシステムコールを直接使用することになるが，その場合はシステムコールのコストやファイルシステムの特性などを十分理解した上でプログラムを作成する必要がある。

4.3 ページング

プロセッサアーキテクチャの節で述べたように，アプリケーションプログラムがメモリアクセスの際に使用するアドレスは仮想アドレスであり，物理的なメモリの位置を示すアドレス（物理アドレス）とは異なる。この仮想アドレスと物理アドレスの変換は4KB程度の「ページ」と呼ばれる単位で行われており，この変換表を「ページテーブル」と呼ぶ。ページテーブルの作成はオペレーティングシステムの役割である。

実は，C 言語のグローバル変数や FORTRAN の配列はプログラム開始時には物理アドレスが割り当てられていない。また C 言語の malloc のように動的に割り当てられるメモリ領域も，malloc 関数が終了した時点ではまだ物理的なメモリは割り当てられていない。これらは実際にアクセスされて初めて物理的なメモリが割り当てられる。このメモリ割り当ての仕組みについて簡単に解説する。

プロセスの項で解説したように，プロセス生成時にはそのプロセス固有のメモリ空間が割り当てられる。これは仮想アドレスの空間であり，この仮想アドレスに対応する物理アドレスは未知のまま実行が始まる。つまりこの段階ではこのプロセス用のページテーブルは空の状態である。プログラムの実行が始まりメモリへのアクセスが発生すると，プロセッサはそのメモリアドレスを物理アドレスに変換しようとページテーブルを参照する。しかし，ページテーブルは空であるため，アドレス変換は失敗する。するとこれは例外となり，プログラム実行は一時停止されオペレーティングシステムへと処理が移る（システムコールと似たような動作になる）。ここで初めてオペレーティングシステムが空いている物理メモリ領域を割り当て，ページテーブルに物理アドレスをセットする。この例外処理が終了すると，実行はアプリケーションプログラムに戻される。システムコールと異なり，メモリ関連の例外はハンドラから戻った直後に再実行されることになっているため，先ほど失敗したメモリアクセス命令は再び実行される。今度はプロセッサが正しいページテーブルを見つけることができるため，このメモリアクセスは成功する。malloc による動的割り当ての場合も動作は同様である。malloc は有効な仮想アドレスの割り当てをオペレーティングシステムに依頼しているのみであり，実際のメモリ割り当てを行っているわけではない。C の入門書などでは malloc に失敗するのはメモリ不足の場合であると解説されているが，これは仮想アドレス空間の不足を意味しており，近年の 64bit アーキテクチャで仮想アドレス空間の不足が発生する可能性はほぼゼロである。したがって，実際に物理的なメモリ不足に陥った場合は malloc が NULL で終了するのではなく，実行時エラーによる強制終了という形でユーザーに報告される。

このように，実際にアクセスされて初めて物理メモリを割り当てる方式を「デマンドページング」と呼ぶ。この方式は実際に使用されない領域にメモリを割り当ててしまう無駄を避けるため，今日ではほとんどのオペレーティングシステムに実装されている。

スパコンプログラミングのためのヒント

デマンドページングはその存在を知らずにプログラミングをしても動作の正しさには影響しない。気をつけるべき点は，本当のメモリ不足は malloc の返値が NULL でないことを見るだけでは判別できないことのみである。ただ，性能評価の際にはデマンドページングを意識する必要がある。特にプログラム開始直後はすべてのページにおいて例外が発生していることに気をつける必要がある。例えばループの一周目にかかる時間を測定して全体の性能を予測したとしてもおそらく正しい値は得られない。ループの一周目ではデマンドページングの処理が行われているため，二周目以降は大幅に高速化されるからである（もちろん TLB やキャッシュの影響もある）。また，性能測定の際にも注意が必要であり，性能測定のために取得した時刻をメモリに書く際に例外を起こすと，その分のペナルティーが計測結果に含まれてしまう。

5. 通信

本章では分散メモリ型並列計算機の重要な構成要素である通信機構について述べる。並列計算機における通信は通常はインターコネクトと呼ばれる専用ネットワークが使用される。本章の最初はこのインターコネクトについて特徴と性能を紹介する。続いて通信の仕組みについて述べた後，最後に通信性能がアプリケーション全体の性能に及ぼす影響について簡単なデータを紹介する。

5.1 インターコネクト

分散メモリ型並列計算機のノード間接続には様々な高速ネットワークが使用されている。これらは広義にはネットワークであるが，短距離を高バンド幅で接続する意味で「インターコネクト」と呼ぶ場合が多い。並列計算機メーカーはそれぞれ固有のインターコネクトを開発しており，10GB/s 以上のバンド幅を持つ非常に高速なものもある。一方で特定の並列計算機専用ではなく，一般的な PC アーキテクチャの計算機で使用できるインターコネクトも開発されており，Infiniband または Myrinet が広く使われている。これらは単体でそれぞれ 2.0GB/s，1.25GB/s であるが，複数リンクを同時に使用することで，5GB/s 程度の性能は実現可能である。さらに，現在発売されているほとんどの計算機に標準で搭載されている Ethernet もインターコネクトとして使用することがある。ただし，PC サーバに内蔵されているような一般的な Ethernet をインターコネクトとして使用する場合は，プロセッサ負荷，遅延，大規模ネットワークでのバンド幅の点で専用インターコネクトに劣る点が多い。したがって Ethernet をインターコネクトとして使用するのは主にコスト的メリットを求める場合である。

5.2 通信の基本性能

通信性能を示すために最もよく用いられるのは通信バンド幅である。バンド幅は bps(bits per second) または MB/s などで表され，1 秒間に転送できるデータ量を意味する。ここで注意が必要なのは，カタログスペックとして示されるバンド幅は設計上の限界値である点であり，実際に得られるバンド幅は転送するデータのサイズに依存する。

実際の通信性能の傾向を示すために，例として Myrinet-10G と呼ばれるインターコネクトを用いた並列計算機で，MPI を使用した通信性能測定プログラムを実行した結果を図 8 に示す。結果は横軸に一度に転送したデータのサイズ，縦軸にバンド幅が示されている。

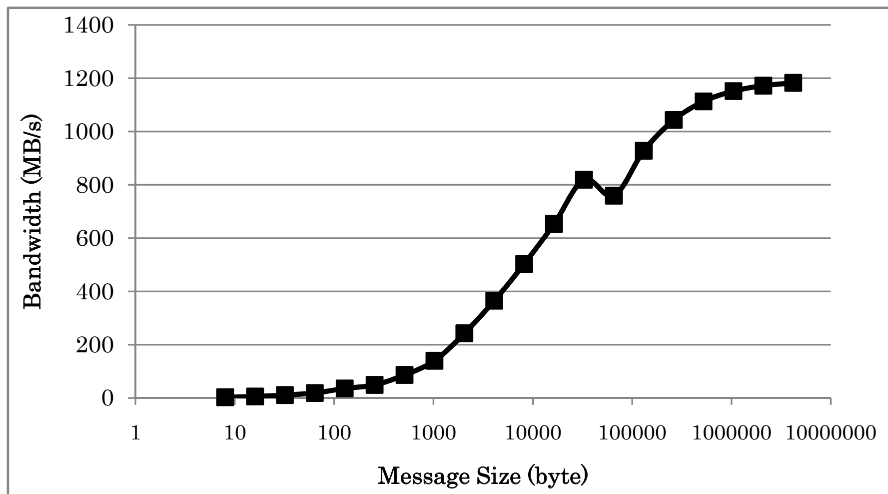


図 8 基本通信性能

測定に用いた Myrinet-10G は理論性能が 10Gbps のインターコネクトであるため，最高性能が 1250MB/s である。グラフの一番右の点はメッセージサイズが 4MB の場合で，そのときの性能は 1188MB/s である。これは理論性能の 95%程度であり，実際に得られる性能としては妥当な値である。

グラフからわかるように，理論性能に近い性能を得るためには 4MB 程度のメッセージサイズが必要であり，小さいメッセージではネットワークの性能を十分に生かすことはできない。これは通信に必要な準備作業の影響であり，小さなメッセージではこの準備作業に要する時間が全体通信時間の大部分を占めるため，ネットワークの物理性能が生かせない。したがって，高い通信性能のためには通信準備にかかる時間が短いことも重要である。上記の測定で用いた Myrinet-10G の場合，4byte のメッセージを送信するのに要する時間は 2.8 マイクロ秒程度である。同じ計算機に搭載されている 1Gbps の Ethernet を使用した場合は 50 マイクロ秒程度の時間がかかるので，インターコネクトとして設計された Myrinet-10G は非常に高性能であることがわかる。

スパコンプログラミングのためのヒント

通信は，なるだけまとめて行うようにする方が効率的である。また，MPI のタイプを使用するときも注意が必要である。MPI で規定されているユーザー定義型を使用すると，非連続領域を一回の通信操作で送受信できる。これはプログラムの見た目はシンプルであるが，実際には細かい非連続領域の集まりなので，小さなメッセージの大量送信になる可能性がある。MPI の実装によっては一度大きな内部バッファにデータを集める可能性もあるが，いずれにしても連続領域を通信するよりは時間のかかる作業となるので，MPI プログラム上で 1 行であることは必ずしも高速に通信できることにはつながらない。

5.4 通信の仕組み

前述したようにインターコネクトには様々なものがあり，種類によって通信の仕組みは異なる。また，メーカー独自のものは動作原理が公開されていないことも多い。したがってここでは PC アーキテクチャで使用されているインターコネクトを例に通信の仕組みを述べる。具体的なインターコネクトとしては Infiniband または Myrinet が該当する。本稿で述べる範囲においては両者に大きな差はない。

5.4.1 DMA 転送と見えないバッファ

周辺機器はプロセッサの関与しないところでメモリと直接データのやりとりができる。これを DMA(Direct Memory Access)と呼ぶ。ネットワークによる通信はネットワークインタフェースカードとメモリの間で DMA 転送をすることにより，プロセッサへの負荷を最小限に抑えながら通信を行う。

DMA 転送を行う際にはネットワークカードからメモリに対して読み込みまたは書き込み要求を発行する。周辺機器にはプロセッサのページテーブルに相当するものは存在しないため DMA 転送は物理アドレスを用いて行われる（厳密には「バスアドレス」であるがここでは物理アドレスと同一視する）。一方，送受信対象となるデータは各プロセスのメモリ空間に存在するため，DMA 転送のためには転送対象のデータが置かれているメモリ領域の仮想アドレスを物理アドレスに変換した上でネットワークカードに知らせる必要がある。オペレーティングシステムの章で述べたように，仮想アドレスと物理アドレスの変換表（ページテーブル）はオペレーティングシステムが作成しているため，オペレーティングシステムに問い合わせればこのアドレス変換は可能である。しかしこうしてアドレス変換を行うだけでは実は危険な場合がある。それは DMA 転送中にオペレーティングシステムが実行プロセスの切り替えを行い，別のプロセスにそのメモリ領域を割り当ててしまった場合である。このような場合でもネットワークカードは

知らずに DMA 転送を続けるため，最悪の場合はネットワークを使用しているプロセスとは関係のないプロセスのデータを送信してしまうことがあり，正しい計算が行われないばかりか無関係なプロセスのデータ破壊や情報漏洩の危険も生じる。

このような危険な状況を避けるため，通信ライブラリは転送に使用するメモリ領域をあらかじめ一定量を確保した上で，オペレーティングシステムに物理アドレスと仮想アドレスの対応を決して変更しないよう依頼している。これをメモリレジストレーションと呼ぶ（ピンダウン，メモリロックなど他の呼び方もある）。結局，DMA 転送はハードウェア的には任意のメモリアドレスに対して実行可能であるものの，実質的にはメモリレジストレーションが完了している領域に対してしか使用できない。

このような制約のため，送受信はメモリレジストレーションが完了しているユーザーには見えないバッファを介して行われている。具体的には，送信の際には通信ライブラリが送信対象のデータをまずメモリレジストレーションされた領域にコピーした上でネットワークカードに送信依頼を出している。また受信データはネットワークカードがメモリレジストレーションされた領域に DMA を行った後，通信ライブラリがその領域からユーザーが希望する受信バッファにコピーしている。

スパコンプログラミングのためのヒント

ユーザーが気をつけるべき点は，通信ライブラリの中にメモリレジストレーション用の見えないバッファ領域が存在している点である。高い通信性能のためにはメモリレジストレーションをするバッファは無視できない程度のサイズが必要であり，並列計算機の規模とネットワークの種類によってはこのバッファだけで 1GB 近くのメモリを使用することがある。したがって，アプリケーションが計算用に使用できるメモリ量は，各ノードの物理メモリ量からオペレーティングシステムなどのシステム領域を減じ，さらに通信バッファの量を減じた残りの部分である。具体的な数字はシステム管理者から目安が示されることが多い。

5.4.1 ゼロコピー通信

前述のように DMA 転送はメモリレジストレーションされた領域に対してしか実行できないため，送受信データは一度送受信用のバッファにコピーされる。ところがユーザーが非常に大きな領域に対して送受信命令を発行した場合は，あらかじめレジストレーションされた領域にコピーをするよりも，ユーザーから指定されたバッファを新たにレジストレーションし，そこから DMA 転送を行った方が高速な場合がある。このような通信をゼロコピー通信と呼ぶ（送受信バッファへのコピーがないのでゼロコピーと呼ばれる）。メモリレジストレーションはコストのかかる操作であるため，コピーをなくすメリットがメモリレジストレーションのコストを上回らなければならない。ゼロコピー通信の方が高速になるメッセージサイズは計算機によって異なるが，現在の計算機では 32KB 程度が境目になることが多い。

実は，通信ライブラリはメッセージサイズによって前節で解説したコピーを伴う通信とゼロコピー通信を使い分けている。通信性能の章で示した性能のグラフ（図 8）で，メッセージサイズを増やしているにもかかわらず性能が下がっている点があるが，これはこの通信方法の切り替えのためである。図 8 の場合は，ゼロコピー通信のメリットが出始めると判断したサイズが小さすぎたため，性能が落ち込む点が出てしまっている。適切に管理されているスーパーコンピュータの場合は，このような切り替え点は最適な点に設定されているため図 8 のような性能カーブにはならず，単調増加になるはずである。

スパコンプログラミングのためのヒント

一度に通信する量が多いほどネットワークを効率的に扱えるのは前述した通りであるが，ゼロコピー通信の方が高速になる程度のサイズで通信を行うことはプロセッサの有効利用にもつながる。メモリレジストレーションされた領域へのコピーを伴う通信では，そのコピー作業にプロセッサの資源を使用してしまうため，その間はユーザーが本来求めている計算はできない。一方，ゼロコピー通信の場合は一度送信命令をネットワークカードに出してしまえば，あとはプロセッサを自由に使用することができる。つまり通信の完了を待たずに他の処理を続けることができるため，通信に要する時間を隠蔽してプロセッサを常に有効な計算に割り当てることができる。このような通信と計算のオーバラップのためには MPI_ISEND などの非同期送受信関数を使用し，完了を待つ前に他の処理を記述する。ただし，この間は通信対象のメモリ領域を使用することはできないので注意が必要である。

5.5 通信がアプリケーション性能に与える影響

通信の話の最後に並列計算における通信性能の重要性を示す簡単なデータを紹介する。下のグラフは並列計算機のベンチマークとして広く使われている NAS Parallel Benchmarks の中の CG を PC クラスタで実行した結果である。横軸は使用したプロセッサ数，縦軸が性能である。黒いグラフが 1Gbps の Ethernet を使用した場合であり，灰色が Myrinet 10G を用いた場合の結果である。まったく同じマシン(Dual Core Opteron 2GHz，DDR2 667MHz) で，使用するネットワークのみを変えて測定している。マシンが同じであるにも関わらず 32 プロセッサでの性能が大きく異なることがわかる。一般に並列計算ではプロセッサ数が増加するにしたがって通信量が多くなるため，ネットワークに求められる性能も高くなる。そしてネットワーク性能が追いつかない程度までプロセッサ数を増やしてしまうと全体的な性能は落ちてしまう。上の例は 16 プロセッサまでは Gigabit Ethernet でも十分に扱える通信量であるが，32 プロセッサの場合は 10Gbps のインターコネクトでないと実用的な性能が出ない通信量になっていることを示している。

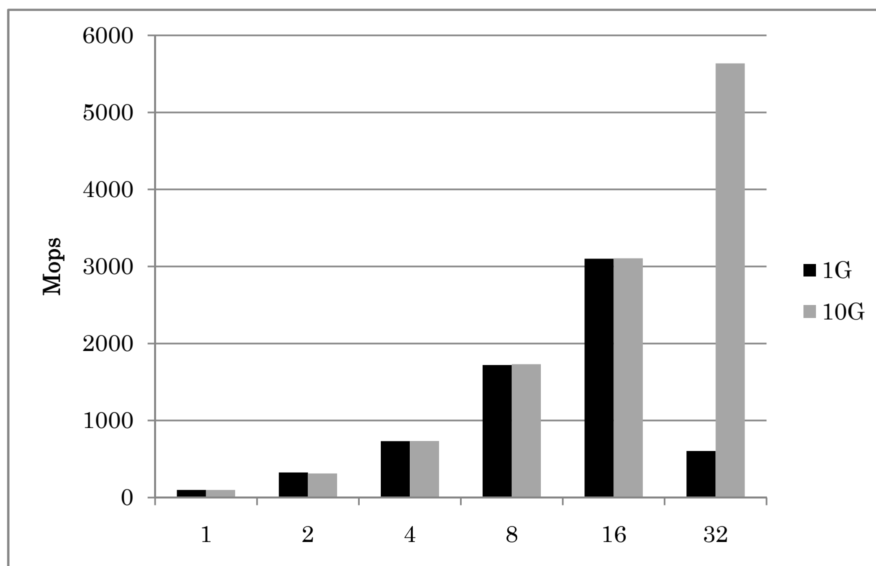


図 9　NAS Parallel Benchmarks CG (class C) の結果

スパコンプログラミングのためのヒント

並列計算機の規模に見合ったネットワーク性能を備えることは並列計算機の設計者の責任である。ただし，アプリケーション側でも通信をなるだけ少なくする工夫を行わないと，少ないプロセッサ数で性能限界に達してしまう。そして限界を超えてプロセッサ数を増やした場合は性能が上がらないのみならず，前ページのグラフのように大きく低下してしまうこともある。性能限界点は計算時間と通信時間の比で決まる。大規模化のためには，計算時間を長く，通信時間を短くしなくてはならない。一般的に，なるだけ計算時間を長くするためには各プロセスに大きな問題を割り当てるのがよい。このため，アプリケーション作成時には各ノードが担当する計算量を実行時のパラメーターとして与えられるようにしておき，何回かのテストランでそのパラメーターを調整することでノードが持つメモリを使い切るようにすべきである。

6. 参考文献

本稿ではプロセッサアーキテクチャ，オペレーティングシステム，インターコネクトについてアーキテクチャの基礎を紹介した。それぞれについて参考文献を一冊ずつ紹介する。

プロセッサアーキテクチャに関する参考書として有名なのは次の書籍である。

David A. Patterson , John L. Hennessy

"Computer Architecture Fourth Edition: A Quantitative Approach"

ただし本書はプロセッサアーキテクチャの複雑化に連動するように版を重ねるごとに内容が高度化している。アーキテクチャの基礎を学ぶには second edition 程度が適切かもしれない。

オペレーティングシステムの参考書としては以下のものがよい。

Abraham Silberschatz, Peter Baer Galvin, Greg Gagne

"Operating System Concepts 7th edition"

通信に関しては通信の専門書よりも並列計算機のアーキテクチャ全般を解説した本の方が適切である。通信の専門書は長距離ネットワークや IP ルーティングなど，インターネット技術に関連するものが多く，並列計算機の通信に特化して解説した本を探すのは困難である。並列計算機の解説書としてはやや古いが次のものがよい。

David E. Culler, Jaswinder Pal Singh, Anoop Gupta

"Parallel Computer Architecture: A Hardware/Software Approach"

本書では並列計算機の構成方式や通信，並列アプリケーションに至るまで並列計算機全般について幅広く解説されている。

7. おわりに

本稿ではスパコンプログラミングのヒントとなるよう，計算機システム全般を広く解説した。プロセッサの高速化に伴って Java やスクリプト言語のような，性能よりも再利用性や移植性を重視した抽象的なプログラミングが主流となりつつある近年だが，スーパーコンピュータは高い性能で計算してこそ意義のある計算機であり，依然として FORTRAN, C, C++ を中心に使用されている。そして，このようなハードウェアに近い言語でプログラミングをするにあたっては，高速化のため，あるいは致命的な性能低下につながるミスを避けるためにアーキテクチャの知識は重要である。特にこれからスーパーコンピュータを使い始める若い読者の方には，本稿で紹介したような並列計算機のアーキテクチャに興味を持っていただき，高速なプログラムを作成するための知識として生かしていただきたいと思う。

キャッシュ性能安定性について

電気通信大学　今村 俊幸

1　準備

本記事の読者は現在殆どのマイクロプロセッサでほぼ同一の方式のキャッシュが搭載されていることをご存知であろうか? 「ハードウェア」や「プロセッサ」と名の付いた多くの書籍に詳細は書かれているので, 本原稿では最小必要限の内容を準備としてから解説を進めていこう.

図 1 は, 一般的なキャッシュの構造を模式化したものである. テキストによっては, 記法が異なる場合があるが, 本記事ではこの方式で説明を進めたい. 多くのプロセッサで採用されているキャッシュの構造は **n-way セットアソシアティブ (n 群連想記憶方式)** キャッシュと呼ばれる. 図 1 は, 4-way 構成した場合のキャッシュの概念図になり, n-way 構成の場合キャッシュ全体は n 個のバンク **(way)** に分けられそれぞれのバンクでは更に**ライン**と呼ばれる単位に分割される. 各ラインには**データ**と**タグ**が対応し, データ部分にはプロセッサがアクセスする主記憶データのコピーが格納される. 更に, タグ部分にはラインの仮想記憶上でのアドレス[1] の上位ビットを記憶する. そして, 列位置 (図では set x と表記) を**インデックス** (x) と呼び, 同一インデックスの n 個のラインの集合を**セット**と呼ぶ.

一般に, ライン数やラインサイズは **2** のベキであり, 本稿ではそれぞれ 2^s, 2^l と表記する (つまりライン数, ラインサイズはそれぞれ s, l ビット幅で表現される). 仮想記憶上のアドレスとセットとの対応は, アドレスの下位 $s+l$ ビットのうち上位の s ビット値のセットで一意に決定される. ただし, このままでは s ビットが一致するアドレスは無数に存在するため, 複数の格納場所を設けなくてはならない. これが, バンクであり最大で n 個の s ビットが一致するアドレスをキャッシュ上に格納することができる. 逆に言えば n 個しか保持できない強い制約が存在するともいえよう.

アドレスの下位 $s+l$ ビットを落とした値をタグ部分に格納した way が存在すれば, キャッシュ上の (set,way) で定まる場所に指定アドレスを含むデータが格納されていることになる. 一方, いずれの way にもタグ部分に対応するアドレス情報がない場合は, キャッシュミスの状態にあると云う. どの way を選択するかは, 通常 **LRU(Least Recently Used, 最長未使用時間)** 方式で行われる. 「最近もっとも使われていない」way が選択され主記憶上のキャッシュラインが格納されることになる. 実際利用者がどの way を利用するかは制御できないし, 特に注意を払う必要もない (但し, これ以降の議論では制御できると便利な面もあるのだが...).

[1] マイクロプロセッサによって物理記憶上のアドレスとなる場合があるが, ここでは仮想記憶上のアドレスを前提とする.

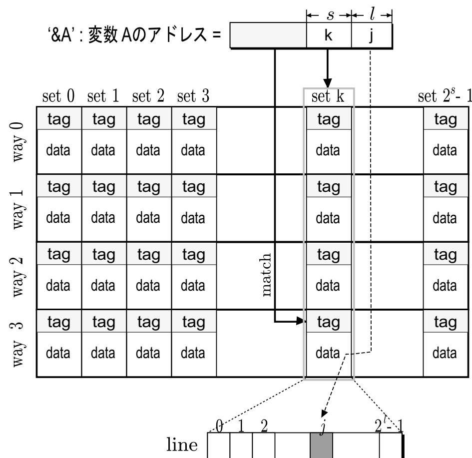


図 1: 4-way セットアソシアティブキャッシュの概念図

次節以降の説明を簡略化するために次のような表記方法を導入する．もし, データ A がキャッシュ上の set x, way y に対応するとき, ただし way y は LRU などによって適時予想がつく際に y を決めることとし, 定まらない場合は可能性のあるいずれかの way が対応するものと約束する．このとき, **データ A はキャッシュ上の (x,y) に格納される**と表記することにする．

さて, 近代的なプロセッサでは, キャッシュは階層構造をなしており, L1(レベル 1) から L2, L3 の 3 階層のものも存在する．L1 はプロセッサに最も近い位置に設置され高速かつ低レインテシであるが容量は小さい．逆に, L2,L3 となるに従って大容量かつ低速になる．更に, キャッシュと主記憶の間には仮想記憶のページと物理記憶装置との対応表を格納する **TLB(Translation Lookaside Buffer)** が存在する．近代的な OS では物理メモリではなく仮想記憶によるページ単位のメモリ管理がされており, そのページ管理システムとして殆どのマイクロプロセッサには備わっている．TLB はキャッシュと同様の構造をしているが, より高価なフルセットアソシアティブ方式 (場合よってはフルではない) をとることが殆どであるため, エントリ数はキャッシュのライン数よりもずっと少ない．また, TLB ミスのペナルティはキャッシュミスよりもずっと大きいといわれている．

2　キャッシュミスによる性能不安定化

本節では幾つかの例を挙げながらキャッシュミスによる性能不安定化の原因とその処方箋について説明をする．説明の中で fortran プログラムを基にするがこれは fortran の多次元配列の持つ連続性や整合寸法の特徴を利用するためである．C 言

語などに精通した読者は, fortranの配列のとり方に注意をしながら読んでいけば特に問題はないであろう.

先節の説明にあるように、キャッシュは主記憶に比べ小規模ではあるがアクセスが数クロック単位でなされるため, 高性能計算ではデータを如何にキャッシュ上にとどめておくかが鍵になる. データがキャッシュ上にない状態つまり「キャッシュミス」を避けることが重要なのであるが, では, どのような場合にキャッシュミスが起こるのであろうか? 参考書によく現れるのが次に3つのミスである.

1. 初期参照ミス (compulsory miss) キャッシュラインを最初にアクセスするときに起こるミス.
2. 容量性ミス (capacity miss) キャッシュしたいライン数がキャッシュ容量を上回ることで起こるミス.
3. 競合性ミス (conflict miss) 同じインデックスをもつ異なるキャッシュラインへのアクセスが発生することで起こるミス.

2.1 初期参照ミス (compulsory miss)

1. のミスはプロセッサとキャッシュの関係から避けることができない. 通常大きな配列は動的に (`malloc`などで) 確保されるが, 確保した段階ではメモリ領域にタッチしないためメモリがキャッシュにロードされることはない. 実際, 初めてそのメモリ領域にアクセスするときにメモリ–キャッシュ間の転送が発生する. 1は避けることができないキャッシュミスではあるが, 最近のマイクロプロセッサにはプリフェッチ (prefetch) 機能が備わっており, データを実際に利用するよりもずっと早い段階でアクセスしてキャッシュへのロードを完了させておくことができる. コンパイラの最適化やマイクロプロセッサが持つハードウェアプリフェッチの機能があるので, 利用者は特に気にする必要がないものである.

2.2 容量性ミス (capacity miss)

2. のミスは利用者がプログラム中で利用するメモリサイズとキャッシュサイズのアンバランスから生じるもので, その処方箋は幾つか存在する. プログラム上で利用するデータをキャッシュ上に留めておき, キャッシュの効果 (短時間でのデータアクセス) を得ることがもともとのキャッシュの考え方である. 一方, キャッシュは非常に高価でメモリサイズと比較してほんの僅かしかプロセッサ上には搭載されないため, 通常のプログラムではプログラム実行中常に全データをキャッシュ上に留めることはできない. そこで, プログラムのある区間に限定しデータの一部をできる限りキャッシュに留める (局所化する) ことでキャッシュの恩恵を得ることができる. よく知られた手法として, 行列·行列積のブロック化がある. また, ブロック化

ができなくてもプリフェッチの恩恵が得られるような場合には, 少々のアンバランスがあっても気にする必要は無い.

2.3 競合性ミス (conflict miss)

2. のミスの処方箋は局所化により利用するデータ (ブロック) をキャッシュサイズに抑えることであり, それを意識的にプログラムに反映することが容易である. 一方, 3. のミスはキャッシュの構造に由来するものであることから, その原因を理解することが容易ではない. プログラムの性能不安定性の主要な部分を占めるにも関わらず, 原因不明と処理されてしまうことも多いのではないだろうか. (実際, プロセッサのタイミングカウンタを見ることで初めてキャッシュミスが性能劣化の原因と判明した場合, このタイプのキャッシュミスであることがある)

簡単な例を紹介しよう. あるプログラムのコアループ中で 1 次元配列 A, B, C, D が利用されているとする. このとき, 各配列がページアライメントされていると, 各配列の同一インデックス要素は同じキャッシュラインへのアクセスに格納される. ここでは, 各配列の先頭 A(1) などが (0,*) に対応するアライメントを仮定する[2]. キャッシュが 4way で 1 ラインが 2 ワードに相当するとき, 各配列は次図 2 のように格納されるであろう.

	(0,*)		(1,*)		(2,*)		···
(*,0)	A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	···
(*,1)	B(1)	B(2)	B(3)	B(4)	B(5)	B(6)	···
(*,2)	C(1)	C(2)	C(3)	C(4)	C(5)	C(6)	···
(*,3)	D(1)	D(2)	D(3)	D(4)	D(5)	D(6)	···

図 2:

この状態で, 配列 A,B,C,D は全く衝突することなくキャッシュに収まることが理解できます. また, コアループが終了しても配列 A,B,C,D の総量がキャッシュサイズを超えなければ, 配列 A,B,C,D はキャッシュ内に留まることも理解できるであろう. 再度コアループに入ったとしても, キャッシュ内に留まるデータを高速に利用できるため, 性能はほぼ最高性能に達すると期待できる.

一方, このコアループにもう一つ配列 E(同じくページアライメントされているとする) の利用が追加された場合に, キャッシュの利用はどうなるだろうか? 答えは「キャッシュミスが発生し性能は数十分の一に劣化する」である.

まず, ライン (0,*) の利用を考えて見よう. (0,*) を利用するのは

[2] ここでは容量の小さい L1 キャッシュ利用を想定するが, ページサイズがキャッシュ1way 分よりも大きいことを仮定する. 大容量のキャッシュの場合はページアライメントはキャッシュ上ランダムになるためこの限りではないが, 基本的議論は同様におこなえるだろう.

{A(1),A(2)},{B(1),B(2)},{C(1),C(2)},{D(1),D(2)},{E(1),E(2)}

です. ここで, 中括弧で囲んだ単位がキャッシュラインに相当します.

A(1)→B(1)→C(1)→D(1)→E(1)→A(2)→ ···

のようなアクセスがループ内でされたら, (0,*) は 4way しかないので D(1) までのアクセスについては問題なく進行するが, E(1) をアクセスした時点で A(1) が入っているライン (0,0) をメモリに書き戻して, その部分に E(1)(実際は {E(1),E(2)} のライン) を格納する. そして, 次の A(2) のアクセス時には {A(1),A(2)} のラインはもはやキャッシュ上にないので, メモリにアクセスしないといけない. このとき, B(1) が入ったライン (0,1) をメモリに書き戻して, A(2) の入ったラインを (0,1) に格納することになる. この様に, 常にキャッシュラインの奪い合いが発生し, 連鎖的なキャッシュミスが発生する. このような現象をキャッシュスラッシング **(Cache thrashing)** と呼ぶことがある.

キャッシュスラッシングはハードウェアから見たらキャッシュの連想性 (way 数) の低さが原因ともいわれるが, ソフトウェアからこの現象をある程度解決することができる. まず, 解消すべきはデータ利用時の同一キャッシュラインへのアクセスをなくすことである. もし, 配列 E がページアライメントされてなく E(1) がキャッシュ上の (1,*) にあったとしよう. このとき,

A(1)→B(1)→C(1)→D(1)→E(1)→A(2)→ ···

のようなアクセスがあったとしても, それらのアクセス中にはキャッシュ競合は起こらないことは理解できるであろう. A(1), B(1), C(1), D(1), E(1) がアクセスされたときのキャッシュの状態は以下のとおりである (図 3).

	(0,*)	(1,*)	(2,*)	···
(*,0)	A(1) A(2)	E(1) E(2)		···
(*,1)	B(1) B(2)			···
(*,2)	C(1) C(2)			···
(*,3)	D(1) D(2)			···

図 3:

先の例の様に, E(1) のアクセス時に A(1) をメモリに追いやって ···, といったラインの競合の連鎖は結果として発生しなくなる. ただし, 容易にわかるようにコアループを終了し再度同じコアに突入する際には, 配列 E の殆どはキャッシュ上に残っていないことが判る. しかし, 配列 A,B,C,D はキャッシュ上にあるためその分はキャッシュの恩恵を得ることができる. ここで, A は E 利用時に上書きされるため B,C,D の効果のみがあることに注意したい. 1. の説明でも言及したように, A,E などのメモリ–キャッシュ間アクセスが発生しても, 一般にプリフェッチによってア

クセス待ちのペナルティを隠すことができる．ただし, プリフェッチは数ステップ先までのデータ利用について発行されるので, 上の例の様にE(1)が(1,*)に対応していては, Aのプリフェッチされたデータと Eのプリフェッチされたデータが衝突することになり, 結局両者のプリフェッチ効果が得られなくなる．したがって, このような場合にはプリフェッチの先読みよりも十分離れた位置にE(1)がくるようにアライメントすればよいということになる．アライメントの方法はいくつか候補があるが, 動的に確保する配列の場合キャッシュの1行分(1way 分)を余分にとった上で先頭アドレスから数ライン分ずらした位置を配列の先頭として利用すればよいということになる．なお, 静的な場合も同様の手法が使える．

2.4 キャッシュ性能不安定性とチューニング

ここまで簡単な競合性ミスの例を説明してきた．基本的には1次元配列で生じることを例示したが, より高次元配列でも当然同様の事例が発生する．また, 性質が悪いことに多次元配列であるが故に発生する場合もある．さらに, 本節のタイトルに挙げたように性能を上げるために行うチューニング作業の結果として, ある特定の条件化で競合性ミスが発生することが知られている．ここでは, チューニングと結びつけて解説をしていく．

まず, 次に挙げるfortranプログラムは特に何のチューニングを施していない行列-ベクトル積を計算するループである．

```
integer :: LDA, N
real    :: A(LDA,N), X(N), Y(N)
Y(1:N)=0.0
do J=1,N
   do I=1,N
      Y(I)=Y(I)+A(I,J)*X(J)
   enddo
enddo
```

このプログラムをチューニングする際に, 外側のループを展開するループアンローリングを行うことがある．Nが2で割り切れると仮定して, 次の様に変形する．

```
integer :: LDA, N
real    :: A(LDA,N), X(N), Y(N)
Y(1:N)=0.0
do J=1,N,2
   do I=1,N
      Y(I)=Y(I)+A(I,J)*X(J)+A(I,J+1)*X(J+1)
   enddo
```

```
      enddo
```

この場合, A(I,J)*X(J) の計算量に対して Y(I) へのアクセス数を比較すると, プログラム変形前に比べて半分になることがわかる. その分 (つまり計算に要する時間が増加するという意味で), 配列 Y がキャッシュ上に長く留まるようになるため性能が向上するのである. 筆者は文献 [1] で, A が対称行列のときの性能測定を行っている. その際のプログラムは以下のようなものである.

```
integer :: LDA, N
real    :: A(LDA,N), X(N), Y(N)
do J=1,N
   do I=1,N
      Y(J)=Y(J)+A(I,J)*X(J)
      Y(I)=Y(I)+A(I,J)*X(J)
   enddo
   Y(J)=Y(J)+A(J,J)*X(J)
enddo
```

このプログラムを以下のようにループアンローリングによりチューニングを行った. (ただし, 下三角要素は 0 であると仮定する)

```
integer :: LDA, N
real    :: A(LDA,N), X(N), Y(N), D(N)
do J=1,N
   D(J)=A(J,J); A(J,J)=0.0
enddo
do J=1,N,2
   do I=1,J
      Y(J+0)=Y(J+0)+A(I,J+0)*X(J+0)
      Y(J+1)=Y(J+1)+A(I,J+1)*X(J+1)
      Y(I)=Y(I)+A(I,J+0)*X(J+0)+A(I,J+1)*X(J+1)
   enddo
enddo
do J=1,N
   Y(J)=Y(J)+D(J)*X(J); A(J,J)=D(J)
enddo
```

このプログラム変形が正しいことは読者の皆さん各自で行って欲しい. この段階ではループアンローリングは 2 段しか行われていないが, これをプロセッサが有するレジスタ資源が許す限りの段数で行えば性能が向上していくことが知られている.

ここで, この性能の上昇は一般的に正しいのであるが, ある特定の状況下ではNGであることを確認できる. 文献[1]では, 行列の次元を4032から4160次元まで1次元毎に(4096次元を中心に±64次元)性能測定がなされている. 測定した計算機は日立製のSR8000F1モデルの1PEで, プロセッサはPower3に準拠し擬似ベクトル機能を追加したものである. また, キャッシャはL1のみで128KBの4way構成である. ループアンローリングは最大15段まで可能であり, その測定結果を次図4に示す(図中の'M-数字'の, 数字部分がアンローリング段数を指す).

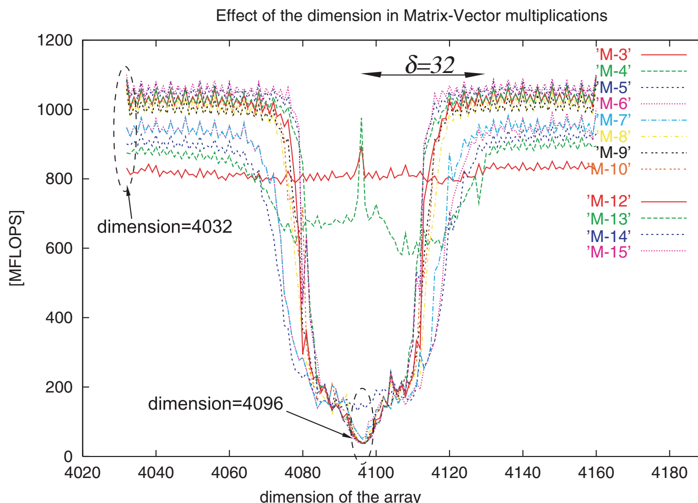


図 4: SR8000F1での行列ベクトル積の性能測定結果

図4は横軸に行列の次元(サイズ), 縦軸に実行時間から算出した行列ベクトル積ルーチンの性能(MFLOPS, FLOPSは1秒間あたりに行われた浮動小数点演算量, Mは10^9を表す単位)をプロットしたものである. なお, 同図は性能測定結果のほんの一部分を拡大したに過ぎないものである. 4080次元から4120次元あたり(4096の前後32次元)で大きな性能の劣化が見られるが, ほとんどの次元では性能はほぼ安定している.

- では, この性能曲線の落ち込みの原因は何だろうか?
- また, 底の中心である4096次元の4096という数値には何の意味があるのだろう?

答えを先に示してしまうと, 前節で解説した競合性ミスによるキャッシュスラッシングが発生しているのである. 先に示したプログラムでLDAなる整合寸法が指定されていたが, この値はNと同じ値を採っていた事が問題を生じさせることとなったのである. これについて説明しよう. fortranでの多次元配列は, 整合寸法

配列 (英語では leading dimension とも呼ばれることがある) により, 1 次元データを折り返し多次元配列表現をしている. このとき, A(1,1) と A(2,1) は必ず連続メモリ上にとられる (仮想記憶でのページ境界の場合を除く), 一方 A(1,1) と A(1,2) は整合寸法配列語分離れている. ループ中に A(I,J),A(I,J+1) といったアクセスがあった場合, A(I,J+1) は A(I,J) から **(整合寸法) % (キャッシュサイズ/way 数/語長 (倍精度なら 8 バイト))** 離れた位置 (語長単位で) に格納されることになる.

SR8000F1 モデルの場合, **(キャッシュサイズ/way 数/語長)** は **4096 語 (=128K/4/8)** に相当する. したがって, N=LDA=4096 のとき A(I,J) や A(I,J+1) などのアクセスは同じキャッシュラインを要求することになる. 一方, way 数は 4 しかないので, 4 段のループアンローリングまでは配列 A によるキャッシュスラッシングは発生しないが, 5 段以上ではキャッシュスラッシングは避けられない. 実際にはベクトルデータが 2 本 (X と Y) あるためそのアクセスも含めると, 3 段までしかスラッシングを回避できないことになる.

では, 次に N が 4096 の近傍のときはどう解釈すべきなのか? N=4097 のとき配列 A がキャッシュに収まる様子を調べてみる. SR8000F1 のキャッシュラインは 128 バイト (倍精度浮動小数点で 16 語) なので, これまでの図とは若干異なる. 図 5 中は添え字のみの省略形で書き出した.

	(0,*)					···
(*,0)	(1,1)	(2,1)	···	(15,1)	(16,1)	···
(*,1)		(1,2)	···	(14,2)	(15,2)	···
(*,2)			···	(13,3)	(14,3)	···
(*,3)			···	(12,4)	(13,4)	···

図 5:

4 段アンローリングを施した場合の J=1 でのキャッシュの様子を示している. この状態でも, 配列 A のみでライン (0,*) を占有しているため, キャッシュスラッシングは必ず発生します. このキャッシュの充填は規則的であり, 賢明な読者であれば配列の折り返しである整合寸法を LDA=4096+6=4102 とすれば, 4 段目の A(1,4) は A(1,1) から 6*3=18(> 16) 離れており, (0,3) ではなく (1,3) に格納されるためスラッシングを回避できることに気づくであろう (図 6).

ただし, 先のグラフを見るところでは 4 段のアンローリングの性能は 4130 次元程度まではよくない. これは, SR8000 がプリフェッチをかなり先読みしているためである. グラフから見て取れるように, もう 2 ライン分ほど大きめにとらないと駄目だという事が判る.

したがって, 本件の ad hoc な処方箋として

- 多次元配列の整合寸法は 4096+ラインサイズ*C (C は 1 より大きな整数)

	(0,*)					(1,*)			⋯
(*,0)	(1,1)	(2,1)	⋯	(15,1)	(16,1)	(17,1)	(18,1)	⋯	⋯
(*,1)			⋯	(9,2)	(10,2)	(11,2)	(12,2)	⋯	⋯
(*,2)			⋯	(3,3)	(4,3)	(5,3)	(6,3)	⋯	⋯
(*,3)							(1,4)	⋯	⋯

図 6:

が挙げられる. ここで, ラインサイズ*C を加えた別の理由として, キャッシュを先頭から必ず利用することでキャッシュ容量を最大限に使い切ることを意図している.

ここで挙げた方法は, 他の多くのプロセッサにあてはまるはずである (ただし, 数値 4096 はキャッシュの構造に依存する).

2.5 スラッシングはあらかじめ予測できないのか?

では, ここで「k 段のアンローリングをしたけれども, ループ内に登場する配列 A の整合寸法が N_A のときスラッシングは起こるのか?」という判定をしたくなる. この様な判別が容易にできれば, 性能劣化のポケットからプログラムを救助でき, 結果的に性能向上をソフトウェアの視点から実施できるはずである.

この判定については, 先のキャッシュの充填パターンから判るように, ある種の規則性 (周期性) を持つので, 簡単な剰余計算で判別することができる. 詳細は文献 [2] によるが, 判別式のみを書けば次のようになる.

> **仮定:** n-way 連想記憶キャッシュを想定し 1way には 2^L 語格納可能とする (つまり, キャッシュの総容量は $n * 2^L$ 語である). また, 整合寸法を N_A と書くことにする.
>
> 今, コアループ中で k 段のループアンローリングを施した際, 以下の不等式を成立させるような整数 i が存在するとき, キャッシュスラッシングが起こる可能性がきわめて高い. ただし, 定数 δ はラインサイズもしくはその数倍を指す.

$$0 < \exists i < k/n, |\mathrm{mod}(i * N_A + 2^L, 2^{L-1}) - 2^{L-1}| < \delta \qquad (1)$$

したがって, 上記の不等式を成立させないような N_A を選んで, 配列を確保すればよいのである.

2.6 さらに精密なキャッシュ性能安定化

ここまで熟読された読者は, 先節は配列 A 単体のアンローリングに対する処方箋であったことを見抜いた事であろう. そして, 処方箋の要点は「データのアクセスパターンに応じてデータレイアウトに細心の気をつけよ」ということを理解したであろう. 実は, 複数配列が同一ループ内に登場した場合には配列単体で登場する以上に競合性ミスの可能性は高くなる.

先節の記号と手法を流用することにしよう. 2 次元配列 A, B が同時にループ内に登場し, さらにそのループをアンローリングしたら競合性ミスの最も簡単な例で示したように way 数を超えた同一ラインへのアクセスが発生する. そこで, B(1,1) のアドレスを A(1,1) から最も離れた位置 (2^{L-1},*) に設定すれば多段のアンローリングを設定しても競合の問題は起こりにくくなる. 配列数がさらに多くなった場合は, お互いの距離が最大となるように配置すれば競合の可能性は下がるはずである. なお, 配列のアクセスパターンが決定的であっても配列数が増加したり高次元配列利用となると完全に競合を起こさない配置を決めることは困難になる. 可能であればループ内の配列数の利用を適切なものに制限するなどしたほうがよいであろう.

3 さいごに

キャッシュミスは容量性ミスばかりが気にされがちであるが, 極端な性能劣化は競合性ミスによって誘発されるキャッシュスラッシングによるものであることが多い. それらは発見が困難であり, ハードウェアカウンタなどを用いない限り見落とされがちである. 本記事の主張は「注意すべきキャッシュ不安定性はデータアクセスパターンとデータレイアウトが相互に影響しあうことで起こるキャッシュスラッシングによるもの」ということであり, それらにも注意をしたプログラミングや性能解析を行うべきであるということである. それらの多くは, 記事内で紹介したように配列データを注意深く配置することによりソフトウェア側からそれを制御し, 性能安定化を行うことが可能である. 本記事の読者の皆様はこの点に注意を払って高性能プログラムの作成に役立てて頂ければと願います.

参考文献

[1] 今村俊幸, 直野健 : 性能安定化を目指した自動チューニング型固有値ソルバーについて, 先進的計算基盤シンポジウム SACSIS2003 論文集, pp.145-152, 2003.

[2] 今村俊幸, 直野健 : キャッシュ競合を制御する性能安定化機構内蔵型数値計算ライブラリについて, 情報処理学会論文誌コンピューティングシステム, No. SIG 6 (ACS 6), Vol. 45, pp. 113-121, 2004.

FFTにおけるキャッシュ最適化方式

高 橋 大 介

筑波大学大学院システム情報工学研究科／
計算科学研究センター

1　はじめに

高速 Fourier 変換（fast Fourier transform、以下 FFT）は、科学技術計算において今日広く用いられているアルゴリズムである。

多くの FFT アルゴリズムは処理するデータがキャッシュメモリに載っている場合には高い性能を示す。しかし、問題サイズがキャッシュメモリのサイズより大きくなった場合においては著しい性能の低下をきたす。FFT アルゴリズムにおける 1 つの目標は、いかにしてキャッシュミスの回数を減らすかということにある。

近年のプロセッサの演算速度に対する主記憶のアクセス速度は相対的に遅くなってきており、主記憶のアクセス回数を減らすことは、より重要になっている。したがって、キャッシュメモリを搭載したプロセッサにおける FFT アルゴリズムでは、演算回数だけではなく、主記憶のアクセス回数も減らすことが重要になる。ここで、キャッシュミスの回数を減らすことができれば、主記憶のアクセス回数を減らすうえで非常に効果があるといえる。

本稿では、FFT におけるキャッシュ最適化方式について述べる。

2　高速 Fourier 変換

2.1　離散 Fourier 変換

FFT は、離散 Fourier 変換（discrete Fourier transform、以下 DFT）を高速に計算するアルゴリズムとして知られている。

DFT は次式で定義される。

$$y_k = \sum_{j=0}^{n-1} x_j \omega_n^{jk}, \quad 0 \le k \le n-1 \tag{1}$$

ここで、入力 x_j および出力 y_k は複素数の値であり、$\omega_n = e^{-2\pi i/n}$、$i = \sqrt{-1}$ である。

式 (1) に従ってそのまま n 点 DFT を計算すると、$O(n^2)$ の計算量が必要になるが、FFT の手法を用いることで、$O(n \log n)$ の計算量で n 点 DFT を計算することが可能である。FFT については、非常に多くの参考書が出版されているが、文献 [4, 16] 等を参考にされたい。

FFT アルゴリズムとしては Cooley-Tukey アルゴリズム [6] や、その変形である Stockham アルゴリズム [5, 13] が良く知られている。

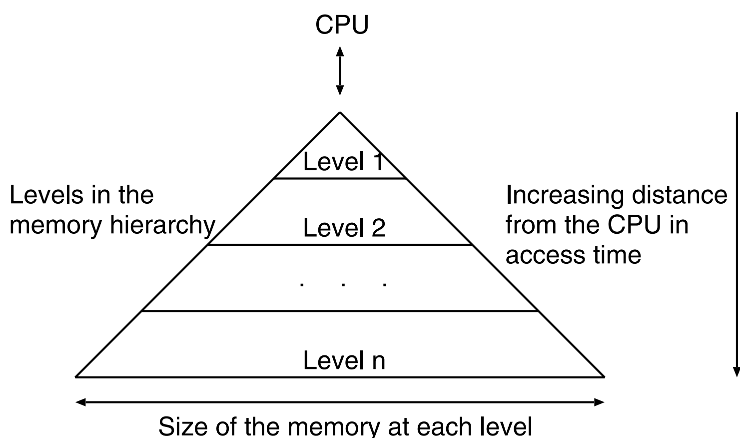


図 1: メモリ階層の例

2.2 FFT カーネル

FFT カーネル [16, 4] は FFT において、最内側のループで計算される処理であり、FFT カーネルの基数（radix）を p で表すと、次式で表される。

$$Y(k) = \sum_{j=0}^{p-1} X(j)\Omega^j \omega_p^{jk} \tag{2}$$

ここで Ω はひねり係数（twiddle factor）[4] と呼ばれる 1 の原始根であり、複素数である。

基数 p の FFT カーネルでは、入力データ $X(j)$ にひねり係数 Ω^j を掛けたものに対して p 点のショート DFT[10] が実行される。

式 (2) を計算するために、これまでにさまざまな手法が提案されている [12, 15]。

3 メモリアクセスの局所性

3.1 メモリ階層

メモリ階層（memory hierarchy）の例を図 1 に示す。メモリ階層は記憶域に対するアクセスパターンの局所性（locality）を前提に設計されている。局所性には時間的局所性と空間的局所性があり、前者は、ある一定のアドレスに対するアクセスは、比較的近い時間内に再発するという性質、後者は、ある一定時間内にアクセスされるデータは、比較的近いアドレスに分布するという性質である。

これらの傾向は、事務計算などの非数値計算プログラムには当てはまることが多いが、数値計算プログラムでは一般的ではない。特に大規模な科学技術計算においては、データ参照に時間的局所性がないことが多い。これが、科学技術計算でベクトル型スーパーコンピュータが有利であった大きな理由である。

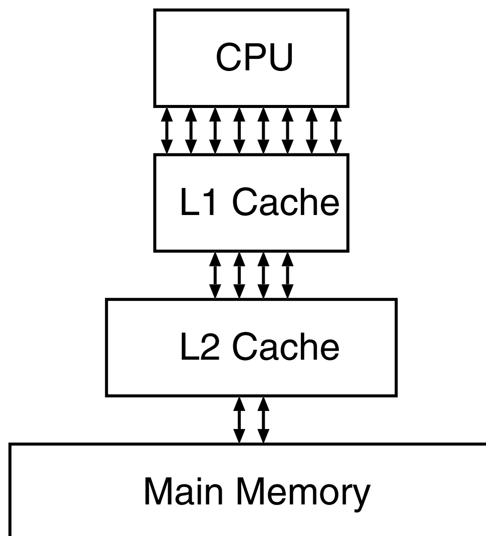


図 2: キャッシュメモリ

ベクトル型スーパーコンピュータはデータに関してキャッシュを用いないが、RISCプロセッサのようなスカラプロセッサは性能をキャッシュに深く依存している。したがって、RISCプロセッサで高い性能を得るためにはブロック化を行うなどして、意図的にアクセスパターンに局所性を与えることが必要になる。

多くの計算機では、メモリとレジスタの間にキャッシュメモリが入っている。キャッシュメモリはメモリとレジスタの中間的な性質を持ち、メモリ内のデータや機械語命令のうち使用頻度の高い部分がキャッシュメモリに入り、さらに使用頻度の高い部分がレジスタに入る。

キャッシュメモリは通常データが入るデータキャッシュと機械語命令が入る命令キャッシュに分かれている。そのうちチューニングに特に関係があるのはデータキャッシュであるので、本稿ではデータキャッシュについて説明する。

3.2 キャッシュミス

メモリ階層のうち、上位階層の小容量高速メモリを、通常キャッシュメモリあるいはキャッシュと呼ぶ。

キャッシュについても、階層的に構成することができる。つまり、より高速な1次キャッシュ（L1 Cache）の下に、アクセス速度は1次キャッシュよりも遅いが、容量は1次キャッシュよりも大きい2次キャッシュ（L2 Cache）を設けるのである。このようにすることにより、メモリ階層の局所性をより生かすことが可能になる。さらに3次キャッシュまで備えたプロセッサもある。

演算に必要なデータがキャッシュメモリにないため、メモリから一旦キャッシュメモリに転送せざるを得ないことをキャッシュミスという。キャッシュメモリよりもメモリの方が低速な半導体を使用しているため、データをメモリからキャッシュメモリに転送する時間はキャッシュメモリから

```
      SUBROUTINE ZAXPY(N,A,X,Y)
      IMPLICIT REAL*8 (A-H,O-Z)
      COMPLEX*16 A,X(*),Y(*)
C
      DO I=1,N
        Y(I)=Y(I)+A*X(I)
      END DO
      RETURN
      END
```

図 3: ZAXPY のプログラム例

レジスタに転送する時間の数倍かかる。したがってパフォーマンスを向上させるためにはキャッシュミスをできるだけ少なくする必要がある。

多階層のキャッシュを用いる場合には、1 次キャッシュのヒット時の速度を高速化するようにすること、2 次キャッシュのミスの割合を最小化するようにすること、が重要である。

3.3 ZAXPY ルーチンの性能

FFT カーネルは、主に複素数演算から構成されている。ここで、FFT カーネルに類似した ZAXPY（A X plus Y、倍精度複素ベクトルの積和）と呼ばれる演算を用いて、ベクトル長を変化させた場合の性能を測定した結果を図 4 に示す。

使用した計算機は Intel Xeon 3.06 GHz（FSB 533 MHz、512 KB L2 cache、PC2100 DDR-SDRAM）であり、コンパイラは Intel C Compiler 8.0 を用いている。

実行においては、Intel Pentium4 などに搭載されている SIMD（Single Instruction Multiple Data）命令である、SSE2 命令 [8] を用いたものと、従来の x87 命令を用いたもの、さらに Intel が提供している MKL（Math Kernel Library、Version 6.1.1）[9] の BLAS（Basic Linear Algebra Subprograms）で比較を行った。

ZAXPY は、図 3 に示すプログラムで記述することができ、1 回の iteration につき倍精度実数の加算、乗算がそれぞれ 4 回、load が 4 回、store が 2 回から成っている。

図 4 から分かるように、配列が L2 キャッシュに収まる領域（$N \leq 8192$）では、SSE2 命令を使ったプログラム（with SSE2）が最も高速である。この場合の最高性能は約 3 GFLOPS と、Xeon 3.06 GHz のピーク性能（6.12 GFLOPS）の約半分程度である。

ところが、L2 キャッシュを外れた場合には、x87 命令で実行した場合とほとんど同じ性能に低下してしまう。これは、メモリアクセスを少なくし、キャッシュの再利用性を高めることが高い性能を得るうえで不可欠であることを示している。

4 Six-Step FFT アルゴリズム

本章では、キャッシュメモリを有効に活用することのできる、six-step FFT アルゴリズム [3, 16] について説明する。six-step FFT アルゴリズムでは、一次元 FFT を二次元表現で表して計算することにより、キャッシュミスを少なくできるのが特徴である。

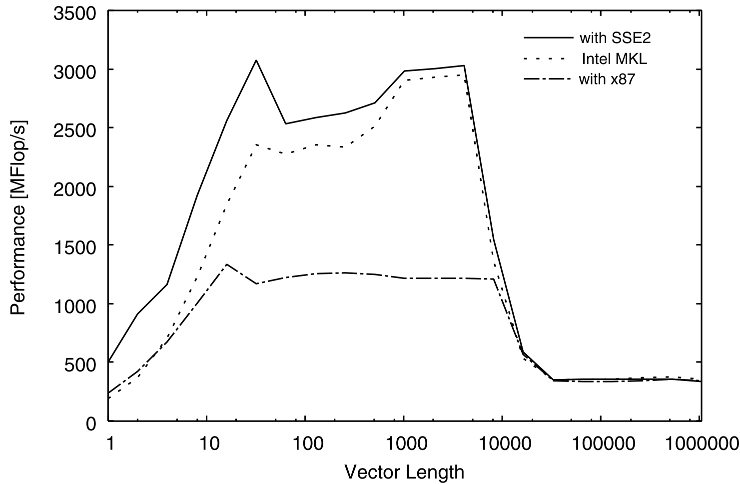


図 4: ZAXPY の性能（Intel Xeon 3.06 GHz）

n 点 DFT を計算する際に $n = n_1 \times n_2$ と分解できるものとすると、式 (1) における j および k は、

$$j = j_1 + j_2 n_1, \quad k = k_2 + k_1 n_2 \tag{3}$$

と書くことができる。そのとき、式 (1) の x と y は次のような二次元配列（columnwise）で表すことができる。

$$x_j = x(j_1, j_2), \quad 0 \le j_1 \le n_1 - 1, 0 \le j_2 \le n_2 - 1 \tag{4}$$

$$y_k = y(k_2, k_1), \quad 0 \le k_1 \le n_1 - 1, 0 \le k_2 \le n_2 - 1 \tag{5}$$

したがって、式 (1) は式 (6) のように変形できる。

$$y(k_2, k_1) = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} x(j_1, j_2) \omega_{n_2}^{j_2 k_2} \omega_{n_1 n_2}^{j_1 k_2} \omega_{n_1}^{j_1 k_1} \tag{6}$$

式 (6) から次に示されるような、six-step FFT アルゴリズム [3, 16] が導かれる。

Step 1: 転置

$$x_1(j_2, j_1) = x(j_1, j_2)$$

Step 2: n_1 組の n_2 点 multicolumn FFT

$$x_2(k_2, j_1) = \sum_{j_2=0}^{n_2-1} x_1(j_2, j_1) \omega_{n_2}^{j_2 k_2}$$

Step 3: ひねり係数の乗算

$$x_3(k_2, j_1) = x_2(k_2, j_1) \omega_{n_1 n_2}^{j_1 k_2}$$

Step 4: 転置

$$x_4(j_1, k_2) = x_3(k_2, j_1)$$

Step 5: n_2 組の n_1 点 multicolumn FFT

$$x_5(k_1, k_2) = \sum_{j_1=0}^{n_1-1} x_4(j_1, k_2)\omega_{n_1}^{j_1k_1}$$

Step 6: 転置

$$y(k_2, k_1) = x_5(k_1, k_2)$$

Step 3 における $\omega_{n_1n_2}^{j_1k_2}$ は、ひねり係数 [4] と呼ばれる 1 の原始根であり、複素数である。

図 5 に従来の six-step FFT アルゴリズムの疑似コードを示す。ここで、ひねり係数 $\omega_{n_1n_2}^{j_1k_2}$ は配列 U に格納されている。

従来の six-step FFT アルゴリズムの特徴を以下に示す。

- $n_1 = n_2 = \sqrt{n}$ とした場合、$\sqrt{n}$ 組の $\sqrt{n}$ 点 multicolumn FFT[16] が Step 2 と 5 で行われる。この $\sqrt{n}$ 点 multicolumn FFT はメモリアクセスの局所性が高く、キャッシュメモリを搭載したプロセッサに適している。

- 行列の転置が 3 回必要になる。

Step 1、4、6 の行列の転置および、Step 3 のひねり係数の乗算をブロック化した six-step FFT アルゴリズムが文献 [16] に示されている。しかし、この FFT アルゴリズムでは multicolumn FFT の部分と行列の転置の部分が分離されているため、multicolumn FFT においてキャッシュメモリに載っていたデータが行列の転置の際に有効に再利用されないという問題点がある。

ここで、3 回の行列の転置がキャッシュメモリを搭載した計算機においてボトルネックとなる。

5　ブロック Six-Step FFT アルゴリズム

本章では、さらにキャッシュ内のデータを有効に再利用し、キャッシュミスの回数を少なくするため、従来の six-step FFT では分離されていた multicolumn FFT と行列の転置を統合した、ブロック six-step FFT アルゴリズムについて説明する。前述した従来の six-step FFT において、$n = n_1n_2$ とし、n_b をブロックサイズとする。ここで、プロセッサは multi-level キャッシュメモリを搭載しているものと仮定する。ブロック six-step FFT アルゴリズムは以下のようになる。

Step 1: $n_1 \times n_2$ の大きさの複素数配列 X に入力データが入っているとする。このとき、$n_1 \times n_2$ 配列 X から n_b 列ずつデータを転置しながら、$n_2 \times n_b$ の大きさの作業用配列 WORK に転送する。ここでブロックサイズ n_b は配列 WORK が L2 キャッシュに載るように定める。

Step 2: n_b 組の n_2 点 multicolumn FFT を L2 キャッシュに載っている $n_2 \times n_b$ 配列 WORK の上で行う。ここで各 column FFT は、ほぼ L1 キャッシュ内で行えるものとする。

Step 3: multicolumn FFT を行った後 L2 キャッシュに残っている $n_2 \times n_b$ 配列 WORK の各要素にひねり係数 U の乗算を行う。そしてこの $n_2 \times n_b$ 配列 WORK のデータを n_b 列ずつ転置しながら元の $n_1 \times n_2$ 配列 X の同じ場所に再び格納する。

Step 4: n_2 組の n_1 点 multicolumn FFT を $n_1 \times n_2$ 配列 X の上で行う。ここでも各 column FFT は、ほぼ L1 キャッシュ内で行える。

Step 5: 最後にこの $n_1 \times n_2$ 配列 X を n_b 列ずつ転置して、$n_2 \times n_1$ 配列 Y に格納する。

```
 1 COMPLEX*16 X(N1,N2),Y(N2,N1),U(N2,N1)
 2 DO I=1,N1
 3   DO J=1,N2
 4     Y(J,I)=X(I,J)
 5   END DO
 6 END DO
 7 DO I=1,N1
 8   CALL IN_CACHE_FFT(Y(1,I),N2)
 9 END DO
10 DO I=1,N1
11   DO J=1,N2
12     Y(J,I)=Y(J,I)*U(J,I)
13   END DO
14 END DO
15 DO J=1,N2
16   DO I=1,N1
17     X(I,J)=Y(J,I)
18   END DO
19 END DO
20 DO J=1,N2
21   CALL IN_CACHE_FFT(X(1,J),N1)
22 END DO
23 DO I=1,N1
24   DO J=1,N2
25     Y(J,I)=X(I,J)
26   END DO
27 END DO
```

図 5: 従来の six-step FFT アルゴリズム

図 6 にブロック six-step FFT アルゴリズムの疑似コードを、図 7 にメモリ配置を示す。図 6 のアルゴリズムにおいて、NB はブロックサイズであり、NP はパディングサイズ、WORK は作業用の配列である。なお、図 7 において、配列 X、WORK、Y の中の 1 から 16 の数字は、配列のアクセス順序を示している。

また、作業用の配列 WORK にパディングを施すことにより、配列 WORK から配列 X にデータを転送する際や、配列 WORK 上で multicolumn FFT を行う際にキャッシュラインコンフリクトの発生を極力防ぐことができる。

ブロック six-step FFT アルゴリズムは、いわゆる *two-pass* アルゴリズム [3, 16] となる。つまり、ブロック six-step FFT アルゴリズムでは n 点 FFT の演算回数は $O(n \log n)$ であるのに対し、主記憶のアクセス回数は理想的には $O(n)$ で済む。

なお、本稿では Step 2 および Step 4 の各 column FFT は L1 キャッシュに載ると想定しているが、問題サイズ n が非常に大きい場合には各 column FFT が L1 キャッシュに載らないことも十分予想される。このような場合は二次元表現ではなく、多次元表現 [1, 2] を用いて、各 column FFT の問題サイズを小さくすることにより、L1 キャッシュ内で各 column FFT を計算することができる。ただし、三次元以上の多次元表現を用いた場合には *two-pass* アルゴリズムとすることはできず、例えば三次元表現を用いた場合には *three-pass* アルゴリズムになる。このように、多次元表現の次元数を大きくするに従って、より大きな問題サイズの FFT に対応することが可能になるが、その反面主記憶のアクセス回数が増加する。これは、ブロック six-step FFT においても性能

```
 1 COMPLEX*16 X(N1,N2),Y(N2,N1),U(N1,N2)
 2 COMPLEX*16 WORK(N2+NP,NB)
 3 DO II=1,N1,NB
 4   DO JJ=1,N2,NB
 5     DO I=II,II+NB-1
 6       DO J=JJ,JJ+NB-1
 7         WORK(J,I-II+1)=X(I,J)
 8       END DO
 9     END DO
10   END DO
11   DO I=1,NB
12     CALL IN_CACHE_FFT(WORK(1,I),N2)
13   END DO
14   DO J=1,N2
15     DO I=II,II+NB-1
16       X(I,J)=WORK(J,I-II+1)*U(I,J)
17     END DO
18   END DO
19 END DO
20 DO JJ=1,N2,NB
21   DO J=JJ,JJ+NB-1
22     CALL IN_CACHE_FFT(X(1,J),N1)
23   END DO
24   DO I=1,N1
25     DO J=JJ,JJ+NB-1
26       Y(J,I)=X(I,J)
27     END DO
28   END DO
29 END DO
```

図 6: ブロック six-step FFT アルゴリズム

はキャッシュメモリの容量に依存することを示している。

なお、out-of-place アルゴリズム（例えば Stockham アルゴリズム [5, 13]）を Step 2、4 の multicolumn FFT に用いたとしても、余分に必要となる配列の大きさは $O(\sqrt{n})$ で済む。

また、一次元 FFT の結果が転置された出力で構わなければ、Step 5 の行列の転置（図 6 では 24∼28 行目）は省略することができる。この場合、作業用の配列は $O(\sqrt{n})$ の大きさの配列 `WORK` だけで済むことが分かる。

6 In-Cache FFT アルゴリズムおよび並列化

前述の multicolumn FFT において、各 column FFT がキャッシュに載る場合の in-cache FFT には Stockham アルゴリズム [5, 13] を用いた。Stockham アルゴリズムは、Cooley-Tukey アルゴリズム [6] のように入力と出力が重ね書きできないために、入力と出力で別の配列が必要となり、その結果 Cooley-Tukey アルゴリズムの 2 倍のメモリ容量が必要になる。しかし、ビットリバース処理 [6] が不要であるという特徴がある。

ここで、基数 2 における Stockham のアルゴリズムについて説明する。

$n = 2lm$ とする。ここで l および m は 2 のべきであるものとする。l の初期値は $n/2$ とし、反

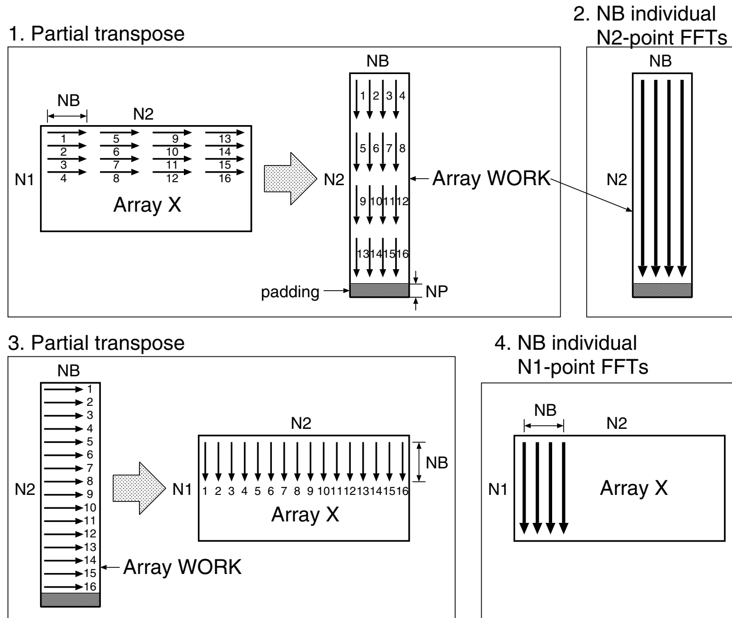


図 7: ブロック six-step FFT アルゴリズムのメモリ配置

復ごとに 2 で割っていく。m の初期値は 1 とし、反復ごとに 2 倍していく。配列 X は入力データであり、Y は出力データである。実際にインプリメントするときには、反復において、X から Y、Y から X に出力されるようにすると、メモリコピーを減らすことができる。ここで $\omega_p = e^{-2\pi i/p}$ である。

$$\begin{aligned} c_0 &= X(k+jm) \\ c_1 &= X(k+jm+lm) \\ Y(k+2jm) &= c_0 + c_1 \\ Y(k+2jm+m) &= \omega_{2l}^{j}(c_0 - c_1) \\ 0 \le j < l & \qquad 0 \le k < m \end{aligned}$$

2 点 FFT を除く 2 べきの FFT では、基数 4 と基数 8 の組み合わせにより FFT を計算し、基数 2 の FFT カーネルを排除することにより、ロードとストア回数および演算回数を減らすことができ、より高い性能を得ることができる [14]。具体的には、$n = 2^p$ $(p \ge 2)$ 点 FFT を $n = 4^q 8^r$ $(0 \le q \le 2,\ r \ge 0)$ として計算することにより、基数 4 と基数 8 の FFT カーネルのみで $n \ge 4$ の場合に 2 べきの FFT を計算することができる。

six-step FFT において、multicolumn FFT の各列は独立であるため、並列性が高いことが知られている [3, 16]。

共有メモリ型並列計算機における並列化は以下のように行うことができる。図 5 に示した従来の six-step FFT アルゴリズムにおいては、2、7、10、15、20、23 行目の各 `DO` ループを並列化（OpenMP[11] の `!$OMP DO` ディレクティブを挿入）する。

表 1: デュアルコア Intel Xeon 5150（2.66 GHz）における FFTE 4.0 （SSE3）の性能

n	1 CPU, 1 core		1 CPU, 2 cores		2 CPUs, 4 cores	
	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS
2^{12}	0.00006	4128.46	0.00006	4128.80	0.00006	4141.40
2^{13}	0.00014	3912.61	0.00014	3900.81	0.00014	3925.46
2^{14}	0.00028	4030.83	0.00029	4020.14	0.00028	4036.37
2^{15}	0.00060	4121.60	0.00060	4113.43	0.00060	4106.24
2^{16}	0.00143	3676.79	0.00141	3713.05	0.00141	3717.98
2^{17}	0.00500	2228.17	0.00380	2931.55	0.00226	4921.67
2^{18}	0.01340	1761.12	0.00747	3159.97	0.00472	4995.93
2^{19}	0.02989	1666.54	0.01678	2968.24	0.01341	3715.39
2^{20}	0.06675	1570.84	0.03735	2807.18	0.03003	3491.69

また、図 6 に示したブロック six-step FFT アルゴリズムにおいては、3、20 行目の各 `DO` ループを並列化する。なお、配列 `WORK` はプライベート変数にする必要がある。MPI を用いて並列 FFT プログラムを記述する際にも、共有メモリ型並列計算機の場合と同様にブロック化することが可能である。

7　ブロック Six-Step FFT の性能

本章では、ブロック six-step FFT を用いた FFT ライブラリである FFTE（version 4.0）[1]と、多くのプロセッサにおいて最も高速な FFT ライブラリとして知られている FFTW（version 3.1.2）[2][7] との性能比較を行った結果を示す。

性能比較にあたっては、$n = 2^m$ の m を変化させて順方向 FFT を連続 10 回実行し、その平均の経過時間を測定した。なお、FFT の計算は倍精度複素数で行い、三角関数のテーブルはあらかじめ作り置きとしている。

計算機としては、デュアルコア Intel Xeon 5150（クロック周波数 2.66 GHz、主記憶 4 GB DDR2-SDRAM、32 KB L1 instruction cache、32 KB L1 data cache、4 MB L2 Cache、Linux 2.6.18-1.2798.fc6）を用いた。

コンパイラは Intel Fortran コンパイラ（version 9.1）および Intel C コンパイラ（version 9.1）を用い、コンパイラの最適化オプションとして、``-O3 -xP -openmp'' を指定した。

表 1 に FFTE（version 4.0）と、FFTW（version 3.1.2）の性能を示す。表 1 から、ブロック six-step FFT を用いている FFTE では、特に 2CPUs、4cores でデータ数 n が大きくキャッシュメモリに収まらない場合に FFTW に比べて性能が高くなっていることが分かる。

8　まとめ

本稿では、FFT におけるキャッシュ最適化方式について述べた。

[1]http://www.ffte.jp
[2]http://www.fftw.org

表 2: デュアルコア Intel Xeon 5150（2.66 GHz）における FFTW 3.1.2 の性能

n	1 CPU, 1 core		1 CPU, 2 cores		2 CPUs, 4 cores	
	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS
2^{12}	0.00005	5340.65	0.00005	5324.67	0.00005	5370.66
2^{13}	0.00010	5246.95	0.00010	5250.29	0.00010	5321.29
2^{14}	0.00022	5288.20	0.00022	5238.77	0.00022	5331.34
2^{15}	0.00050	4959.85	0.00050	4955.32	0.00049	4971.35
2^{16}	0.00111	4722.97	0.00110	4782.46	0.00119	4405.83
2^{17}	0.00376	2963.07	0.00400	2785.17	0.00396	2811.49
2^{18}	0.00997	2366.58	0.01011	2333.85	0.01000	2358.56
2^{19}	0.02351	2118.26	0.02288	2177.28	0.02166	2299.11
2^{20}	0.05060	2072.33	0.04003	2619.43	0.03698	2835.30

ブロック six-step FFT に基づく並列一次元 FFT では、キャッシュメモリの再利用率を高くすることにより、キャッシュミスを少なくし、その結果主記憶のアクセス回数も少なくすることができる。データがキャッシュに入り切らないような大きな問題サイズの FFT では、ブロック six-step FFT は従来の FFT アルゴリズムに比べて有利である。特に、プロセッサの演算速度と主記憶のアクセス速度との差が大きい場合に、より効果的である。

本稿で述べたブロック化の手法は、他のアプリケーションの高速化にも適用できると考えられる。

参考文献

[1] R. C. Agarwal. An efficient formulation of the mixed-radix FFT algorithm. In *Proc. International Conference on Computers, Systems and Signal Processing*, pages 769–772, 1984.

[2] R. C. Agarwal and J. W. Cooley. Vectorized mixed radix discrete Fourier transform algorithms. In *Proc. IEEE*, volume 75, pages 1283–1292, 1987.

[3] D. H. Bailey. FFTs in external or hierarchical memory. *The Journal of Supercomputing*, 4:23–35, 1990.

[4] E. O. Brigham. *The Fast Fourier Transform and its Applications.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[5] W. T. Cochrane, J. W. Cooley, D. L. Favin, H. D. Helms, R. A. Kaenel, W. W. Lang, G. C. Maling, Jr., D. E. Nelson, C. M. Rader, and P. D. Welch. What is the fast Fourier transform? *IEEE Trans. Audio Electroacoust.*, 15:45–55, 1967.

[6] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comput.*, 19:297–301, 1965.

[7] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proc. IEEE*, 93:216–231, 2003.

[8] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference*, 2003.

[9] Intel Corporation. *Intel Math Kernel Library Reference Manual*, 2003.

[10] H. J. Nussbaumer. *Fast Fourier Transform and Convolution Algorithms*. Springer-Verlag, New York, second corrected and updated edition, 1982.

[11] OpenMP. Simple, portable, scalable smp programming. http://www.openmp.org.

[12] R. C. Singleton. An algorithm for computing the mixed radix fast Fourier transform. *IEEE Trans. Audio Electroacoust.*, 17:93–103, 1969.

[13] P. N. Swarztrauber. FFT algorithms for vector computers. *Parallel Computing*, 1:45–63, 1984.

[14] D. Takahashi. A parallel 1-D FFT algorithm for the Hitachi SR8000. *Parallel Computing*, 29(6):679–690, 2003.

[15] C. Temperton. Self-sorting mixed-radix fast Fourier transforms. *J. Comput. Phys.*, 52:1–23, 1983.

[16] C. Van Loan. *Computational Frameworks for the Fast Fourier Transform*. SIAM Press, Philadelphia, PA, 1992.

電磁場解析における数値ソルバのキャッシュ最適化チューニング

岩下　武史

京都大学学術情報メディアセンター

１．はじめに

本稿では，東京大学情報基盤センターからの依頼に基づき，具体的な応用分野における数値計算を対象としたキャッシュの影響について述べる。一般に，所謂スカラ型の計算機では，キャッシュを効果的に利用することが計算速度に大きな影響を与える。ここでは，著者がこれまでに取り組んできた電磁場解析[1,2]における線形反復法[3]を対象として，並列処理に関連したキャッシュの利用性について紹介する。まず，２節において電磁場解析に関する概説を述べ，３節以降では，電磁場解析においてよく用いられる線形反復法である ICCG 法（Incomplete Cholesky Conjugate Gradient 法）[4]を対象とした２種類のベンチマークについて，その並列化の詳細，数値実験結果について述べる。

２．電磁場解析

電磁場解析はモータや電子デバイスの設計や評価において産業応用，学術分野において欠くことのできないものとなっている。電磁場解析にはいくつかのカテゴリがあり，線形解析／非線形解析，時間領域／周波数領域，高周波問題／低周波問題等の区別がある。例えば，線形・非線形の解析の違いでいえば，非線形解析にはヒステリシス特性を持つ磁性体を扱うような解析が該当し，例えば鉄心を持つ電磁モータの解析が一例である。時間領域の解析とは，過渡現象等の時間的に変化する電磁場を扱う解析であり，周波数領域の解析とは電磁場が周期的に変化する場合を扱う解析である。次に高周波問題とは，例えば携帯電話やプリント基板から発生する電磁場を扱う問題であり，低周波問題とは実応用例で言えば，50Hz や 60Hz の商用電力で駆動されるモータ等を扱う。こうした各々の区分に対して，一般的によく使われる手法が存在しており,例えば線形の高周波問題を対象とする時間領域解析では FDTD 法が盛んに用いられている。一方，非線形の低周波時間領域問題や線形の高周波周波数領域解析では，辺要素を用いた有限要素法を使用するのが一般的である。有限要素法は電磁場解析のみならず，構造解析や流体の解析等においても広く用いられている手法である。そこで，ここでは有限要素解析について考えていく。

有限要素解析のプロセス（プログラムといってもよい）は以下のような手順による。（１）メッシュ分割（２）連立一次方程式の作成（３）連立一次方程式の求解　ここで，（１）のメッシュ分割は CAD ソフト等のメッシュ分割ソフトウェアにおいてなされるので，有限要素解析自体は（２）（３）のプロセスによると考えてよい。このうち，解析のほとんどの時間を占めるのが（３）の連立一次方程式の求解である。そこで，有限要素解析の高速化においては連立一次方程式の求解部（線形ソルバといわれる）の高速化が極めて重要である。

連立一次方程式の求解法には大きく分けて直接法と反復法（間接法ともいう）がある。一般

によく知られているガウスの消去法，LU 分解法は直接法に該当する。直接法におけるキャッシュチューニングは極めて重要で，様々な本に述べられている [5]。一方，有限要素解析で扱う連立一次方程式は疎行列を係数行列とするという特殊な性質を持っている。ここで，疎行列とは係数行列の大部分の要素が 0 である行列である。例えば，大規模な解析では，全体の要素数の 0.1%以下しか非ゼロの要素がないような解析は珍しくない。こうした疎行列を係数とする連立一次方程式の解法としてはメモリ使用量，計算時間の点で反復法が有利である。そこで有限要素解析では一般に反復法が使用される。反復法とは，ある初期予想解から出発し，ある種の計算手順を繰り返し適用することにより，真の解に対して解析上十分な精度をもつ近似解を得る方法である。(従って，対象とする連立一次方程式によっては近似解が収束せず，解を求められない場合がある。) 電磁場有限要素解析において最もよく用いられる反復法は ICCG 法である。ICCG 法は係数行列が対称な場合に適用可能な方法である。ICCG 法の反復部分は以下のような計算により構成される。(i)前進・後退代入計算，(ii)ベクトルの内積計算，(iii)行列ベクトル積，(iv)ベクトルの更新およびノルムの計算 である。このうち計算量の点で支配的であるのは，前進・後退代入計算と行列ベクトル積計算である。そこで，次節以降では，このうち並列化が困難であるとされている前進・後退代入計算を対象として，その並列化とそれに伴うキャッシュのヒット率への影響について述べることとする。

３．ICCG 法の差分解析ベンチマーク

本節では，有限要素解析とは異なるが，有限差分解析における ICCG 法について述べる。まず，ここで扱う問題は次式で与えられる 2 次元ポアソン方程式の境界値問題である。

$$-\nabla\cdot(\kappa\nabla u(x,y)) = f \qquad in \quad \Omega:(0,1)\times(0,1) \tag{1}$$

$$u(x,y) = 0 \qquad on \quad \partial\Omega \tag{2}$$

ここで，$0.25\leq x\leq 0.75$ かつ $0.25\leq y\leq 0.75$ を満たす領域では κ を 100 とし，それ以外では 1 とした。ここで f は，格子点を図 1 のように辞書式順序付けで並べた場合の格子番号を k として，$0.5\sin(k+1)$ である。 未知数の個数は 1001×1001 とする。 式(1)を 5 点差分公式により離散化し，得られる連立一次方程式を ICCG 法により解く。ここで，解析プログラム上において，係数行列と前処理行列は辞書式順序付け法に基づきそれぞれ 3 つの一次元配列と 1 つの一次元配列に格納される。疎行列に関連する計算において，データをどのようにメモリ上に確保するかという事は，キャッシュのヒット率や並列実行可能性に大きな影響を与える。そこで，このデータ格納方法について詳細に述べる。まず，辞書式順序付け法を用いた場合，係数行列は図 2 に示されるような 5 重対角行列として与えられる。これは，図 3 にみられるように一つの節点（○）は上下左右の 4 つの節点（□）と関係をもち，自分自身を含めて 1 行あたり 5 つの要素を持つこと，および，格子番号 k の左右の節点番号は辞書式順序付けでは $k-1$，$k+1$ であり，上下の節点の番号は $k+nx$，$k-nx$，但し nx は x 方向の格子点数であることによる。そこで，ここでは係数行列の対称性を考慮して，行列の上三角部分だけを格納することとし，格子番号が k の要素に対応する係数行列 k 行目の各要素を a(k),b(k),c(k)に格納する。また不完全コレスキー分解の結果得られる分解行列の対角要素が配列 d に格納されているものとする。このとき，ICCG 法の前進代入計算は図 4 のように与えられる（但し，ny は y 方向の格子点数)。ここで，このプログラムをみると，z(i)を求めるために z(i-1),z(i-nx)の値が必要な値となっている。従って，この前進代入計算はこのままでは並列処理することがでず，後退代入計算についても

同様である。そこで，本稿ではマルチカラーオーダリングを用いることによりこれらの計算を並列化する。マルチカラーオーダリングは並列オーダリング法[6]の1種である。並列オーダリング法とは，節点の番号を並列処理に適した順番に並び替えることにより代入計算の並列処理を可能とする方法である。ここで，マルチカラーオーダリングでは，互いに関係のない節点を複数抽出し，これを一つのグループ（色）とみなす。今回の5点差分解析の場合，互いに関係のない節点とは，お互いに上下左右の隣接節点でない節点を意味する。このとき，各グループ（色）の中では節点は互いに無関係となるため並列処理が可能となる。図5は2色によるマルチカラーオーダリングで赤－黒順序付けと呼ばれる。この場合，(1,2,3,4,5)の節点が1つの色を構成し，(6,7,8,9)がさらに1つの色を構成する。具体的に代入計算を並列化する場合，このような並列処理可能な順序に節点を陽的に並び替える方法が考えられる。この場合，例えば，図6のような係数行列が得られる。但し，本実装による場合，現在処理している節点の前後左右の節点の色番号と節点番号の取得のために間接アドレッシングが通常必要となる（著者の検討によると直接アドレッシングは不可能ではないが，実装は困難である)。そこで，本稿の解析におけるマルチカラーオーダリングの実装は，Washio，Hayamiが文献7に示している間接参照を必要とせず，計算順序のみを入れ替える方法を用いる。その結果，前進・後退代入計算は色数をストライド幅とするストライドアクセスをもつ計算手順により行われる。図7に同手法により並列化した前進代入計算のプログラムを示す。並列処理のAPIとしてOpenMPを使用している。図7において,myidは各スレッドのスレッドIDを表す。また,icolorは色数であり,is(myid),ie(myid)は各スレッドが計算する領域の開始行番号と終了行番号を表す。内側のループが各スレッドで並行的に行われ,その後バリア同期をとる。この操作が色数の数だけ行われる。なお,後退代入計算も同様の手順で並列化される。

ここで，図7のプログラムにおけるキャッシュデータの再利用性について考える。図7のプログラムでは，内側のループにおいてストライドアクセスを用いることにより並列化を可能としている。また，アルゴリズム上，このストライド幅は色数を意味し，色数が多いほど収束性がよい（逐次版の収束性に近づく）という性質がある。従って，ICCG 法の反復回数を削減し，全体の計算量を少なくするにはなるべく大きな色数を使用するほうが有利である。一方，ストライドアクセスはキャッシュデータの利用性という点では悪影響を与える。つまり，ストライドの幅を大きくすると，折角キャッシュにデータを保持してもそれらの一部しか使用しない，あるいは全く使用せずにキャッシュ上のデータが置き換わるということが生じる。従って，色数を増加することは本実装では1反復の計算時間を増加させることになる。また，並列性についてはなるべく色数が少ない方が有利である。これは，1 色あたりの未知変数の数が最大利用可能な並列度になること，また代入計算については色数に比例した同期が必要となるためである。以上をまとめると，色数の増加は　収束性：良化，反復あたりの計算時間（キャッシュ利用性)：悪化，並列性：不利　となる。従って，これらを適切に調整することが必要となる。

４．ICCG 法の差分解析ベンチマークの数値計算結果

４．１．実行環境

本ベンチマークは京都大学学術情報メディアセンターの富士通 HPC2500（SPARC64V 1.3GHz）上で行った。プログラムは FORTRAN90 により書かれ，並列処理の API として OpenMP を用いている。コンパイル時の最適化オプションには-Kfast_GP2=3 を指定した。また，ICCG 法の収束基準

として右辺ベクトルノルムと残差ノルムの比を用い，その値が 10^{-7} 以下となった時点で収束とみなす。

４．２．数値計算結果

表１に差分解析ベンチマークの結果を示す。なお，表１中において計算時間は反復の開始から終了までの間の経過時間を示しており，不完全コレスキー分解などの反復解法部のセットアップ部分の時間は含まれていない。まず，マルチカラーオーダリング法による並列化 ICCG 法の収束性については，色数を増やすほど向上しており，従来の研究結果と合致している [8,9,10]。一方，表２に 1CPU 時における１反復あたりの計算時間と色数の関係を示すが，色数が増すにつれて大幅に１反復あたりの計算時間を要していることがわかる。これはストライド幅が増加することにより，キャッシュのヒット率が大きく低下したことによる。2 色の場合と 100 色の場合を比較すると，5 倍以上も計算時間が増加しており，同じ計算量であってもメモリへのアクセスパターンが異なることにより大きな性能差が生まれることがわかる。ベクトル計算機上での実行では，バンクコンフリクトを起こす場合を除けば１反復あたりの計算時間は色数に対してあまり変化しないため，高い収束性が得られる（トータルの演算量の少ない）色数の多い場合が有効となるがキャッシュをベースとするメモリ構成をもつ計算機では必ずしもそうとはいえず，注意が必要である。次に，速度向上については概ね高い並列化効率を得ている。特に色数が多い場合にはスーパーリニアな性能を示しており，色数 100 の場合には顕著である。ここで，スーパーリニアな性能とは 1 台の CPU で計算した場合と比べて，N 台の CPU で N 倍以上の速度向上を得る場合を指す。本現象もキャッシュの影響と考えることができる。本ベンチマークでは問題のサイズが固定されており，並列化により各 CPU がアクセスするメモリ領域は使用 CPU が増加するにつれて減少する。その結果，データアクセスの局所性が高まり，キャッシュのヒット率が向上する。本解析におけるプロファイラによる分析では，色数 100 の場合の代入計算ループについて 1CPU 時と 16CPU 時で，L1 キャッシュミス率，L2 キャッシュミス率，TLB ミス率がそれぞれ 23%→1.5%，31%→0.3%，1.86%→0.0031%のように大幅に向上している。即ち，16CPU 時の高い台数効果はキャッシュの利用性が高まった結果といえる。最後に総合的な計算性能を考えると，本解析では色数 100，CPU 数 16 の場合が最もよい結果となった。これは，色数を多くした場合でも十分に各プロセッサが扱うデータサイズが小さい場合にはキャッシュのヒット率が向上し，収束性に優れた色数の多い場合が有効であることを示している。しかし，問題サイズに対してプロセッサ数が十分ではない場合には，最小の色数である２色の場合が有効となると考えられる。以下に，スカラ型の並列計算機における注意点をまとめる。

- 演算量だけでなくキャッシュの有効利用についても十分に配慮する必要がある。
- 問題のサイズが固定されている場合には並列化によりデータアクセスの局所化を促進できる可能性があるため，キャッシュのヒット率の改善により大きな効果を得られる可能性がある。

５．ICCG 法の有限要素解析ベンチマーク

本節では，実際の電磁場解析の応用分野で用いられる辺要素有限要素法による３次元渦電流場の解析を対象とする。解析対象内の電磁界を記述する方程式は，マクスウェル方程式において変位電流の項を無視することにより与えられる。本解析では，辺要素を使用し，磁気ベクトルポテンシャルのみによる定式化を行う A-法を用いるので，支配方程式は次式で与えられる [2]。

$$\nabla\times(\nu\nabla\times A_m) = -\sigma\frac{\partial A_m}{\partial t} + J_0 \tag{3}$$

ここで，A_mは磁気ベクトルポテンシャル，J_0は強制電流の電流密度，νは磁気抵抗率，σは導電率を表す。 磁気ベクトルポテンシャルをベクトル補間関数により近似展開し，式(3)にガラーキン法を適用することにより，次式が得られる。

$$[K]\{A_m\} + [M_A]\frac{\partial\{A_m\}}{\partial t} - \{J\} = \{0\} \tag{4}$$

ここで，$\{A_m\}$は未知変数 A_{mi}からなる列ベクトルを表す。［K］，［M_A］は行列，$\{J\}$は列ベクトルを表し，以下のように与えられる。

$$K_{ij} = \sum_{e_m}\iiint_e \nu(\nabla\times N_i)\cdot(\nabla\times N_j)dV \tag{5}$$

$$M_{Aij} = \sum_{e_m}\iiint_e \sigma N_i\cdot N_j dV \tag{6}$$

$$J_i = \sum_{e_m}\iiint_e N_i\cdot J_0 dV \tag{7}$$

ここで，eは各要素，e_m は全要素数，N はベクトル補間関数を表す。未知変数の総数を n として，行列$[K]$，$[M_A]$は n 次正方行列，$\{A_m\}$および$\{J\}$は n 次元ベクトルである。式(4)中の時間微分項を後退差分法により解くと，

$$Q\{A_m\} = \{f\} \tag{8}$$

但し，

$$[Q] = ([K] + \frac{1}{\Delta t}[M_A]) \tag{9}$$

$$\{f\} = \frac{1}{\Delta t}[M_A]\{A_{mold}\} + \{J\} \tag{10}$$

の連立一次方程式が得られる。ここで，本稿では解析対象として電気学会 3 次元渦電流解析モデル[11]を用いる。表 3 に解析の諸元を示す。本解析では，解析領域中に非導電性の部分（空気領域）が含まれるため，係数行列[Q]は半正定値となる。

本解析では，時間発展問題のある 1 ステップをベンチマーク問題とする。本解析のような辺要素を用いた電磁場解析では，係数行列は正値性を失っている場合がほとんどあり，ICCG 法をそのまま用いることができない。そこで，加速パラメータ（シフト量）を 1.3 としたシフト付き ICCG 法[12]を用いることとする。

本解析は非構造型の解析であるため，並列化 ICCG 法ソルバにおける行列・ベクトル積演算や前進・後退代入計算では間接アドレッシングが用いられる。ここでは，係数行列の格納形式として，非対角要素については CRS 形式を用い，対角要素は別途 1 次元配列に格納した。また，マルチカラーオーダリング法の実装については未知変数を陽的に並び替える手法を使用した[13]。このとき，前進代入計算の並列化プログラムは図 8 のように与えられる。図中において，icspo(ic)は ic 番目の色の未知変数の開始番号であり，各色毎に代入計算が並列化される。

６．ICCG 法の有限要素解析ベンチマークの数値計算結果

ベンチマークの使用計算機，ICCG 法の収束判定基準は 4．1 節で述べた差分解析ベンチマー

クと同様としている。表 4 に有限要素解析ベンチマークの結果を示す。なお，表中において計算時間は反復の開始から終了までの間の経過時間を示している。まず，色数と反復回数との関係では，40 から 120 程度の色数の変化ではそれほど大きな差はなかった。但し，色数を 500 以上とした場合には改善が見られることが分かっている。次に，1 反復あたりの計算時間については，色数が増大するに従って増加しているが差分解析ほど顕著ではない。これは色数の増加に従ってキャッシュのヒット率が下がることが原因と考えられるが，差分解析と比べて未知変数間のデータ関係が複雑な有限要素解析では元来ランダムアクセスとなっているためにその低下率は小さい。次に並列化効率については全体的に高い値を得ており，マルチカラーオーダリング法の有効性を示している。総合的なベンチマーク性能では，色数 40 の場合が最も計算時間が短かった。

７．おわりに

本稿では，電磁場を対象とした有限要素解析において最もよく用いられる ICCG 法を対象に，差分解析と有限要素解析の 2 種類のベンチマークについて述べた。本解析で扱った疎行列を対象とした計算では，一般的にキャッシュチューニングを効果的に施すことは困難である。これは，データが対象とするモデルに応じてその構造を大きく変える可能性があることや，連続的なデータを扱う場合に比べてブロッキングなどのよく知られたキャッシュチューニングアルゴリズムを適用するのが難しいためである。ただし，本稿で述べたように，並列処理に関連したキャッシュの利用性については注意が必要である。問題のサイズが固定されている場合には，並列処理によって，メモリアクセスの局所化が副次的な作用として得られる可能性があり，並列処理によって一石二鳥の効果が期待できる場合がある。こうした観点からも並列処理については意欲的に取り組む価値があるといえる。また，一般的な応用分野におけるキャッシュチューニングとしては，プログラムをシンプルにかくこと，無駄なメモリを使わないことをあげたい。メモリの階層構造を意識したプログラムは計算機工学の専門家でも手間がかかる仕事である。このような高度なチューニングは困難でも，無駄な作業配列を使わないなどの工夫によりメモリアクセスの局所化の可能性を追求することができる。

7	8	9
4	5	6
1	2	3

図１　3×3 の 2 次元格子における辞書式番号付け

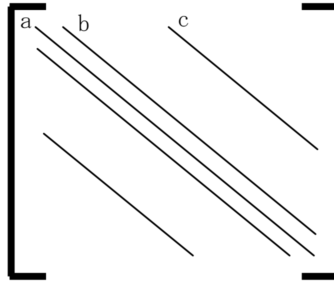


図２　５重対角行列

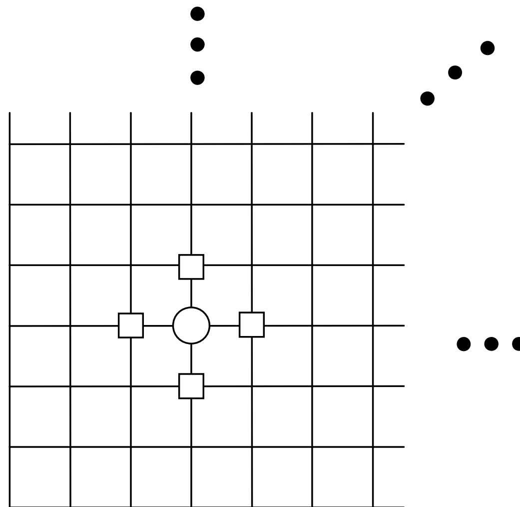


図３　５点差分公式における節点間の関係

```
do  i=1,nx*ny
  z(i)=z(i)-b(i-1)*z(i-1)/diag(i-1)-c(i-nx)*z(i-nx)/diag(i-nx)
enddo
```

図４　逐次型前進代入計算

4	9	5
7	3	8
1	6	2

図５　3×3 の２次元格子における赤－黒番号付け

図６　３色のマルチカラーオーダリングによる係数行列

```
!$OMP PARALLEL PRIVATE(myid,i,icolo)
             :
do icolo=1,icolor-2
  do i=is(myid)+icolo,ie(myid),icolor
    z(i)=z(i)-b(i-1)*z(i-1)/diag(i-1)-c(i-nx)*z(i-nx)/diag(i-nx)
  enddo
  !$OMP BARRIER
enddo
            :
```

図７ マルチカラーオーダリング法による並列化前進代入計算（差分解析）

表１　差分解析ベンチマーク結果

(a) 色数２の場合

CPU 数	計算時間（秒）	反復回数	速度向上
1	248	1600	1.0
2	138	1600	1.80
4	72.3	1599	3.42
8	45.3	1599	5.48
12	27.5	1599	9.03
16	18.9	1600	13.1

(b) 色数４の場合

CPU 数	計算時間（秒）	反復回数	速度向上
1	304	1333	1.0
2	176	1332	1.73
4	92.3	1331	3.29
8	61.3	1331	4.95
12	33.3	1332	9.12

16	20.1	1331	15.1

(c) 色数 8 の場合

CPU 数	計算時間（秒）	反復回数	速度向上
1	444	1176	1.0
2	258	1176	1.72
4	136	1176	3.25
8	95.8	1171	4.61
12	43.7	1171	10.2
16	21.7	1174	20.4

(d) 色数 20 の場合

CPU 数	計算時間（秒）	反復回数	速度向上
1	595	1074	1.0
2	327	1073	1.82
4	185	1070	3.22
8	133	1069	4.47
12	57.8	1068	10.3
16	23.7	1073	25.1

(e) 色数 100 の場合

CPU 数	計算時間（秒）	反復回数	速度向上
1	773	1012	1.0
2	314	1013	2.47
4	98.1	1012	7.88
8	39.0	1012	19.8
12	23.2	1012	33.3
16	14.6	1012	52.9

表 2 一反復あたりの計算時間と色数の関係

色数	2	4	8	20	100
計算時間（ミリ秒）	155	228	378	554	764

表 3 有限要素解析ベンチマーク（3 次元渦電流解析）の諸元

未知変数の総数	1011920
要素数	327680
節点数	342225

```
!$OMP PARALLEL
     :
do icolo=1,icolor-2
!$OMP DO PRIVATE(j,jj)
do i=icspo(ic),icspo(ic+1)-1
  do j=lnrowptr(i),lnrowptr(i+1)-1
    jj=llnt(j)
    z(i)=z(i)-z(jj)*alic(j)/adic(jj)
  enddo
enddo
enddo
         :
```

図 8 マルチカラーオーダリング法による並列化前進代入計算（有限要素解析ベンチマーク）

表 4 有限要素解析ベンチマーク結果

(a) 色数 40 の場合

CPU 数	計算時間（秒）	反復回数	速度向上
1	823	496	1.0
2	416	496	1.97
4	212	496	3.88
8	115	496	7.14
12	77.6	496	10.6
16	61.1	496	13.5

(b) 色数 80 の場合

CPU 数	計算時間（秒）	反復回数	速度向上
1	899	509	1.0
2	454	509	1.98
4	228	509	3.95
8	122	509	7.39
12	83.4	509	10.8
16	67.1	509	13.4

(c) 色数 120 の場合

CPU 数	計算時間（秒）	反復回数	速度向上
1	1190	497	1.0
2	592	497	2.01
4	288	497	4.13
8	149	497	7.96

12	101	497	11.7
16	81.6	497	14.6

参　考　文　献

[1] C. Balanis, “Advanced Engineering Electromagnetics”, John Willey & Sons, Hoboken, NJ, 1989.

[2] J. Volakis, A. Cbatterjee, and L. Kempel, “Finite Element Method for Electromagnetics”, IEEE, New York, NY, 1998.

[3] Y. Saad, “Iterative Methods for Sparse Linear Systems”, Second ed., SIAM, Philadelphia, PA, 2003.

[4] J. Meijerink and H. A. van der Vorst, “An Iterative Solution Method for Linear Systems of Which the Coefficient Matrix Is a Symmetric M-matrix,” Mathematics of Computation, 31, (1977), pp. 148-162.

[5] 寒川光, 「RISC 超高速化プログラミング技法」, 共立出版, 1995.

[6] I. S. Duff and G. A. Meurant, ``The Effect of Ordering on Preconditioned Conjugate Gradients'', BIT, 29, (1989), pp.635-657.

[7] T. Washio and K. Hayami, “Overlapped Multicolor MILU Preconditioning,” SIAM Journal on Scientific Computing, 16, (1995), pp. 636-650.

[8] S. Doi and T. Washio, “Ordering Strategies and Related Techniques to Overcome the Trade-off Between Parallelism and Convergence in Incomplete Factorization,” Parallel Computing, 25, (1999), pp. 1995-2014.

[9] 岩下　武史, 島崎　眞昭 ;「同期点の少ない並列化 ICCG 法のためのブロック化赤－黒順序付け」, 情報処理学会論文誌, Vol.43 No. 4, (2002), pp. 893-904.

[10] S. Doi and A. Lichnewsky, “A Graph-Theory Approach for Analyzing the Effects of Ordering on ILU Preconditioning,” INRIA report 1452, (1991).

[11] T. Nakata, N. Takahashi, T. Imai, and K. Muramatsu, “Comparison of Various Methods of Analysis and Finite Elements in 3-D Magnetic Field Analysis,” IEEE Trans. Magn., vol. 27, (1991), pp.4073-4076.

[12] K. Fujiwara, T. Nakata, and H. Fusayasu, “Acceleration of Convergence Characteristic of the ICCG Method,” IEEE Trans. Magn., vol. 27, (1993), pp.1958-1961.

[13] T. Iwashita and M. Shimasaki, “Algebraic Multi-color Ordering for Parallelized ICCG Solver in Finite Element Analyses,” IEEE Trans. Magn., vol. 38, (2002), pp. 429-432.

第３部
共通高速化技術

C 言語による OpenMP 入門

黒 田 久 泰

東京大学情報基盤センター

1. はじめに

OpenMP は非営利団体 OpenMP Architecture Review Board（ARB）によって規定されている業界標準規格です。共有メモリ型並列計算機用のプログラムの並列化を記述するための指示文、ライブラリ関数、環境変数などが規定されています。OpenMP を利用するには、OpenMP に対応したコンパイラが必要となりますが、現在、多くのコンパイラが OpenMP に対応しています。

OpenMP の特長としては、下記のような点が挙げられます。

・並列プログラム（マルチスレッドプログラム）が簡単かつ短いコード量で記述できる。
・異なるシステム間でソースプログラムを共通化できるので移植性が高い。
・逐次プログラムから段階的に並列化していくことが可能である。
・同一のソースプログラムで並列環境と非並列環境を共存できる。
・コンパイラの自動並列化機能に比べてプログラムの高速化を達成し易い。

OpenMP は下記のようにいくつかのバージョンがあります。現在、広く普及しているのは Version 2.0 のものですので、本記事も Version 2.0 をもとに説明します。OpenMP は現在のところ Fortran 言語と C/C++言語で利用できますが、本記事は C 言語（C++言語でも使い方は同じ）を取り扱います。

1997 年 10 月	OpenMP Fortran API 1.0
1998 年 10 月	OpenMP C/C++ API 1.0
1999 年 11 月	OpenMP Fortran API 1.1
2000 年 11 月	OpenMP Fortran API 2.0
2002 年 3 月	OpenMP C/C++ API 2.0
2005 年 5 月	OpenMP Fortran C/C++ API Version 2.5
2007 年 10 月	OpenMP Fortran C/C++ API Version 3.0 Draft

OpenMP では、プラグマ・ディレクティブ（#pragma）と呼ばれるコンパイラへの命令文を用いて記述します。OpenMP をサポートしていないコンパイラでは単にコメント行とみなされます。

なお、複数ノードにまたがる並列プログラムは OpenMP では記述できません。MPI のような通信ライブラリを用いた並列プログラミングが必要となります。しかしながら、1 ノードが複数プロセッサで構成されているような並列計算機であればノード内を OpenMP で並列化を行い、ノード間を MPI で並列化するといった方法で高性能なアプリケーション開発を行うことができます。

OpenMP について詳しく知りたい方は OpenMP のホームページ（http://www.openmp.org/）をご覧ください。

2. OpenMP におけるキーワード一覧

OpenMP の全体像を理解するために、指示文、指示節、実行時ライブラリ関数、環境変数にそれぞれどうようなものがあるのかを最初に示します。各詳細については第 4 章以降で説明します。

2.1 OpenMP の指示文

OpenMP の指示文は、プログラム内で並列化を行う場所に挿入して並列化の方法を指定します。プラグマ（#pragma）によって記述され、必ず「#pragma omp ・・・」のような形をとります。

並列リージョン指示文	同期に関する指示文
#pragma omp parallel	#pragma omp single
処理分散指示文	#pragma omp master
#pragma omp for	#pragma omp critical
#pragma omp sections	#pragma omp atomic
	#pragma omp barrier
	#pragma omp ordered
	#pragma omp flush
結合指示文（並列リージョン指示文と処理分散指示文を結合したもの）	
#pragma omp parallel for	
#pragma omp parallel sections	
データ属性指示文	
#pragma omp threadprivate	

「#pragma omp sections」指示文は特別に次のような宣言子を利用します。

section 宣言子
#pragma omp section

2.2 OpenMP の指示節

OpenMP の指示節は必ず OpenMP の指示文とともに使われ、「#pragma omp ・・・ 指示節」のような形をとります。

スコープ指示節	その他の指示節
private	ordered
firstprivate	schedule
lastprivate	if
shared	num_threads
default	nowait
reduction	
copyin	
copyprivate	

2.3　実行時ライブラリ関数

OpenMP では指示文以外にも役に立つ実行時ライブラリ関数が提供されています。これらの関数を一切使わなくても並列化は行えますが、より高度な並列化を行う際に利用します。実行時ライブラリ関数を利用する場合には、プログラムの先頭部分に「#include <omp.h>」を記述して OpenMP 用のヘッダファイル omp.h を読み込む必要があります。

実行環境ルーチン	ロックルーチン
omp_set_num_threads(int)	omp_init_lock(omp_lock_t *)
omp_grt_num_threads()	omp_destroy_lock(omp_lock_t *)
omp_get_max_threads()	omp_set_lock(omp_lock_t *)
omp_get_thread_num()	omp_unset_lock(omp_lock_t *)
omp_get_num_procs()	omp_test_lock(omp_lock_t *)
omp_in_parallel()	ネスト可能なロックルーチン
omp_set_dynamic(int)	omp_init_nest_lock(omp_nest_lock_t *)
omp_get_dynamic()	omp_destroy_nest_lock(omp_nest_lock_t *)
omp_set_nested(int)	omp_set_nest_lock(omp_nest_lock_t *)
omp_get_nested()	omp_unset_nest_lock(omp_nest_lock_t *)
時間計測ルーチン	omp_test_nest_lock(omp_nest_lock_t *)
omp_get_wtime()	
omp_get_wtick()	

OpenMP のデータ型（OpenMP で定義されているデータ型は下記の 2 つです）

omp_lock_t	ロック情報を格納する型（ロックルーチンで使用）
omp_nest_lock_t	ロック情報を格納する型（ネスト可能なロックルーチンで使用）

2.4　環境変数

プログラムを実行する際に OpenMP で定義されている環境変数を設定することで、プログラムで使用する際のスレッド数などを指定することができます。

実行時の動作環境に関するもの
OMP_NUM_THREADS
OMP_SCHEDULE
OMP_DYNAMIC
OMP_NESTED
OMP_WAIT_POLICY（OpenMP 3.0 の機能）
OMP_STACK_SIZE　（OpenMP 3.0 の機能）

3. コンパイルと実行方法

3.1 コンパイラとコンパイルオプション

OpenMP に対応しているコンパイラとよく使われるコンパイルオプションを示します。プログラムの実行がうまくいかない場合には、最適化のレベルを下げる必要があります。

コンパイラ	コマンド名	オプションの説明	推奨する最適化
gcc	gcc (*1)	man gcc	-O3
日立コンパイラ	cc	man cc	-Os +Op
Intel コンパイラ	icc	icc -help	-fast
PGI コンパイラ	pgcc	pgcc -help	-fastsse -O4 (*2)
Sun Studio	cc	cc -flags	-fast
Visual Studio	cl.exe	cl.exe /help	/Ox
IBM XLC	xlc	xlc -help	-O5
PathScale	pathcc	pathcc -help	-O3

(*1) gcc はバージョン 4.1 から OpenMP に対応。コマンド名が gcc41 や gcc42 の場合もある。
(*2) PGI コンパイラで-fastsse だけだと-O2 が設定されるため、-O3 や-O4 を追加で設定する。

OpenMP の規格ではコンパイルオプションの記述方法までは規定されていません。そのためコンパイラごとに OpenMP を有効にしてコンパイルする方法が異なります。OpenMP が使われているプログラムを、OpenMP 並列化のみ、OpenMP を無効にして自動並列化のみ、OpenMP 並列化と自動並列化の両方を行う、のそれぞれの場合のコンパイルオプションを示します。

コンパイラ	OpenMP 並列化のみ	自動並列化のみ	OpenMP＋自動並列化
gcc	-fopenmp -lgomp	なし	なし
日立コンパイラ	-parallel -omp	-parallel	なし
Intel コンパイラ	-openmp	-parallel	-openmp -parallel
PGI コンパイラ	-mp	-Mconcur	-mp -Mconcur
Sun Studio	-xopenmp=parallel	-xautopar	-xopenmp=parallel -xautopar
Visual Studio	/openmp	なし	なし
IBM XLC	-qsmp=omp	なし(*1)	-qsmp
PathScale	-mp	-apo	-mp -apo

(*1) -qsmp=noomp だと OpenMP と自動並列化の両方を有効にします。OpenMP だけを無効にするオプションはありません。

詳細については、各コンパイラのマニュアルをご覧ください。

3.2 実行方法

プログラムを実行する前に、環境変数 OMP_NUM_THREADS に並列スレッド数を設定します。並列スレッド数を 16 に指定する場合は下記のようにします。あとは普通に実行するだけです。

シェル	環境変数の設定方法
sh	OMP_NUM_THREADS=16 export OMP_NUM_THREADS
csh と tcsh	setenv OMP_NUM_THREADS 16
bash と Linux や FreeBSD 上の sh	export OMP_NUM_THREADS=16
Windows	set OMP_NUM_THREADS=16

4　OpenMP の指示文

4.1　#pragma omp parallel

「#pragma omp parallel」の次の 1 文またはブロック（ { から } の部分）が並列に実行されます。並列に実行される区間を並列リージョンと呼びます。

```
#include <stdio.h>
int main(void)
{
  #pragma omp parallel
  {
    printf("Hello World!¥n");
  }
}
```

上記のプログラムの場合、「Hello World!」が実行スレッドの数だけ表示されます。OpenMP ならこれだけのコード量で並列化が実現できます。

4.2　#pragma omp for

for 文を並列化します。並列リージョン内で指定する必要があります。
int 型の配列 a[100]の全要素を 0 に初期化するプログラムを並列化するには下記のようにします。

```
int main(void)
{
  int i, a[100];
  #pragma omp parallel
  {
    #pragma omp for
    for(i=0;i<100;i++){
      a[i]=0;
    }
  }
}
```

```
int main(void)
{
  int i, a[100];
  #pragma omp parallel for
  for(i=0;i<100;i++){
    a[i]=0;
  }
}
```

上記の 2 つのプログラムは基本的に同じ意味です（コンパイラによっては異なるバイナリコードを吐き出すものもあります)。並列リージョンの中に「#pragma omp for」が 1 つしかない場合には、右のプログラムのように結合指示文「#pragma omp parallel for」を使って書くと行数が短くなります。
上記のプログラムの場合、例えば実行スレッド数を 4 にして実行すると、4 つのスレッドが下記のように for ループの処理を分担して実行します。

```
スレッド 0 番：  for(i= 0;i< 25;i++) a[i]=0;
スレッド 1 番：  for(i=25;i< 50;i++) a[i]=0;
スレッド 2 番：  for(i=50;i< 75;i++) a[i]=0;
スレッド 3 番：  for(i=75;i<100;i++) a[i]=0;
```

実行スレッド数は動的に決定されるため、このような分担をプログラム実行時に行うような実行コードがコンパイラによって生成されています。

4.3 #pragma omp sections

「#pragma omp sections」指示文ではブロック内を並列に処理します。必ず宣言子「#pragma omp section」とともに使われます。その宣言子「#pragma omp section」の次の 1 文またはブロックを 1 つのスレッドに割り当てて並列に実行します。なお、「#pragma omp section」の後の 1 文またはブロックをセクションと呼びます。

```
#include <stdio.h>
int main(void)
{
  #pragma omp parallel
  {
    #pragma omp sections
    {
      #pragma omp section
      printf("Hello 1¥n");
      #pragma omp section
      printf("Hello 2¥n");
      #pragma omp section
      {
        printf("Hello 3¥n");
        printf("Hello 3¥n");
      }
    }
  }
}
```

```
#include <stdio.h>
int main(void)
{
  #pragma omp parallel sections
  {
    #pragma omp section
    printf("Hello 1¥n");
    #pragma omp section
    printf("Hello 2¥n");
    #pragma omp section
    {
      printf("Hello 3¥n");
      printf("Hello 3¥n");
    }
  }
}
```

あるスレッドが「Hello 1」と表示、別のスレッドが「Hello 2」と表示、さらに別のスレッドが「Hello 3」を 2 回表示します。実行スレッド数が「#pragma omp section」宣言子で指定したセクションの数より少ない場合には、早く処理の終わったスレッドがまだ実行が開始されていないセクションを実行していきます。逆に実行スレッド数が「#pragma omp section」宣言子で指定したセクションの数より多い場合には、仕事を一切行わないスレッドが出てくることになります。

最初のセクションの始まりの「#pragma omp section」宣言子は記述を省略することができます。

「#pragma omp section」宣言子には指示節を付けることはできません。

「#pragma omp sections」指示文で指定したブロックの出口では暗黙のバリア（全てのスレッドがその場所に到達するまで待機する）があります。そのため、ブロック以降の処理を開始する段階で、全てのセクションの実行が終えていることが保証されています。ただし、これは「#pragma omp sections」指示文に nowait 指示節が指定されていない場合に限ります。

並列リージョンの中に指示文が「#pragma omp sections」指示文の 1 つしかない場合には、右のプログラムのように結合指示文「#pragma omp parallel sections」を使って書くと行数が短くなります。

セクション部分では、当然ながら関数呼び出しを行うことも可能です。そうすると、各スレッドで全く独立した処理を実行するようなことも可能となります。

4.4 #pragma omp single

1 スレッドだけが実行するブロックであることを指定します。どのスレッドが実行するかは決まっていません。

```
#include <stdio.h>
int main(void)
{
  #pragma omp parallel
  {
    printf("Hello 1¥n");
    #pragma omp single
    {
       printf("Hello 2¥n");
    }
    printf("Hello 3¥n");
  }
}
```

「Hello 1」と「Hello 3」は実行スレッドの数だけ表示されますが、「Hello 2」は 1 回だけ表示されます。

「#pragma omp single」指示文の出口では暗黙のバリアがあります。そのため、「Hello 1」と「Hello 2」の表示が全て行われたあとに、「Hello 3」が表示されます。逆に入口では暗黙のバリアがないため、「Hello 1」が全て表示される前に「Hello 2」が表示されることがあります。

4.5 #pragma omp master

マスタースレッド(0 番スレッド)だけが実行することを指定します。

```
#include <stdio.h>
int main(void)
{
  #pragma omp parallel
  {
    printf("Hello 1¥n");
    #pragma omp master
    {
       printf("I am a master¥n");
    }
    printf("Hello 2¥n");
  }
}
```

「Hello 1」と「Hello 2」は実行スレッドの数だけ表示されます。「I am a master」はマスタースレッドだけが表示します。一般には、マスタースレッドに限定するよりは「#pragma omp single」指示文を使ってどのスレッドが実行してもいいようにした方が効率は良くなります。

「#pragma omp master」指示文では、指定されたブロックの入口と出口で暗黙のバリアは存在しません。「I am a master」が表示される前に「Hello 2」が表示されることがあります。その点は「#pragma omp single」指示文とは異なっているので注意が必要です。

4.6 #pragma omp critical

直後の 1 文またはブロックの実行を一度に 1 つのスレッドに制限します。このような領域をクリティカル領域といいます。全てのスレッドが実行を行うという点に注意してください。

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{
  #pragma omp parallel
  {
    #pragma omp critical (name)
    {
      sleep(1);
      printf("sleep end¥n");
    }
  }
}
```

上記のプログラムでは、各スレッドが sleep(1)で 1 秒待機した後、「sleep end」と表示します。一度に 1 つのスレッドに制限しているので、結果的に、1 秒おきに「sleep end」が表示されます（実行スレッドの数だけ）。

オプションの name は、クリティカル領域を識別するのに使用します。スレッドは、同じ名前の クリティカル領域を他のどのスレッドも実行していない状態になるまで、クリティカル領域の入り口で待機します。名前のないクリティカル領域は全て同一のクリティカル領域として扱われます。

4.7 #pragma omp atomic

直後の 1 文をアトミック命令（複数のスレッドが衝突せずに安全に共有変数の値を更新する）として実行することを指定します。ブロックを指定することはできません。

```
#include <stdio.h>
int main(void)
{
  int i=0;
  #pragma omp parallel
  {
    #pragma omp atomic
    i++;
    printf("i=%d¥n",i);
  }
}
```

「#pragma omp atomic」宣言文の後に続く 1 文は下記のものだけに限定されています。x はスカラ変数、値には x を参照しない一般の式を記述できます。

```
x++     ++x     x--     --x     x+=値    x-=値
x*=値   x/=値   x&=値   x^=値   x|=値    x<<=値   x>>=値
```

「#pragma omp atomic」指示文はいつでも「#pragma omp critical」指示文に置き換えることが可能です。いかし、「#pragma omp atomic」指示文を用いると一般にハードウェア命令による値の更新を行うため「#pragma omp critical」指示文を使うよりも効率が良くなります。

4.8 #pragma omp barrier

同時に実行されている全てのスレッドの同期を取ります。

```
#include <stdio.h>
int main(void)
{
  #pragma omp parallel
  {
    printf("Hello 1¥n");
    #pragma omp barrier
    printf("Hello 2¥n");
  }
}
```

上記の場合、全てのスレッドが「Hello 1」を表示し終わってから、「Hello 2」が表示されます。

共有変数の値の参照・更新、プログラム内部での時間計測、出力を順番通りに行いたい場合などに利用します。また、プログラムのデバッグの際にもよく使われます。ただし 「#pragma omp barrier」指示文をたくさん入れてしまうと速度低下の一因になりますので、利用は最小限に留めましょう。

4.9 #pragma omp ordered

直後の 1 文またはブロックを for ループが逐次実行された場合の順番で実行します。これは、「#pragma omp for」指示文または「#pragma omp parallel for」指示文のブロック内で指定する必要があります。また、それらの指示文には ordered 指示節を付ける必要があります。

```
#include <stdio.h>
#include <omp.h>
int main(void)
{
  int i, a[100];
  #pragma omp parallel for ordered
  for(i=0;i<100;i++){
    a[i]=0;
    #pragma omp ordered
    printf(" i=%d thread_num=%d¥n",i,omp_get_thread_num());
  }
}
```

上のプログラムを実行スレッド数 4 で実行すると下記のように表示されます。

i=0 thread_num=0
i=1 thread_num=0
・・・
i=24 thread_num=0
i=25 thread_num=1
・・・
i=49 thread_num=1
i=50 thread_num=2
・・・
i=74 thread_num=2
i=75 thread_num=3
・・・
i=99 thread_num=3

この場合、i=0～24、i=25～49、i=50～74、i=75～99 の各ループは並列に実行されていないため、全くマルチプロセッサを有効に利用できておらず速度向上にはなりません。通常は、5.8 節で説明する schedule 指示節を指定して、for ループをサイクリックに並列処理するなどの指定が必要となります。

4.10　#pragma omp flush

実行中のスレッド間で共有変数や配列要素などの値の一貫性は通常保証されていません。これは、最適化などによりレジスタ内で値が保持されている間はメモリに書き込まれないことがあるからです。「#pragma omp flush」指示文を使うことで、明示的にレジスタに保持されている値をメモリに書き出すことができます。

ただし、下記の場所では、自動的に flush が行われ、共有変数や配列要素などのメモリの一貫性が保たれます。

```
#pragma omp parallel の入口と出口
#pragma omp for と #pragma omp parallel for の出口
#pragma omp sections と #pragma omp parallel sections の出口
#pragma omp single の出口
#pragma omp critical の入口と出口
#pragma omp barrier
#pragma omp ordered の入口と出口
```

ただし、nowait 指示節を入れると flush は行われません。

逆に、下記の場所では、flush が行われないので注意が必要です。

```
#pragma omp for と#pragma omp parallel for の入口
#pragma omp sections と #pragma omp parallel sections の入口
#pragma omp single の入口
#pragma omp master の入口と出口
```

4.11　#pragma omp threadprivate(list)

スレッドごとにプライベートで、スレッド内ではグローバルにアクセスできる変数（threadprivate 変数）を宣言します。通常はプログラムのグローバル領域（関数の外）で宣言します。複数の変数を指定する場合には、カンマ（,）で区切ります。

スレッドは他のスレッドの threadprivate 変数を参照することはできません。プログラムの逐次実行領域では、マスタースレッドの持つ値を参考することになります。

threadprivate 変数の初期値は、宣言した変数の初期値と等しくなります。

「#pragma omp parallel」指示文に copyin 指示節を指定することでマスタースレッドの値が全てのスレッドの値としてコピーされます。

threadprivate 変数は、copyin 指示節、 schedule 指示節、if 指示節に指定することができますが、private 指示節、firstprivate 指示節、 lastprivate 指示節、shared 指示節、reduction 指示節で指定することはできません。また、threadprivate 変数については default 指示節は適用されません。

「#pragma omp threadprivate」指示文の使い方の例を示します。

```
 1: #include <stdio.h>
 2: #include <omp.h>
 3:
 4: int i=100;
 5: #pragma omp threadprivate(i)
 6:
 7: int main()
 8: {
 9:   i=200;
10:   #pragma omp parallel
11:   printf("thread_num=%d i=%d¥n",omp_get_thread_num(),i);
12:
13:   i=1000;
14:   #pragma omp parallel copyin(i)
15:   {
16:     i+=omp_get_thread_num();
17:     printf("thread_num=%d i=%d¥n",omp_get_thread_num(),i);
18:   }
19: }
```

5 行目で int 変数である i を threadprivate 変数にしています。4 行目に i の初期値として 100 を代入しているため各スレッドの参照する i の値も全て 100 が初期値として代入されます。9 行目に i=200 としておりこれはマスタースレッドの保持する i のみが 200 になります。マスタースレッド以外のスレッドの保持している値は 100 のままです。したがって、11 行目の printf()で表示される結果は次のようになります。

```
thread_num=0 i=200
thread_num=1 i=100
thread_num=2 i=100
thread_num=3 i=100
```

次に、13 行目で i=1000 とし、14 行目では copyin(i)を指定しているためマスタースレッドの保持する i の値である 1000 が全てのスレッドの threadprivate 変数の i にコピーされます。16 行目で各スレッドの番号を threadprivate 変数の i に加えています。その結果、17 行目の printf()で出力される結果は次のようになります。

```
thread_num=0 i=1000
thread_num=1 i=1001
thread_num=2 i=1002
thread_num=3 i=1003
```

5　OpenMP の指示節

5.1　private(list)

list に指定された変数が各スレッドでプライベートであることを宣言します。複数の変数を指定する場合には、カンマ（,）で区切ります。「#pragma omp for」指示文の対象となる for ループのインデックス変数は自動的にプライベートになります。そのためこのインデックス変数については記述を省略できます。default 指示節を指定しない場合、変数のデフォルトの属性は共有変数（shared）になっていますので、プライベート変数がある場合にはこの private 指示節を使って宣言する必要があります。

5.2　firstprivate(list)

list に指定された変数は private 指示節と同様、プライベートであることを宣言します。private 指示節との違いは、変数の初期値として並列リージョン開始時の変数の値が各スレッドのプライベート変数にコピーされるという点です。並列リージョン内でプライベート変数の初期値を設定する場合に firstprivate 指示節を用いると効率が悪くなります。

5.3　lastprivate(list)

list に指定された変数は private 指示節と同様、プライベートであることを宣言します。private 指示節との違いは次の点です。「#pragma omp for」指示文で指定された場合には、for ループの最後の繰り返しを実行したスレッドの持つプライベート変数の値が元の変数の値に代入されます。「#pragma omp sections」指示文で指定された場合には、最後のセクションを実行したスレッドの持つプライベート変数の値が元の変数の値に代入されます。

5.4　shared(list)

list に指定された変数が各スレッドで共有変数であることを宣言します。default 指示節を指定しない場合、デフォルトの属性が共有変数（shared）ですので、shared 指示節を使う必要はありません。

5.5　default(shared | none)

並列リージョン内の全ての変数に対してデフォルトの属性を与えます。

default(shared)を指定した場合、デフォルトを共有変数とします。default(none)を指定すると、デフォルトの属性を与えません。この場合、変数は private、shared、firstprivate、lastprivate、reduction のどれかの指示節で指定されていなくてはなりません。なお、default 指示節を指定しなかった場合には、default(shared)が設定されているとみなします。

5.6 reduction(operator:list)

list で指定した変数に対して演算子 operator のリダクション演算を行うことを宣言します。リダクション演算とは総和を求めるような計算のことです。operator には、次のいずれかを指定します。

+	*	-	&	^	|	&&	||

list には複数の変数を指定することできますが、この場合、カンマ（,）で区切ります。

「#pragma omp atomic」指示文や「#pragma omp critical」指示文でリダクション演算を記述することもできますが、この reduction 指示節を使える場合には使った方が高速になります。

double 型配列 a[]の n 個の要素の和を求める関数であれば、次のようになります。

```
double sum(double a[], int n)
{
   int i;
   double sum=0.0;
   #pragma omp parallel for reduction(+:sum)
   for(i=0; i<n; i++) {
      sum += a[i];
   }
   return sum;
}
```

一見、「reduction(+:sum)」の部分がなくても正しく動作するように見えるかもしれませんが、複数スレッドで実行すると共有変数 sum には正しい結果が入る保証はありません。共有変数 sum が各スレッド内ではレジスタで処理されたり更新の際にメモリ競合が起こったりするためです。reduction 指示節を用いて、共有変数 sum を宣言しておくと、正しい結果が格納されるようになります。

reduction 指示節は次のように「#pragma omp parallel sections」指示文においても指定できます。

```
double sum(double a[], int n)
{
   int i;
   double sum=0.0;
   #pragma omp parallel sections private(i) reduction(+:sum)
   {
      #pragma omp section
      for(i=0; i<n/2; i++) sum += a[i];
      #pragma omp section
      for(i=n/2;i<n;i++) sum += a[i];
   }
   return sum;
}
```

5.7 ordered

for ループ中に「#pragma omp ordered」指示文が含まれていることを宣言します。

5.8 schedule

for ループにおいてループ反復をどのようにスレッドに割り当てるかを宣言します。schedule(type) または schedule(type,chunk)の形で使用します。type には static、dynamic、guided、runtime のうちのいずれかが入ります。ここで、チャンクとは 1 つのスレッドが処理を行う最小単位であるループの反復回数のことをいいます。

static では静的すなわち反復開始前に各スレッドに対してチャンクが割り当てられます。chunk を指定しなかった場合には、チャンクは総反復回数を実行スレッド数で割った値となります。schedule 指示節を指定しなかった場合のデフォルトは static でチャンク指定なしと同じです。

dynamic では、チャンクは遊んでいるスレッドに対して動的に割り当てられます。すなわち、処理を終了した順に次のチャンク分の反復の処理がスレッドに割り当てられます。chunk を省略すると chunk=1 とみなされます。

guided でも遊んでいるスレッドに対して動的に割り当てられますが、チャンクの値は自動的に決定

されます。この場合、chunk には、分割の際に最小となる反復回数を指定します。chunk を省略すると chunk=1 とみなされます。

runtime だけは他と異なり、環境変数 OMP_SCHEDULE の値を利用することを指定します。この場合、chunk は指定できません。

5.9 copyin

copyin 指示節は threadprivate 指示文で指定された変数に適用できます。copyin 節で指定された変数は、並列リージョンの開始時にマスタースレッド上の threadprivate 変数の値を、各スレッドの threadprivate 変数にコピーします。

5.10 copyprivate(list)

「#pragma omp single」指示文だけで利用できる指示節です。list に指定された変数を他のスレッドにコピーします。

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main()
{
  int i;
  #pragma omp parallel private(i)
  {
    #pragma omp single copyprivate(i)
    i=random();

    printf("thread_num=%d i=%d¥n",omp_get_thread_num(),i);
  }
}
```

1 スレッドだけが random()を実行します。そして、その結果を全てのスレッドのプライベート変数 i にコピーします。

5.11 if

「#pragma omp parallel」指示文を並列実行する場合の条件を記述します。条件が成立しない場合には逐次実行されます。次の場合であれば、変数 n の値が 100 以上の場合に並列実行を行います。

```
#pragma omp parallel for if(n>=100)
{
  . . .
```

5.12 num_threads

実行スレッド数を指定します。環境変数 OMP_NUM_THREADS よりも優先されます。ただし、システムで上限が決められている場合にはそれを超える値を指定すると実行時にエラーになります。

```
#pragma omp parallel num_threads(4)
{
  . . .
```

5.13 nowait

処理が終了したスレッドは他のスレッドの状況に関係なく、次の処理に移ってもよいことを宣言します。

5.14 指示文と指示節の可能な組み合わせ

指示文と指示節の可能な組み合わせをまとめると次の表のようになります。

	parallel 指示文	for 指示文	sections 指示文	single 指示文	parallel for 指示文	parallel sections 指示文
private	●	●	●	●	●	●
firstprivate	●	●	●	●	●	●
lastprivate		●	●		●	●
shared	●				●	●
default	●				●	●
reduction	●	●	●		●	●
ordered		●			●	
schedule		●			●	
copyin	●				●	●
copyprivate				●		
if	●				●	●
num_threads	●				●	●
nowait		●	●	●		

6 OpenMP の実行時ライブラリ関数

6.1 ヘッダファイルについて

OpenMP の実行時ライブラリ関数を利用するには、ヘッダファイル omp.h をインクルードします。

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

このままだと OpenMP 非対応コンパイラや OpenMP を無効にしてコンパイルしようとするとインクルードファイルがないというエラーになってしまいます。それを避けるには_OPENMP がマクロ定義されているかどうかを利用します。OpenMP の規格では、OpenMP を有効にしてコンパイルすると、OpenMP Version 1.0 の仕様に基づいている場合には「_OPENMP =199810」が OpenMP Version 2.0 の仕様に基づいている場合には「_OPENMP =200505」がマクロ定義されています。

同一のソースプログラムで OpenMP 対応と非対応の両方を記述するには、次のプログラムのように_OPENMP がマクロ定義されているかどうかを利用します。

```
#ifdef _OPENMP
#include <omp.h>
#endif
```

6.2 実行環境ルーチン

実行環境の設定、現在の実行状態を参照するためのルーチンには下記のものがあります。

void **omp_set_num_threads(int);**

次の並列リージョン開始時に起動されるスレッド数を指定します。システムで上限が決められている場合にはそれを超える値を指定すると実行時にエラーになります。

int **omp_get_num_threads();**

現在、起動しているスレッドの数を返します。

int **omp_get_max_threads();**

スレッドの最大生成数を返します。

int **omp_get_thread_num();**

自分のスレッド番号を得ます。スレッド番号は 0 から p-1（スレッド数を p とする）の値になります。

int **omp_get_num_procs();**

プログラムで使用可能なプロセッサの数を返します。

int **omp_in_parallel();**

並列起動が行われている場合は 0 以外の値を返します。そうでない場合は 0 を返します。

void **omp_set_dynamic(int);**

スレッド数の動的調整機能を有効にする場合は 0 以外の値、無効にする場合は 0 を指定します。

int **omp_get_dynamic();**

スレッド数の動的調整機能が有効である場合は 0 以外の値、無効である場合には 0 を返します。

void **omp_set_nested(int);**

並列のネストを有効にする場合は 0 以外の値、無効にする場合は 0 を指定します。

int **omp_get_nested();**

並列のネストが有効である場合は 0 以外の値、無効である場合には 0 を返します。

6.3 ロックルーチン

OpenMP ではロック機構として単純ロックとネスト可能なロックの 2 種類が用意されています。これらを利用するために使うルーチンです。

void **omp_init_lock(&lock);**

ロック変数を初期化します。lock は「omp_lock_t lock」のように宣言された変数です。

void **omp_destory_lock(&lock);**

ロック変数を破棄します。

void **omp_set_lock(&lock);**

ロックの所有権を得るまで待機し、ロックの所有権を得ると処理が戻ります。ロックの所有権を得たスレッド自身が omp_unset_lock()で明示的にロックの所有権を解放するまで他のスレッドはロックの所有権を得ることはできません。

void **omp_unset_lock(&lock);**

ロックの所有権を解放します。これにより他のスレッドがロックの所有権を得ることができるよう

になります。

int omp_test_lock(&lock);

ロックの所有権を得ることを試みます。ロックの所有権を得ると 1 を返し、そうでない場合には 0 を返します。ロックの所有権が得られるまで待機しないという点が omp_set_lock()との違いです。

void omp_init_nest_lock(&lock);

ネスト可能なロック変数を初期化します。ネストカウンタ（所有権を持っているスレッドがロックを行った回数のこと）も 0 に設定されます。lock は「omp_nest_lock_t lock」のように宣言された変数です。ネスト可能なロックでは、同じスレッドによって複数回ロックすることができます。

void omp_destory_nest_lock(&lock);

ネスト可能なロック変数を破棄します。

void omp_set_nest_lock(&lock);

すでに同じスレッドによって所有権を獲得している場合にはネストカウンタを 1 増やして処理が戻ります。そうでない場合には、ネスト可能なロックの所有権を得るまで待機し、ロックの所有権を得ると処理が戻ります。

void omp_unset_nest_lock(&lock);

ネスト可能なロックのネストカウンタを 1 だけ減らします。その結果ネストカウンタが 0 になった場合にはロックの所有権を解放します。これにより他のスレッドがロックの所有権を得ることができるようになります。

int omp_test_nest_lock(&lock);

すでに同じスレッドによって所有権を獲得している場合にはネストカウンタを 1 増やしてその値を返します。そうでない場合には、ネスト可能なロックの所有権を得ることを試みます。ロックの所有権を得ると 1 を返し、そうでない場合には 0 を返します。ロックの所有権が得られるまで待機しないという点が omp_set_nest_lock()との違いです。

6.4 時間計測ルーチン

double omp_get_wtime();

1970 年 1 月 1 日午前 0 時からの経過秒数、あるいは、システムを起動してからの経過秒数を倍精度実数で返します。経過時間の測定では、計測開始時点と終了時点の 2 個所でこの関数を呼び出し、その差を経過時間とします。

double omp_get_wtick();

omp_get_wtime()の返す値の刻み幅を倍精度実数で返します。

7　OpenMP の環境変数

環境変数は全て大文字ですが、環境変数に設定する値については大文字と小文字の区別はありませんので、どちらで指定しても構いません。

7.1　OMP_NUM_THREADS

使用するスレッド数を指定します。

通常は、物理プロセッサの数を超える値を指定することができますが、システムによっては上限が決められていることもあります。また omp_set_num_threads() ライブラリルーチンを呼び出すか、「#pragma omp parallel」指示文で num_threads 指示節を使って明示的にスレッド数を指定している場合には、この変更後の値が優先されます。

7.2　OMP_SCHEDULE

「#pragma omp for」指示文と「#pragma omp parallel for」指示文において、schedule(runtime) 指示節を指定した場合のスケジューリング方法を指定します。type または type,chunk という値を指定します。type は STATIC、DYNAMIC、GUIDED のいずれかであり、chunk はチャンクの大きさを指定します。

例：setenv OMP_SCHEDULE STATIC,5
　　setenv OMP_SCHEDULE DYNAMIC,5
　　setenv OMP_SCHEDULE GUIDED

7.3　OMP_DYNAMIC

スレッド数の動的調整機能（システムの負荷によって実行スレッド数を変更する機能）を有効にする場合は TRUE、無効にする場合は FALSE を指定します。デフォルト値は実装依存とされているため、システムによって異なります。決まった数のスレッドを必要とする場合には、TRUE に設定する必要があります。PGI などの一部のコンパイラでは利用できません。

7.4　OMP_NESTED

並列のネストを有効にする場合は TRUE、無効にする場合は FALSE を指定します。デフォルト値は FALSE になっています。PGI などの一部のコンパイラでは利用できません。

7.5　OMP_WAIT_POLICY　（OpenMP 3.0 の機能）

スレッドの待機中の挙動を指定します。スピンループして待機する場合には ACTIVE（デフォルト）、スリープして待機する場合には PASSIVE を指定します。

7.6　OMP_STACK_SIZE（OpenMP 3.0 の機能）

各スレッドが生成されるときのスタックサイズを指定します。

例：setenv OMP_STACK_SIZE 2K
　　setenv OMP_STACK_SIZE 2M
　　setenv OMP_STACK_SIZE 2G
　　setenv OMP_STACK_SIZE 256B

K はキロバイト、M はメガバイト、G はギガバイト、B はバイトを示します。これらを省略した場合には、K が指定されたものとみなします。

8　サンプルプログラム

8.1 マルチ ping プログラム

192.168.0.1～192.168.0.254 のように連続した 254 個の IP アドレスに対して一斉に ping コマンドを実行するプログラムです。ping コマンドで -t オプションでタイムアウトの時間を指定できますが、タイムアウトを 1 秒に設定したとしても最大で 254 秒かかることになります。OpenMP で並列化して、実行スレッド数を 254 に設定すると 2 秒ほどで終わります。

使い方は、192.168.0.1～192.168.0.254 を調査したい場合には、次のように実行します。

```
% ./multiping 192.168.0
```

ping の応答があった場合には、「192.168.0.1 is up. time=0.980 ms」のように表示されます。

```
 1: #include <stdio.h>
 2: #include <string.h>
 3:
 4: int main(int argc,char *argv[])
 5: {
 6:   int i;
 7:   if( argc!=2 ){
 8:     printf("usage : multiping 192.168.0¥n");
 9:     return 1;
10:   }
11:
12:   #pragma omp parallel for schedule(dynamic) ordered
13:   for(i=1;i<255;i++){
14:     char command[64],output[1024];
15:     FILE *in_pipe;
16:     int outputsize;
17:     sprintf(command,"ping -t 1 %s.%d",argv[1],i);
18:     in_pipe=popen(command,"r");
19:     outputsize=fread(output,1,1024,in_pipe);
20:     output[outputsize]='¥0';
21:     pclose(in_pipe);
22:
23:     #pragma omp ordered   // IP アドレス順に表示するために挿入
24:     {
25:       int k=0;
26:       char *tmp;
27:       if((tmp=strstr(output,"time="))!=NULL){
28:         while(tmp[k]!='¥n') k++;
29:         tmp[k]='¥0';
30:         printf("%s.%d is up. %s¥n",argv[1],i,tmp);
31:       }else{
32:         printf("%s.%d is down.¥n",argv[1],i);
33:       }
34:     }
35:   }
36:   return 0;
37: }
```

上記のプログラムでは、プロセッサの個数が 1 個でも実行スレッド数を増やすことで実行時間が短くなります。このようにプロセッサの個数が 1 個であっても高速化が実現できることもあります。

8.2 ファイルコピープログラム

2つのスレッドを用いてファイルコピーを行うプログラムです。1つのスレッドはファイルの読み込みを担当し、もう1つのスレッドは書き込みを担当します。

使い方は、次のようになります。

% ./filecopy 元のファイル名 コピー先のファイル名

同じディスク上でファイルをコピーする場合には、かえって遅くなることがあります。

```
 1: #include <stdio.h>
 2: #include <fcntl.h>
 3: #include <sys/stat.h>
 4: #include <sys/types.h>
 5: #include <sys/uio.h>
 6: #include <unistd.h>
 7: #define BUF_SIZE 4096*1024
 8: char buf1[BUF_SIZE];
 9: char buf2[BUF_SIZE];
10:
11: int main(int argc,char *argv[])
12: {
13:   int file_src,file_dst;
14:   int size1,size2;
15:   if( argc!=3 ){
16:     printf("usage : filecopy source-file dest-file¥n");
17:     return 1;
18:   }
19:   file_src=open(argv[1],O_RDONLY);
20:   file_dst=open(argv[2],O_WRONLY|O_CREAT|O_TRUNC,S_IREAD|S_IWRITE);
21:   if( file_src==-1 || file_dst==-1 ){
22:     printf("file read/write error¥n");
23:     return 1;
24:   }
25:   size1=read(file_src,buf1,BUF_SIZE);
26:
27:   #pragma omp parallel sections num_threads(2)
28:   {
29:     #pragma omp section
30:     while(1){
31:       size2=read(file_src,buf2,BUF_SIZE);
32:       #pragma omp barrier
33:       if( size2<=0 ) break;
34:       size1=read(file_src,buf1,BUF_SIZE);
35:       #pragma omp barrier
36:       if( size1<=0 ) break;
37:     }
38:     #pragma omp section
39:     while(1){
40:       write(file_dst,buf1,size1);
41:       #pragma omp barrier
42:       if( size2<=0 ) break;
43:       write(file_dst,buf2,size2);
44:       #pragma omp barrier
45:       if( size1<=0 ) break;
46:     }
47:   }
48:   close(file_src);
49:   close(file_dst);
50:   return 0;
51: }
```

C言語によるMPIプログラミング入門

片桐　孝洋

東京大学情報基盤センター

１．はじめに

本稿は、東京大学工学部・工学系研究科の共通科目として平成19年度から夏・冬学期に開講している「スパコンプログラミング（1）および（I）」の講義資料を基にして、C言語によりMPI のプログラムを行う並列プログラム初心者に対する参考資料として構成・執筆しました。基本的な概念、用語、使い方の説明を重点的に行っております。また、MPI を用いた並列プログラミングのコツをつかむための２、３の実例を載せてあります。スパコンを用いて並列化をおこなうユーザの参考資料になることを期待しています。

２．並列プログラミングとは

並列プログラミングは、高速化手段の一つです。この高速化は、処理対象を並列化し、並列処理を記述できる計算機言語でプログラミングをし、さらにその並列プログラムを複数の要素計算機(Processing Element, PE)で構成された並列計算機上で実行することで達成されます。簡単にいうと、1つの計算機で1時間かかる処理を、n台の計算機を用いることで1/n時間で処理をすることを狙う技術です。

ところが実際は、単純に1/n時間となりません。これは処理対象の逐次部分の顕著化、利用する並列計算用ソフトウエア(たとえば、本稿で学習する通信ライブラリ MPI)のオーバヘッドなど、さまざまな並列化を阻害する要因が生じてしまうからです。なるべく1/n時間となるように、アルゴリズムや実装上の工夫をすることが効率の良い並列プログラム(並列アルゴリズム)を開発するために重要となります。効率の良い並列プログラムを開発するためには、処理すべき対象の性質、利用する計算機言語の性質、および実行する並列計算機環境(ソフトウエア、ハードウエアの両面)の知識が必要になります。

効率の良い並列プログラム開発のための技術は興味深いものですが、ここでは紙面の都合上、あまり述べることができません。そこで並列プログラミングの初心者が、最低限の並列プログラミングができるようになる程度の知識のみに限定し解説します。

２．１　Flynnの分類

並列計算機の分類として、スタンフォード大学のM.Flynn教授が与えた、**Flynnの分類**という分類が知られています。Flynn の分類では、以下のように、**命令流(Instruction Stream)**と**データ流(Data Stream)**が、**単一(Single)**か**複数(Multiple)**かによって分類分けをしています。

- SISD (Single Instruction stream Single Data stream)：逐次計算機
- SIMD (Single Instruction stream Multiple Data stream)：ベクトルPE
- MISD (Multiple Instruction stream Single Data stream)：例なし
- MIMD (Multiple Instruction stream Multiple Data stream)：マルチPE、PCクラスタ、データフローマシン

本稿で紹介するMPIによる並列プログラムの特性における分類は、SIMD もしくはMIMD の分

類に当たります[1]。しかし MIMD の概念を用いた並列プログラミングは、複雑なため初心者に向きません。

本稿では以降、SIMD の分類(プログラミング・モデル)で並列プログラミングをすることを前提とします。図 1に、4台の計算機を使う場合の SIMD の概念図を載せます。

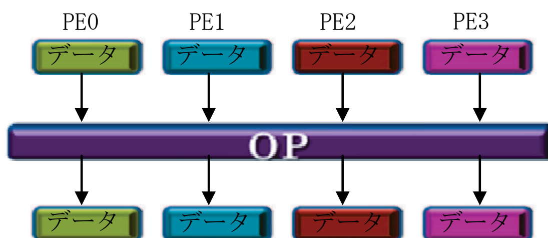


図 1 SIMD の概念

２．２ メモリ構成による分類

並列計算機アーキテクチャを、メモリ構成の違いにより分類する分類法が知られています。詳しくは、本特集号の「オープンスパコンの OS とアーキテクチャの基礎」(松葉浩也著) をご覧ください。

共有メモリ型並列計算機は、メモリを複数の PE が共有している並列計算機です。各 PE から共通のメモリ空間が参照できるため、プログラミングが容易となります。しかしメモリにアクセスするネットワークが複雑化する理由から、PE 数が多い並列計算機は構成できません(**スケーラビリティが低い**)。

分散メモリ型並列計算機は、各 PE にローカルなメモリを所有します。ローカルなメモリを所有するため、メモリ空間は PE 毎に分割され、それゆえ、他の PE が所有するデータをすぐには参照できません。したがって、プログラミングが困難となります。しかし PE 数が多い並列計算機が構成可能です(**スケーラビリティが高い**)。

最後に共有分散メモリ型並列計算機ですが、各 PE が共有分散メモリを所有するものです。共有分散メモリは、物理的には分散メモリなのですが、ハードウエアで論理的に共有メモリ化されているメモリといえます。そのためユーザの観点では、メモリ空間は単一のメモリ空間だとみなせます。それゆえプログラミングが容易となります。分散共有メモリ型並列計算機は、共有メモリ型並列計算機のプログラミングの容易さという長所、分散メモリ型並列計算機の高いスケーラビリティという長所を両方もつ並列計算機といえます。しかし、単一メモリを実現するハードウエア技術(メモリ内容の一貫性保証)が困難であること、またハードウエア実装においてハードウエア量の増加や金銭コストが増大するという問題が生じます[2]。

さて本稿で学習する MPI は、分散メモリ型並列計算機を対象に開発された通信ライブラリです。分散メモリ型並列計算機を対象としていることから、分散メモリ型並列計算機のみでしか

---

[1] 厳密には、MPI による並列プログラムの挙動は MIMD といえます。

[2] ハードウエア量増加の問題を解決するため、ソフトウエアで DSM を実現する研究がなされています(ソフトウエア DSM)。しかしソフトウエア DSM では、処理の遅さによる性能低下が常に問題となります。

実行できないと思うかもしれません。ところがMPIで記述された並列プログラムは、共有メモリ型並列計算機でも共有分散メモリ型並列計算機でも実行できるのです。この理由は、分散メモリ空間は単一メモリ空間を包含するので、一切変更をしなくても論理的に動作可能であるからです。さらにMPIで記述された並列プログラムは、データの参照が局所化されている理由から、共有分散メモリ型並列計算機用のプログラムより高速である事例が報告されています。

以上のことから、現在高速な並列プログラムを開発したい場合には、プログラミングは少々しにくいけれどもMPIを用いて開発するのがベストであるといえます。

３．MPIによる並列プログラミング入門

３．１　MPIとは

MPI (Message Passing Interface) とは、分散メモリ型並列処理の基本であるメッセージパッシングのライブラリの規格です。並列処理をするために初期は、PVM (Parallel Virtual Machine)、各社の並列計算機用の通信ライブラリなど、多数の通信ライブラリがありました。ところがここ十年あまりで、MPI が世界標準の通信ライブラリとなってしまいました。ものの本によると[1]、MPIが話題にのぼるようになってからPVMがMPIに置き換わるまで、1年ぐらいであったとされます。この理由は、MPI の設計方針のよさや、Linux とその上で動く mpich などのおかげで、広くユーザに支持されたことが大きいとのことです。現在、ほぼすべての並列計算機上で MPIライブラリが動きます。

もう1つの世の中の変化は、クラスタコンピューティング、分散メモリ型並列処理の広まりがあったとされます。共有メモリ型の並列処理には、物理上の限界（メモリバンド幅がとれない）があり、やはり大規模計算を行うには分散メモリ型の並列処理に最終的には到達するわけです。

一方でクラスタシステム上の大規模アプリケーションは広く利用されると考えられています。なぜならハードウエアの構築は（理論上は）際限なくできますし、プログラムの書き方によっては高い性能を引き出せるからです。

したがってアプリケーション開発者は、従来型のプログラミング技術（逐次型、メモリ階層型、もしくはベクトル型のプログラミング技術）に加えて、メッセージパッシングという新しい要素技術を勉強しなくてはならなくなっています。この流れは、時代の要請であるといえます。

３．２　MPIの背景

MPIの背景について、簡単に説明しておきます。現在，普通はMPIといえばMPI-1を指し、ここでの説明もMPI-1に基づいています。この拡張版にMPI-2(1997年7月）があります。MPI-2の規格は、一部MPI-1.2に実装されています。

MPI-2では、動的プロセス生成、単方向通信、外部インタフェース、拡張集団通信、並列I/O, C++ およびFortran90インタフェースなど、非常に広範囲の拡張が行われました。特にMPI-1とMPI-2の大きな違いは、動的にプロセスが生成／消滅ができるので、そのような並列処理に向いているということです。例えば、並列の探索処理がその例です。

MPI-1.2については、それに準拠する実装が mpich 1.2 などすでにいくつかあるのですが、MPI-2 への動きはまだ十分でないといえます。ただ国産メーカのスーパコンピュータ（日立、

富士通、NEC など）には、MPI-2 が実装され利用できる状況にあります。

３．３　MPI が提供する主なインタフェース

MPI の全インタフェース（関数）について説明することは大変ですし、じつはあまり利用しない関数もあることから無駄といえます。したがってここでは、よく利用すると思われる、以下の関数についてのみ説明することとします。

- システム関数
  - MPI_Init 関数
  - MPI_Comm_rank 関数
  - MPI_Comm_size 関数
  - MPI_Finalize 関数
- 1 対 1 通信関数
  - ブロッキング型
    - MPI_Recv 関数
    - MPI_Send 関数
  - ノンブロッキング型
    - MPI_Isend 関数
    - MPI_Wait 関数
- 集団通信関数
  - MPI_Reduce 関数
  - MPI_Allreduce 関数
  - MPI_Barrier 関数
- 1 対全通信関数
  - MPI_Bcast 関数
- 時間計測関数
  - MPI_Wtime 関数
- 通信対象分割関数
  - MPI_Comm_split 関数

これらの関数の利用法を、次章に示すプログラム例を基に説明していきます。

４．プログラム例

４．１　初期設定など

それでは、具体的に MPI のプログラム例を示すことで、MPI プログラムに慣れていきましょう。

以下の例は、円周率を求めるプログラムを MPI で並列化したものです。

```
< 1>   #include <stdio.h>
< 2>   #include <math.h>
< 3>   #include <mpi.h>
< 4>   void main(int argc, char* argv[]) {
```

```
< 5>      double PI25DT = 3.141592653589793238462643;
< 6>      double mypi, pi, h, sum, x, f, a;
< 7>      int    n, myid, numprocs, i, rc;
< 8>      int    ierr;
< 9>      ierr = MPI_Init(&argc, &argv);
<10>      ierr = MPI_Comm_rank(MPI_COMM_WORLD, &myid);
<11>      ierr = MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
<12>      if ( myid == 0 ) {
<13>         printf("Enter the number of intervals ¥n");
<14>         scanf("%d",&n);
<15>         printf("n=%d ¥n",n);
<16>      }
<17>      ierr = MPI_Bcast(&n, 1, MPI_INT,0,MPI_COMM_WORLD);
<18>      h = 1.0 / n;
<19>      sum  = 0.0;
<20>      for (i = myid+1; i<=n; i+= numprocs) {
<21>         x = h * (i - 0.5);
<22>         sum = sum + 4.0 / (1.0 + x*x);
<23>      }
<24>      mypi = h * sum;
<25>      MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE,
<26>                 MPI_SUM, 0, MPI_COMM_WORLD);
<27>      if (myid == 0) {
<28>        printf("  pi is approximately: %18.16lf Error is: %18.16lf ¥n",
<29>               pi, fabs(pi-PI25DT));
<30>      }
<31>      rc = MPI_Finalize();
<32>   }
```

なおアルゴリズムの詳細はここでは説明しませんが、興味のある方はプログラムを追跡して調べてみることを勧めます。

まずはじめに MPI では、上記のプログラムがすべての PE 上で実行されます。このような実行形態を、**SPMD (Single Program Multiple Data)**とよびます。

図 2 に、このプログラムの処理形態を載せます。図 2 の処理形態では、概述の SIMD の処理形態であることがわかります。

以降、このプログラムについて簡単な説明をします。

まずプログラム中で、<9>行にある関数で MPI の利用を初期化します。また<31>行の関数で、MPI を終了します。したがって全ての MPI プログラムは、MPI_Init 関数で始まり、MPI_Finalize 関数で終了します。

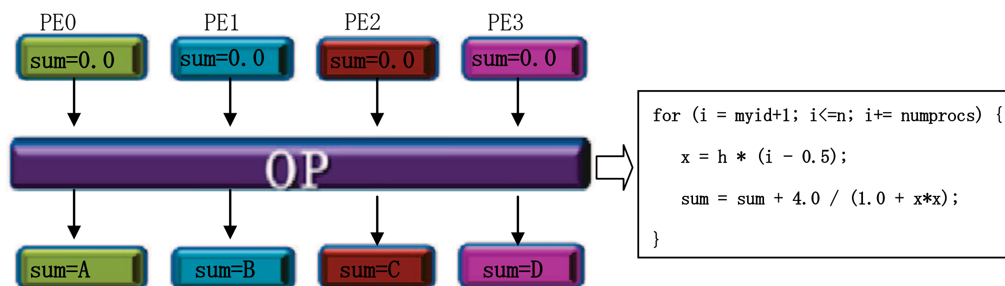


図 2　円周率プログラムの処理形態

<10>行の MPI_Comm_rank 関数では、各 PE に個別に割り付けられた認識番号を得ます。このプログラムでは、その番号は整数変数 myid に保存されます。この認識番号は 0 から始まり PE 台数より一つ少ない数で終ります。この認識番号のことを、MPI では**ランク**とよんでいます。

また MPI_COMM_WORLD は、**コミュニケータ**とよばれる MPI の制御変数です。MPI_COMM_WORLD に通信すべき対象の PE グループが代入されています。

<11>行の MPI_Comm_size 関数では、変数 numprocs に利用する全ての PE 数が代入されます。この値は、各 PE で同一の値となります。

とりあえずここでは、以上の関数の理解のみで終ることにします。

４．２　PE 間での総和演算

多くの並列処理プログラムでは、PE 間で所有している異なるデータに対して、加算演算（総和演算）をしたいことがあります。この PE 間での総和演算は、通信を必要としますが、通信方式が全体の性能に大きく影響します。

まずここでは、素朴な通信方式である各 PE が隣接 PE からのデータを受信した後、加算してその結果を隣接 PE に転送する実装法を示します。

```
<1 >  MPI_Status istatus;
<2 >  int  ierr;
<3 >  int  myid, nprocs;
<4 >  double  dsendbuf, drecvbuf;
<5 >  ierr = MPI_Init(&argc, &argv);
<6 >  ierr = MPI_Comm_rank( MPI_COMM_WORLD, &myid );
<7 >  ierr = MPI_Comm_size( MPI_COMM_WORLD, &nprocs );
<8 >  /* === 各 PE における総和演算したい値。ここでは myid とする。*/
<9 >  dsendbuf = myid;
<10>  printf ("myid:%d, dsendbuf=%4.2lf ¥n", myid, dsendbuf);
<11>  drecvbuf = 0.0;
<12>  if (myid != 0) {
<13>     ierr = MPI_Recv(&drecvbuf, 1, MPI_DOUBLE, myid-1, 0, MPI_COMM_WORLD, &istatus);
<14>  }
```

```
<15>  dsendbuf = dsendbuf + drecvbuf;
<16>  if (myid != nprocs-1) {
<17>    ierr = MPI_Send(&dsendbuf, 1, MPI_DOUBLE, myid+1, 0, MPI_COMM_WORLD);
<18>  }
<19>  if (myid == nprocs-1) printf ("Total = %4.2lf ¥n", dsendbuf);
<20>  ierr = MPI_Finalize();
```

このプログラムでは、行<12>-<14>で各 PE で計算済の部分和との加算を行うために隣接 PE である myid-1 番の PE からの受信を待ちます。MPI 関数 MPI_Recv の各引数の意味は以下の通りです。

```
int MPI_Recv(recvbuf, icount, idatatype, isource, itag, icomm, istatus);
```

- Recvbuf : 受信領域の先頭番地を指定する。
- icount : 整数型。受信領域のデータ要素数を指定する。
- idatatype : 整数型。受信領域のデータの型を指定する。よく用いられるデータ型としては、以下である。
  - MPI_INT (整数型)
  - MPI_REAL (実数型)
  - MPI_DOUBLE (倍精度実数型)
- isource : 整数型。受信したいメッセージを送信する PE のランクを指定する。
  - 任意の PE から受信したいときは、MPI_ANY_SOURCE を指定する。
- itag : 整数型。受信したいメッセージに付いているタグの値を指定する。
  - 任意のタグ値のメッセージを受信したいときは、MPI_ANY_TAG を指定する。
- icomm : 整数型。PE 集団を認識する番号であるコミュニケータを指定する。通常では MPI_COMM_WORLD を指定すればよい。
- istatus : MPI_Status 型の配列の先頭番地を指定する。受信状況に関する情報が入る。
  - 例題のように MPI_Status 型の配列を宣言して、指定する。
  - 受信したメッセージの送信元のランクが整数フィールド istatus.MPI_SOURCE に、タグが整数フィールド istatus.MPI_TAG に代入される。
- 戻り値 : 整数型。エラーコードが入る。

一方、行<16>-<18>で各 PE で計算済の部分和を隣接 PE である myid+1 番の PE に送信します。MPI 関数 MPI_Send の各引数の意味は以下の通りです。

```
int MPI_Send(sendbuf, icount, datatype, idest, itag, icomm);
```

- sendbuf : 送信領域の先頭番地を指定する。
- icount : 整数型。送信領域のデータ要素数を指定する。
- datatype : 整数型。送信領域のデータの型を指定する。

- idest ： 整数型。送信したい PE の icomm 内でのランクを指定する。
- itag ： 整数型。受信したいメッセージに付けられたタグの値を指定する。
- icomm ： 整数型。PE 集団を認識する番号であるコミュニケータを指定する。
  - 通常では MPI_COMM_WORLD を指定すればよい。
- 戻り値 ： 整数型。エラーコードが入る。

この例題では、最終的に PEnprocs-1 番にのみ演算の結果が得られます。また結果が得られるまで、通信が（PE 台数-1）回である nprocs-1 回必要となる点に注意してください。

高速プログラミングためのヒント：

さて次に、この素朴な通信方式を改良してみましょう。この改良された実装法は、データ構造とアルゴリズムを学ぶ上できわめて重要な概念である、二分木構造を利用するものです。先程の例題の行<12>-<18>を以下のプログラムに書き換えることで、二分木形態を用いた通信を実現できます。ここで説明を簡単にするために PE 台数は 2 の冪数をとるものとします。

```
<12'>   inlogp = 3; /* PE 台数 nprocs の 2 を底とする対数 log_2(nprocs) */
<13'>   k = 1;
<14'>   for(i=0; i<=inlogp-1; i++) {
<15'>     if ( (myid&k)  == k ) {
<16'>       ierr = MPI_Recv(&drecvbuf, 1, MPI_DOUBLE, myid-k, i, MPI_COMM_WORLD,
                      &istatus);
<17'>       dsendbuf = dsendbuf + drecvbuf;
<18'>       k *= 2;
<19'>     } else {
<20'>       ierr = MPI_Send(&dsendbuf, 1, MPI_DOUBLE, myid+k, i, MPI_COMM_WORLD);
<21'>       break;
<22'>     }
<23'>   }
```

8 台の PE を用いる場合を考えましょう。この場合の通信手順を図 3 に示します。

図 3 では、各 PE のランクを二進数で考えると i 反復時において、下位ビットから数えて i 番目のビットが 0 の PE がデータを送信、ビットが 1 の PE がデータを受信することがわかります。この受信-送信の関係を整理して書き直すと、二分木の形状をなすことが見てとれます。このことから、この通信方式を、**二分木通信方式**とよびます。また、各種ゲームにおけるトーナメント形式(勝ち抜き戦)の対戦表にも類似していることから、**トーナメント方式の通信方式**ともよばれます。

次に、この二分木通信方式の通信回数を考えましょう。トーナメント方式であるので、試合数に相当する合計の通信回数は nprocs-1 回ですが、各 log_2(nprocs)段のステージで同時に試合（通信）を行います。この同時通信について、通信衝突などの時間増加の要因がないと過程できれば、log_2(nprocs)回の通信回数で、結果を得ることができます。

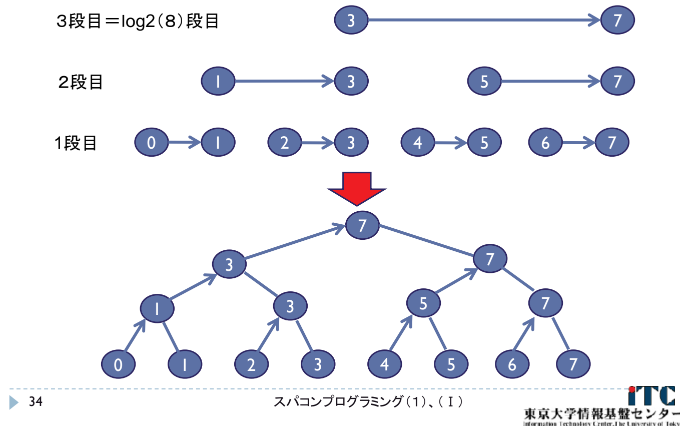


図 3　二分木通信方式の通信手順

この通信回数に関する改良は非常に有効であることに注意してください。たとえば 1024 台の PE を用いる場合、改良前の方法では 1023 回通信が必要なのに対して、二分木通信方式を用いた通信方式では log_2(1024)=10 なので、たった 10 回で済みます[3]。

４．３　リダクション演算

多くの並列プログラムでは、先程の計算例で示したように、通信と演算が必要な PE 間での演算処理が必要となります。このような演算を**縮約演算**または**リダクション演算**(reduction operation)、もしくは**集団通信演算**(collective communication operation) とよびます。

代表的なリダクション演算は、総和演算の他に最大値や最小値を求める演算などがあげられます。MPI ではこれらのリダクション演算を行う関数を、演算結果の持ち方の違いで以下の二つに分けています。

まず演算結果をある一つの PE に所有させる関数として、MPI_Reduce 関数があります。

```
int MPI_Reduce(sendbuf, recvbuf, icount, idatatype, iop, iroot, icomm);
```

- sendbuf ： 送信領域の先頭番地を指定する。
- recvbuf ： 受信領域の先頭番地を指定する。
  - iroot で指定した PE のみで書き込みがなされる。
  - 送信領域と受信領域は、同一であってはならない。すなわち、異なる配列を確保しなくてはならない。

[3] ただし、二分木通信方式が必ずしも高速でない点に注意してください。なぜなら、ネットワークの形状や性能に大きく依存するからです。たとえば PE 台数が少ない場合、1 つの PE にデータを収集する実装法が高速であることもあります。

- icount ： 整数型。送信領域のデータ要素数を指定する。
- idatatype ： 整数型。送信領域のデータの型を指定する。
  - 特に最小値や最大値と位置を返す演算を指定する場合は、以下を指定する。
    - MPI_2INT （整数型）
    - MPI_2FLOAT （単精度型）
    - MPI_2DOUBLE （倍精度型）
- iop ： 整数型。演算の種類を指定する。たとえば、以下を指定する。
  - MPI_SUM （総和）
  - MPI_PROD （積）
  - MPI_MAX （最大）
  - MPI_PROD （最小）
  - MPI_MAXLOC （最大と位置）
  - MPI_MINLOC （最小と位置）
- iroot ： 整数型。結果を受け取る PE の icomm 内でのランクを指定する。
  - 全ての icomm 内の PE で同じ値を指定する必要がある。
- icomm ： 整数型。PE 集団を認識する番号であるコミュニケータを指定する。
- 戻り値 ： 整数型。エラーコードが入る。

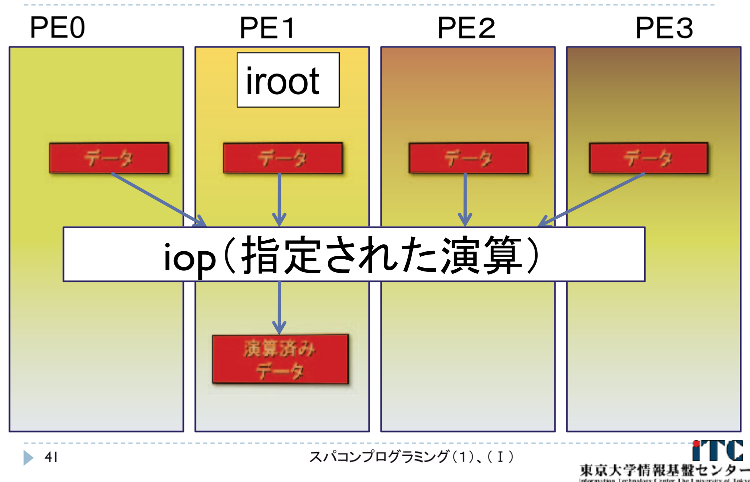


図 4 MPI_Reduce の概念

次に、演算結果を全ての PE に所有させる関数として MPI_Allreduce 関数があります。

```
int MPI_Allreduce(sendbuf, recvbuf, icount, idatatype, iop, icomm);
```

- sendbuf ： 送信領域の先頭番地を指定する。
- recvbuf ： 受信領域の先頭番地を指定する。
  - iroot で指定した PE のみで書き込みがなされる。

- なお、<u>送信領域と受信領域は、同一であってはならない</u>。すなわち、異なる配列を確保しなくてはならない。

- icount : 整数型。送信領域のデータ要素数を指定する。
- idatatype : 整数型。送信領域のデータの型を指定する。
  - 特に最小値や最大値と位置を返す演算を指定する場合は、
    - MPI_2INT (整数型)
    - MPI_2FLOAT (単精度型)
    - MPI_2DOUBLE (倍精度型)

  で指定する。
- iop : 整数型。演算の種類を指定する。
  - MPI_SUM (総和)
  - MPI_PROD (積)
  - MPI_MAX (最大)
  - MPI_PROD (最小)
  - MPI_MAXLOC (最大と位置)
  - MPI_MINLOC (最小と位置)

  など。
- icomm : 整数型。PE 集団を認識する番号であるコミュニケータを指定する。
- 戻り値 : 整数型。エラーコードが入る。

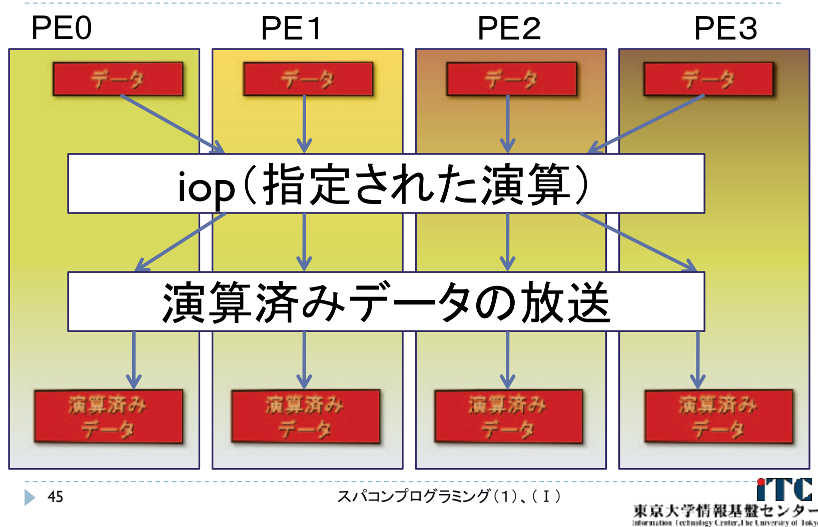


図 5　MPI_Allreduce の概念

高速プログラミングためのヒント：

MPI_Allreduce 関数は MPI_Reduce 関数に比べ、演算結果を全 PE に所有させるための放送処理が必要なことから、時間がかかることに注意してください。またこれらのリダクション演算

は、一般的に通信と演算を必要とするので単純な放送処理よりも時間がかかることを利用者は認識すべきです。すなわちリダクション演算を多用する実装方式は、性能の観点から好ましくないということを注意すべきです。

４．４　放送処理

放送処理を行う MPI 関数 MPI_Bcast について示します。

```
int MPI_Bcast(buf, icount, idatatype, iroot, icomm);
```

- buf ： 送受信領域の先頭番地を指定する。
- icount ： 整数型。送信領域のデータ要素数を指定する。
- idatatype ： 整数型。送信領域のデータの型を指定する。
  - よく用いられるデータ型としては、
    - MPI_INT （整数型）
    - MPI_REAL （実数型）
    - MPI_DOUBLE （倍精度実数型）

  などがある。
- iroot ： 整数型。結果を受け取る PE の icomm 内でのランクを指定する。
  - 全ての icomm 内の PE で同じ値を指定する必要がある。
- icomm ： 整数型。PE 集団を認識する番号であるコミュニケータを指定する。
- 戻り値 ： 整数型。エラーコードが入る。

MPI_Bcast 関数では、iroot で指定した PE が所有するデータを、すべての PE に変数 buf を通して放送することができます。

MPI_Bcastの概念（集団通信）

PE0　PE1　PE2　PE3

iroot

メッセージ　メッセージ　メッセージ　メッセージ

29　スパコンプログラミング(1)、(Ⅰ)　iTC 東京大学情報基盤センター Information Technology Center, The University of Tokyo

図 6　MPI_Bcast の概念

４．５　ノンブロッキング通信

いままでは、通信が**ブロッキング**である前提で説明をしてきました。通信がブロッキングとは、１対１通信の場合、MPI_Send が発行された PE は、どこかで MPI_Recv が発行されるまで処理が停止するということです。MPI では、ブロッキングではない通信（**ノンブロッキング**通信）についても、仕様を定めています。以降、ブロッキングとノンブロッキングの説明をします。

- **ブロッキング**
  - 送信／受信側のバッファ領域にメッセージが格納され、受信／送信側のバッファ領域が自由にアクセス・上書きできるまで、呼び出しが戻らない処理。
  - バッファ領域上のデータの一貫性を保障する。
- **ノンブロッキング**
  - 送信／受信側のバッファ領域のデータを保障せずすぐに呼び出しが戻る処理。
  - バッファ領域上のデータの一貫性を保障しない。
  - データ一貫性の保証はユーザの責任とする。

また、**ブロッキング**、**ノンブロッキング**が定義されると、各関数の挙動が定義できます。これは、ローカルとノンローカルという概念です。以下にそれを示します。

- **ローカル**
  - 手続きの完了が、それを実行しているプロセスのみに依存する。
  - ほかのユーザプロセスとの通信を必要としない処理。
- **ノンローカル**
  - 操作を完了するために、別のプロセスでの何らかの MPI 手続きの実行が必要かもしれない。
  - 別のユーザプロセスとの通信を必要とするかもしれない処理。

ノンブロッキングな通信を実現するには、通信するデータのコピーに関するとり扱い（バッファリング）が必要になります。MPI では、以下の４つのバッファリング方式が規定されています。

1. **標準通信モード**
   - デフォルトの方式
   - ノンローカル
   - 送出／受入メッセージのバッファリングは MPI に任せる。
     - バッファリングされるとき：　受信起動前に送信を完了できる
     - バッファリングされないとき：　送信が終了するまで待機する
2. **バッファ通信モード**
   - ローカル
   - 必ずバッファリングする。バッファ領域がないときはエラーとなる。
3. **同期通信モード**
   - ノンローカル

- バッファ領域が再利用でき、かつ、対応する受信／送信が開始されるまで待つ。

4. **レディ通信モード**
   - 処理自体はローカル
   - 対応する受信／送信が既に発行されている場合のみ実行できる。それ以外はエラーとなる。
     - ハンドシェーク処理を無くせるため、高い性能を発揮する。

いままで使ってきたMPI_Send関数は、以下のように説明できます。

- MPI_Send関数
  - ブロッキング
  - 標準通信モード（ノンローカル）
  - バッファ領域が安全な状態になるまで戻らない
    - バッファ領域がとれる場合：　メッセージがバッファリングされる。対応する受信が起動する前に、送信を完了できる。
    - バッファ領域がとれない場合：　対応する受信が発行されて、かつ、メッセージが受信側に完全にコピーされるまで、送信処理を完了できない。

ノンブロッキングな送信を実現する、MPI_Isend 関数のインタフェースについて、以下に示します。

```
int MPI_Isend(sendbuf,  icount,  datatype, idest,  itag,  icomm,  irequest);
```

- sendbuf ：　送信領域の先頭番地を指定する。
- icount ：　整数型。送信領域のデータ要素数を指定する。
- datatype ：　整数型。送信領域のデータの型を指定する。
- idest ：　整数型。送信したいPEのicomm 内でのランクを指定する。
- itag ：　整数型。受信したいメッセージに付けられたタグの値を指定する。
- icomm ：　整数型。PE集団を認識する番号であるコミュニケータを指定する。
  - 通常ではMPI_COMM_WORLD を指定すればよい。
- irequest：　MPI_Request型（整数型の配列）。送信を要求したメッセージにつけられた識別子が戻る。
- 戻り値 ：　整数型。エラーコードが入る。

MPI_Isend関数は、以下のように説明ができます。

- MPI_Isend関数
  - ノンブロッキング
  - 標準通信モード（ノンローカル）
    - 通信バッファ領域の状態にかかわらず戻る
      - バッファ領域がとれる場合は、メッセージがバッファリングされ、対応する

受信が起動する前に、送信処理が完了できる。

- バッファ領域がとれない場合は、対応する受信が発行され、メッセージが受信側に完全にコピーされるまで、送信処理が完了できない。
- 以上はシステムの挙動で、MPI_Wait 関数（後述）が呼ばれた場合の振舞いと理解すべき。

以下のように解釈してください。MPI_Send 関数は、この関数中に MPI_Wait 関数が入っている関数と見なせます。MPI_Isend 関数は、この関数中に MPI_Wait 関数が入っていない、かつ、すぐにユーザプログラム戻る関数であると見なせます。

MPI_Isend では、送信領域を書き換えても安全かどうかはチェックしません。そこで、送信領域を書き換えてもよい状況になるまで待つ関数が用意されています。これが、以下に説明する MPI_Wait 関数です。

```
int MPI_Wait(irequest,  istatus);
```

- irequest： MPI_Request 型（整数型配列）。送信を要求したメッセージにつけられた識別子。
- istatus ： MPI_Status 型（整数型配列）。受信状況に関する情報が入る。
  - 要素数が MPI_STATUS_SIZE の整数配列を宣言して指定する。
  - 受信したメッセージの送信元のランクが istatus[MPI_SOURCE] 、タグが istatus[MPI_TAG] に代入される。
- 戻り値： 整数型。エラーコードが入る。

高速プログラミングためのヒント：

MPI_Send を利用した高速実装法として、以下のように、**通信と計算をオーバラップ**させるように記述する実装方法があります。

```
/* PE 0 が持つデータ a をあらかじめ非同期通信で PE1 から PE nprocs-1 に、送っておく */
if (myid == 0)  {
    for (i=1;  i<numprocs;  i++)  {
        ierr = MPI_Isend( a,  N,  MPI_DOUBLE,  i,
            i_loop,  MPI_COMM_WORLD,  &irequest[i] );
    }
 }  else  {
    ierr = MPI_Recv( a,  N,  MPI_DOUBLE,  0,  i_loop,
                 MPI_COMM_WORLD,  &istatus );
}

/* PE0 が 他の PE が受信中に、オーバラップして計算できる部分 */
```

```
/* PE0 の送信領域が書きかえれるか待つ */
if (myid == 0)   {
     for (i=1； i<numprocs； i++)  {
         ierr = MPI_Wait(&irequest[i],  &istatus);
     }
}
```

ここで注意は、通信中に計算が行えるハードウエアやＯＳが実装されていることが条件です。もしそうでない場合、MPI_Send を利用した場合や、通信と計算をオーバラップさせないように記述した方法と変わらかったり、むしろ遅くなることも珍しくありません。

４．６　コミュニケータの分割

今までの前提では、初期化した時に存在した PE すべてが、放送処理やリダクション演算に関連します。その関連する PE の集合情報は、コミュニケータとよばれる変数に格納され、通常は MPI_COMM_WORLD であることを説明しました。

放送処理やリダクション演算は、１対１通信よりも時間を必要とする通信であることを説明しました。この処理の時間は、PE 数に関連します。そこで PE 数を減らせば、処理時間の短縮が望めます。

高速プログラミングためのヒント：

リダクション演算などにおいて対象となる PE 数を減らすには、コミュニケータを分割する必要があります。このコミュニケータを分割する関数が、以下に示す MPI_Comm_split 関数です。

```
int MPI_Comm_split(icomm, icolor, ikey, inewcomm);
```

- icomm：　整数型。分割するコミュニケータを指定する。
  - 通常では MPI_COMM_WORLD を指定すればよい。
- icolor：整数型。分割するコミュニケータにおける同一グループを決める制御変数である。
- ikey ：整数型。分割したコミュニケータ内におけるランクを指定する。
- inewcomm:　整数型。新しいコミュニケータを指定する。
- 戻り値 ：　整数型。エラーコードが入る。

ここで、分割されたコミュニケータは、inewcom になるのですが、この inewcom では、複数 PE の集合になっています(図 7)。

したがって、この新しい inewcom を用いて放送処理やリダクション演算を記述する場合、放送処理やリダクション処理が複数同時に実行される点に注意してください。このことで、それぞれの放送処理やリダクション演算時間が短縮されるだけでなく、並列に処理が実行されることで、さらに高速化されます。

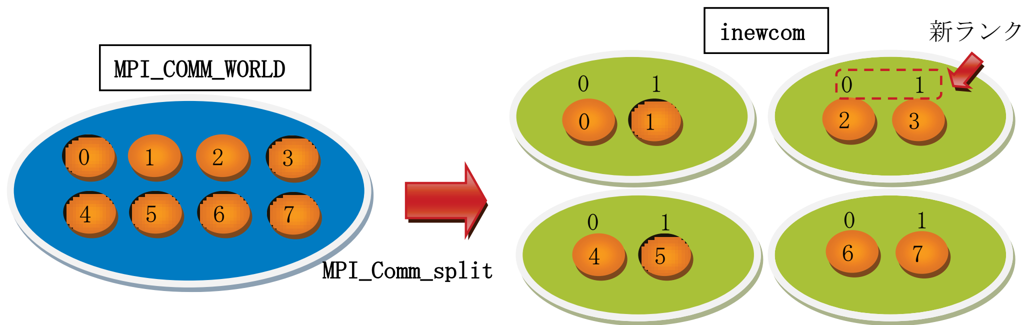


図 7　MPI_Comm_split でのコミュニケータの分割

以下に、プログラム例を示します。

```
int     myid_x, myid_y;
int     MPI_COMM_Y;
…
myid_x = myid / NPROCS_Y;
myid_y = myid % NPROCS_Y;
MPI_Comm_split(MPI_COMM_WORLD, myid_y, myid_x, &MPI_COMM_Y);
…
MPI_Reduce(&iflag, &iflag_t, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_Y);
…
```

以上の例では、0～nprocs-1 に割り当てられた元のランクは、NPROCS_Y 個づつ別のコミュニケータに分割されます。コミュニケータのグループは、myid_y（0～myid%NPROCS_Y-1）で決まります。それぞれのコミュニケータ内のランクは、myid_x となります。最後の MPI_Reduce は、NPROCS_Y 個並列に実行され、それぞれの MPI_Reduce に関連する PE 数は ceil(nprocs/NPROCS_Y)個です。ここで、ceil(x)は、x の小数点切り上げの整数を返す関数です。

５．実行性能評価

実行性能を評価するにあたり、プログラム中の測定したい箇所の時間をどのように計るのか知る必要があります。ここではまず、その方法を説明します。

５．１　実行時間の測定方法について

MPI では、実行時間測定のための関数として MPI_Wtime があります。使用例を以下に示します。

```
<1 >      double t1, t2, t0, t_w;
<2 >
<3 >      ierr = MPI_Barrier(MPI_COMM_WORLD);
<4 >      t1 = MPI_Wtime();
```

```
<5 >      /*  測定したい部分の始まり */
<6 >          .....
<7 >
<8 >      /*  測定したい部分の終り */
<9 >      ierr = MPI_Barrier(MPI_COMM_WORLD);
<10>      t2 = MPI_Wtime();
<11>      t0 =  t2 - t1;
<12>      ierr = MPI_Reduce(&t0, &t_w, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
<13>
<14>      /* 実行時間は 0 番 PE が持つ */
<15>      if (myid == 0) printf("  execution time = : %8.4lf  [sec.] ¥n", t_w);
```

なお、MPI_Barrier 関数は、全 PE で同期をとる関数です。この方法では、各 PE で得られた実行時間の内、最も大きいもの（遅いもの）を全体の実行時間としています。

５．２　性能評価例

それでは最初に示した円周率プログラムを、東京大学情報基盤センターに設置されている HITACHI SR11000/J2 の 1 ノード（最大で 16PE ですが、実験環境では 8PE まで利用可能）を用いて性能評価した例を示します。図 8 に、実行時間をプロットしたものを載せます。

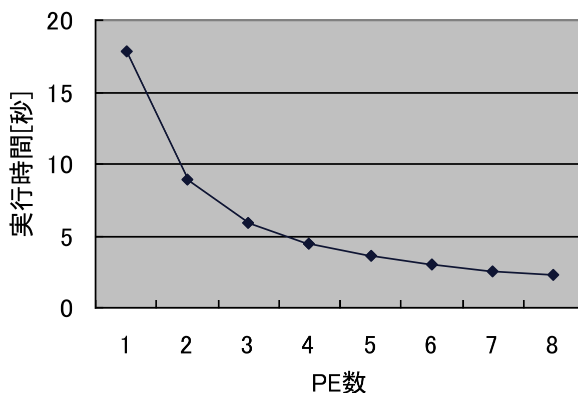


図 8　円周率計算プログラムの実行時間[秒]。n=1,000,000,000 を指定。

ここで図 8 の実行時間が、性能のよいものかどうか評価する尺度の 1 つとして、台数効果という指標があります。台数効果は以下のように定義されます。

p 台のときの台数効果　：＝ 1 台の実行時間 / p 台の実行時間　　　…(1)

式(1)から p 台のときの台数効果が p に近い程、性能が良いということがいえます。
図 9 は円周率のプログラムに関して、台数効果を示したものです。

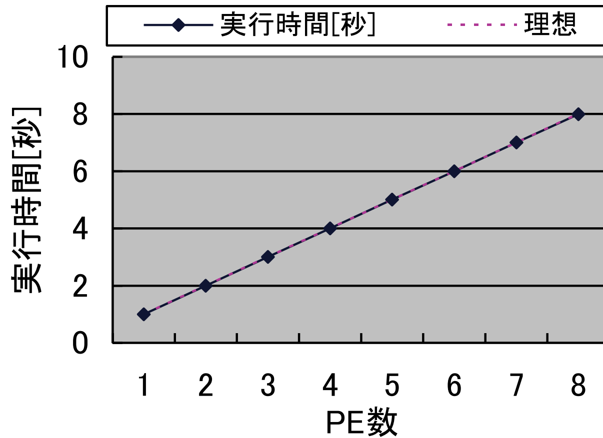


図 9　円周率計算プログラムの台数効果。n=1,000,000,000 を指定。

図 9 の結果から、円周率計算の並列プログラムは、きわめて性能が良いプログラムであるということがいえます[4]。

6．その他の高速プログラミングのためのヒント

数値計算処理において性能を向上させる技法として、以下のものが知られています。

- **最内ループ連続アクセス**：
  ネストされたループにおいて、最も内側のループに対する配列のアクセスが連続となるように、データの複製、ループネストの順番変更、およびデータ構造の変更などを行う技法。連続アクセスにより、データの読みだしに関するキャッシュミスによる速度低下を防いだり、データの書き込み処理の効率化をねらう。たとえば C 言語を用いる場合、2 次元配列は行方向にデータが収納[5]されるため、最内ループのアクセス方向を行方向にするようにループを構成する。
- **ループアンローリング**：
  ループを展開することで、(1)ループ自体のオーバヘッドの削減、(2)データの先行読みだし(プリロード)や書き込み(ポストストア)の機会の増加、(3)レジスタ割り当ての最適化可能性の増大、などコンパイラによるコード最適化を促すことによる高速化技法。
- **ブロック化**：
  頻繁にアクセスするデータをキャッシュにのせることで、キャッシュミスによる速度低下を防ぐ技法。ソースコードのみ変更する場合は、**タイリング**と呼ばれる。タイリングは、

[4]台数効果の指標では、1 台利用のときの性能が特に重要となります。すなわち 1 台利用のときの実行時間を効率の悪いものにすると、台数効果は非常によいものとなり、正当な評価にならないことに注意してください。一般的に公平な性能評価をするために 1 台の実行性能は、MPI コードなど並列処理に必要だが逐次処理に不必要なコードの除去をした上で、実行時間を計測すべきです。この性能評価では、MPI コードの除去はしていません。

[5] 行列 A=(a_{ij})とするとき、j 方向のこと。

コンパイラのコード最適化技法の１つとして知られている。

また現在のコンパイラによるコード最適化では実現できない、アルゴリズムレベルでのブロック化技法も知られている。これは、**ブロック化アルゴリズム**と呼ばれる(例：LU 分解における多段多列同時消去アルゴリズム)。

本稿では、以上の技法に関する詳細な説明は割愛します。ここでは、行列積におけるループアンローリングの一例を示すに留めます。

(a) オリジナルコード

```
for (i=0, i<n, i++)
  for (j=0, j<n, j++)
    for (k=0, k<n, k++)
      c[i][j] += a[i][k] * b[k][j];
```

(b) k-ループをアンローリング (n は 2 で割り切れるとする)

```
for (i=0, i<n, i++)
  for (j=0, j<n, j++)
    for (k=0, k<n, k+=2)
      c[i][j] += a[i][k  ] * b[k  ][j]
               + a[i][k+1] * b[k+1][j];
```

(c) i, j-ループをアンローリング (n は 2 で割り切れるとする)

```
 for (i=0; i<n;  i+=2)
   for (j=0; j<n; j+=2)
      for (k=0; k<n; k++)  {
         C[i   ][j   ] += A[i  ][k] *B[k][j  ];
         C[i   ][j+1 ] += A[i  ][k] *B[k][j+1];
         C[i+1 ][j   ] += A[i+1][k] *B[k][j  ];
         C[i+1 ][j+1 ] += A[i+1][k] *B[k][j+1];
      }
```

参 考 文 献

[1] P.パチェコ 著 ／ 秋葉 博 訳、MPI 並列プログラミング、培風館、2001

[2] 青山幸也 著、並列プログラミング虎の巻 MPI 版、理化学研究所情報基盤センター
(http://accc.riken.jp/HPC/training/text.html)

[3] Message Passing Interface Forum
(http://www.mpi-forum.org/)

[4] MPI-J メーリングリスト
(http://phase.hpcc.jp/phase/mpi-j/ml/)

[5] 富田眞治著、並列コンピュータ工学、昭晃堂、1996

東京大学情報基盤センター・スーパーコンピューティングニュース
Vol. 9 No. Special Issue 1 (2008.2)

スーパーコンピューティングニュース編集スタッフ
編　集　長　　米澤明憲
編 集 幹 事　　西澤明生
編 集 委 員　　石川裕，佐藤周行，佐藤文俊，黒田久泰，松葉浩也，片桐孝洋，
吉廣保，大日方一男，櫻田芳男，丹下藤夫，有賀浩，井爪健雄

編集・発行　　東京大学情報基盤センター
スーパーコンピューティング部門

〒113-8658　東京都文京区弥生 2-11-16
（電話）　03-5841-2717（ダイヤルイン）
（FAX）　03-5841-2708