JAIST Repository
https://dspace.jaist.ac.jp/

Title	Cray XT5 における数値流体プログラミングの Hybrid 並列による高速化について
Author(s)	西條, 晶彦
Citation	
Issue Date	2010-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/8957
Rights	
Description	Supervisor:松澤照男, 情報科学研究科, 修士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

修 士 論 文

Cray XT5 における数値流体プログラミングの Hybrid 並列による高速化について

北陸先端科学技術大学院大学
情報科学研究科情報科学専攻

西條 晶彦

2010 年 3 月

修 士 論 文

Cray XT5 における数値流体プログラミングの Hybrid 並列による高速化について

指導教官　松澤照男 教授

審査委員主査　松澤照男 教授
審査委員　前園涼 講師
審査委員　井口寧 准教授

北陸先端科学技術大学院大学
情報科学研究科情報科学専攻

0810026 西條 晶彦

提出年月: 2010 年 2 月

Copyright © 2010 by Akihiko SAIJOU

概 要

HPC 用アプリケーションで並列処理を行うには共有メモリ型の並列手法とメッセージ通信による並列手法がある．近年，HPC の多くを占めるようになった SMP クラスタマシンにおいてはこれら 2 つの並列手法を組み合わせた Hybrid 並列が効率が高いと言われている．本論文では SMP クラスタマシンである Cray XT5 において OpenMP と MPI のベンチマークを行い，マシン特性を調べた．その結果から Hybrid 並列に適した次に Hybrid 並列化を行った数値流体アプリケーションを構成し，Cray XT5 による大規模高並列数値流体計算を行った． ベンチマークの結果から，Cray XT5 の並列化において通信の同期の部分で最も時間を消費していることがわかり，Hybrid 並列によりこの同期時間を減らすことができることがわかった．また，OpenMP スレッド並列による並列化部分を高速化するために，スレッド並列をループごとに生成，同期をする時間を削減する OpenMP SPMD の手法を用いた．Cray XT5 で実行させたところ，Hybrid 並列アプリケーションは Pure 並列アプリケーションよりも同程度からやや低下したが，OpenMP SPMD の手法は並列度が上がると通常のマスターオンリー Hybrid 並列よりも性能をあげることがわかった．

目 次

第 1 章　はじめに **1**
1.1　研究の背景と目的 1
1.2　本稿の構成 2

第 2 章　並列計算機 **3**
2.1　共有メモリ型 3
2.1.1　UMA 3
2.1.2　NUMA 3
2.2　分散メモリ型 4
2.3　分散共有メモリ型 4

第 3 章　Hybrid 並列 **6**
3.1　Hybrid 並列とは 6
3.1.1　Message Passing Interface 7
3.1.2　OpenMP 7
3.2　Hybrid プログラミングモデル 8
3.2.1　マスターオンリーモデル 8
3.2.2　スレッドオーバーラップ 9

第 4 章　Cray XT5 **13**
4.0.3　Hybrid ジョブの実行 13

第 5 章　計算性能ベンチマーク **15**
5.1　OpenMP ベンチマーク 15
5.1.1　Stream Benchmarks 15
5.1.2　ECPP Microbenchmarks 16
5.2　MPI ベンチマーク 17
5.2.1　IMB 17
5.3　NAS Parallel Benchmark, Multi-Zone Versions 18
5.3.1　計算条件 19
5.3.2　結果と考察 21
5.4　まとめ 22

第 6 章　Hybrid 並列数値流体アプリケーション **30**
6.1　対象問題 30
6.2　有限要素法 31
6.2.1　重み付き残差法 31
6.3　共役勾配法 31
6.3.1　原理 32
6.4　並列化 32
6.4.1　領域分割 33
6.4.2　ループレベル分割 34
6.5　結果と考察 35

第 7 章　結論 **39**
7.1　結論 39
7.2　今後の課題 39

謝辞 **40**

図 目 次

2.1 共有メモリ計算機 4
2.2 分散メモリ計算機 5
2.3 分散共有メモリ計算機 5

3.1 マスターオンリーモデル 8
3.2 MPI + OpenMP SPMD モデル 11

4.1 Cray XT5 のノード 14

5.1 OpenMP Triad 16
5.2 EPCC: OpenMP Overhead 17
5.3 IMB Pingpong の概要 [] 18
5.4 IMB Pingpong の結果 19
5.5 ネットワークレイテンシ 20
5.6 領域のゾーン分割 [6] 21
5.7 SP (CLASS C) における計算時間と通信時間 22
5.8 SP (CLASS D) における計算時間と通信時間 23
5.9 SP (CLASS E) における計算時間と通信時間 24
5.10 SP (CLASS E) 2048 並列における各ノードの通信時間 25
5.11 BT (CLASS C) における計算時間と通信時間 26
5.12 BT (CLASS D) における計算時間と通信時間 27
5.13 BT (CLASS E) における計算時間と通信時間 28
5.14 BT (CLASS E) 2048 並列における各ノードの通信時間 29

6.1 ポテンシャル問題 36
6.2 計算領域の要素分割 37
6.3 SMALL (1024^2 節点) における計算結果 38
6.4 LARGE (2048^2 節点) における計算結果 38

表 目 次

4.1 Cray XT5 のマシン仕様 13

第1章　はじめに

1.1　研究の背景と目的

並列計算機のアーキテクチャは変化している．大規模な計算を高速に行うには並列計算機による並列処理が欠かせない．並列計算機の能力を決める要素にはコアプロセッサの性能やネットワーク構造など様々なものがあるが，メモリアーキテクチャは影響の大きいものの一つである．並列計算機のメモリアーキテクチャは大きく分けて，複数のプロセサがメモリ空間を共有する共有メモリ型，計算機をネットワークで結び通信によって大規模メモリを実現する分散メモリ型，そしてこれら組み合わせである分散共有メモリ型がある．コストパフォーマンスの良さから，現在の並列計算機の多くは分散共有メモリ型である．

分散共有メモリ型並列計算機は共有メモリ型の計算機を 1 ノードとし，複数のノードをネットワークで接続した形態の計算機である．このようなアーキテクチャの計算機においてはノード内の並列計算を OpenMP などによる共有メモリ並列で，ノード間の並列計算をメッセージパッシングによる通信 (MPI) で行うという，二つの並列化モデルを混ぜ込んだ Hybrid 並列化手法が性能を引き出すのに有効であると言われている．先行研究では数値流体計算において Hybrid 並列は MPI のみを用いる Pure 並列の方と比べて同程度かやや劣るが，一部のケースにおいて Hybrid 並列の方が性能が良いという結果が報告されている [1].

Hybrid 並列の有効性は未だ明らかではない．Hybrid 並列は MPI の通信オーバーヘッドを避けることができるが，その代わりに共有メモリ並列を混ぜ込むこと y によって生じるスレッド間の同期オーバーヘッドが加わるため，常にどちらかのモデルが良いとは言えない．実行する計算機のメモリ性能，ネットワーク性能，計算対象の性質やプログラミングの手法によっても， Hybrid 並列の性能は変わりうる．数値計算問題において，共有メモリ型に並列化手法の標準規格として OpenMP がある．OpenMP はコード中に並列化ディレクティブを埋め込むことによって並列化を行うが，これをどのように挿入するかによっても Hybrid 並列の性能は変わりうる．

したがって，Hybrid の有効性調べるためにはターゲットマシンと計算対象問題を明確にしたほうがよい．中尾 [10] は Hybrid 並列のための新しい手法を提唱したが，評価に用いているのは行列積あるいは逆行列ソルバであり，大規模並列計算機の主要アプリケーションでは用いられないアプリケーションである．

本研究では，本学に 2009 年に導入された大規模並列分散共有メモリ型計算機である Cray XT5 上を用いる．本学における Cray XT5 の構成はノードあたり 4 コア CPU を 2 ソケッ

ト，全体で 256 ノードが使用可能な大規模並列計算機である．Cray XT5 の OpenMP によるノード内の並列処理性能，MPI によるノード間の通信性能を調査し，Hybrid 並列に適した Crayx XT5 の特徴を調査する．そのうえで数値流体計算で最も使われることの多い 計算コストの高いポワソンを，大規模計算に適した反復法である共役勾配法を用いて Hybrid 並列化を行い，OpenMP のプログラミングモデルを比較して Hybrid 並列の有効性を検証する．

1.2 本稿の構成

本稿では，まず最初に問題の背景と研究の目的を述べた．2 章では並列計算機についてメモリアーキテクチャの観点から解説する．3 章では Hybrid 並列とは何かを述べた後，Hybrid 並列の 2 つのプログラグラミングモデルを考察する．4 章では本研究のターゲットマシンである Cray XT5 の構成とソフトウェアまわりを説明する．5 章では並列計算のベンチマークを行い，Cray XT5 の性能を調査した．6 章では Hybrid 並列数値流体コードを開発し，2 つのプログラミングモデルを比較した．7 章でこれらの結果をまとめた．

第2章　並列計算機

並列計算機には様々な形態のものがある．プロセッサのメモリ間通信をどのように行うかということは並列計算の性能を出す上で重要である．メモリとネットワークの構造により並列計算機のアーキテクチャは以下のように，共有メモリ型，分散メモリ型，分散共有メモリ型に大別できる．

2.1　共有メモリ型

共有メモリ型とはプロセッサがメモリを論理的に共有している並列計算機のアーキテクチャである．複数のプロセッサが単一のメモリアドレスにアクセスをすることができる．このため，データを分割，配置しなくても並列化処理が可能である．一方，同一メモリ空間に複数プロセッサがアクセスする競合が生じるため，共有バスの処理能力によりプロセッサ数の上限が決まってくる．

複数の同一なプロセッサが単一のメモリに接続された共有メモリ型アーキテクチャのことを SMP (Symmetric Multiprocessor) という．

共有メモリ型アーキテクチャにはさらにプロセッサのメモリアクセス時間に関して，UMA と NUMA がある．

2.1.1　UMA

UMA とはあるプロセッサのメモリアクセスに必要時間が，メモリアドレスの位置によらず均一であるようなアーキテクチャのことである．

2.1.2　NUMA

共有メモリ型において，あるプロセッサからのメモリアクセス時間がメモリアドレスの位置に依存し，非均一であるようなアーキテクチャのことを NUMA (Non-Unified Memory Arcess) という．実装方式として，複数のメモリをアドレス変換器を用いることで仮想的な単一メモリとしてプロセッサからアクセスできるものがある．

現代の多くのプロセッサは主メモリとは別にキャッシュという高速で小容量なメモリを局所的に持っている．このキャッシュに対して一貫性を持ったメモリアクセスが出来るよ

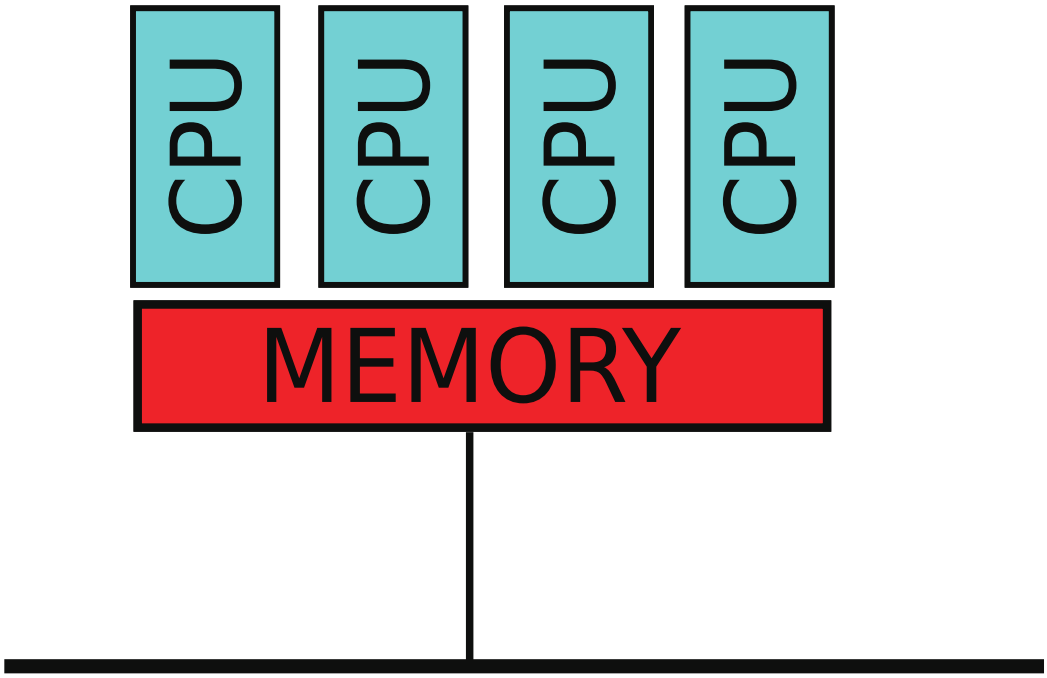

図 2.1: 共有メモリ計算機

うな NUMA アーキテクチャのことを ccNUMA (cache-coherent NUMA) という．ほとんの NUMA マシンは ccNUMA である．

2.2 分散メモリ型

分散メモリ方式ではプロセッサとメモリを共有しない．メモリ間通信はマシンのネットワーク通信を通して行うアーキテクチャである．大規模な並列計算機を構築することが可能であるが，共有メモリ型に比べて通信のオーバヘッドは非常に大きい．汎用の PC と LAN カードなどを用いて非常に安価に構築された並列計算機を PC Cluster ということがある．

2.3 分散共有メモリ型

分散共有メモリ型並列計算とは，共有メモリ型計算機を分散メモリ型のようにネットワークで複数繋いで並列化させた構造をしている．このように共有メモリ型の SMP を分散メモリ的に繋いだ並列計算機のことを SMP Cluster ということがある．近年，多くの大規模並列計算機は SMP Cluster である．

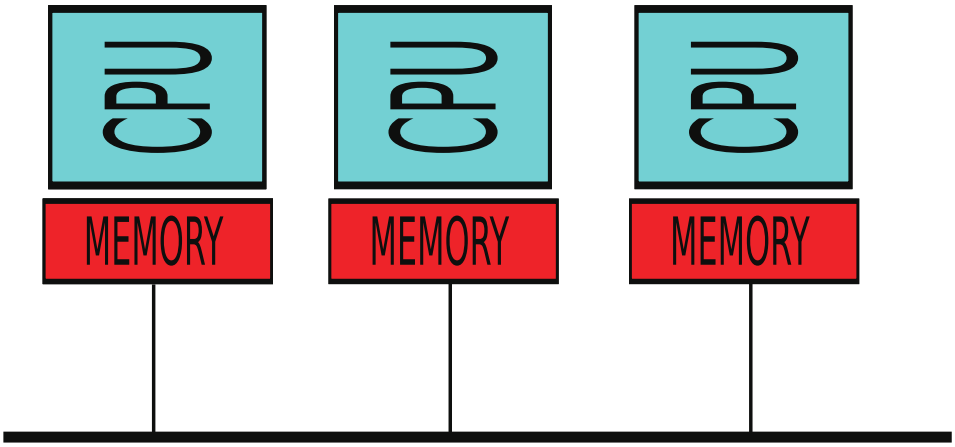

図 2.2: 分散メモリ計算機

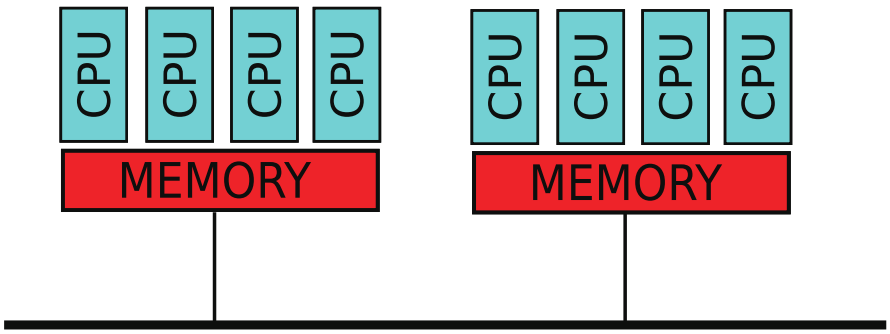

図 2.3: 分散共有メモリ計算機

第3章　Hybrid並列

3.1　Hybrid並列とは

メッセージ通信による分散並列手法と、共有メモリに並列手法を混ぜる行う並列プログラミングモデルのことを Hybrid 並列という．現在のほとんどの大規模計算機は共有メモリを持った SMP を，分散並列に繋いだ SMP Cluster 型であるため．並列アプリケーションでメッセージ通信並列と共有メモリ並列を実現するための標準的な規格として、それぞれ MPI と OpenMP がある．

共有メモリ並列と分散並列手法は並列化アルゴリズムとして本質的な違いはないためどんな一般的な状況でも Hybrid 並列は優れている技術ではないが、アプリケーションが大規模・高並列になると多くの場合，計算時間に対して通信過多になるため，Hybrid 並列によって通信時間を減らすことによる高速化が可能になる場合がある．また，既に MPI 並列化されたコードを，コンパイラの自動並列化機構によりスレッド並列化し，Hybrid 並列コードにするという手法も有用である．

このような共有メモリ並列の優位性を現在のマシンにおいて実現するために両者を混ぜて性能を出そう、というのが Hybrid 並列の考え方である．

分散並列に用いる MPI の実装では、内部においてすでに共有メモリ並列を用いている．MPI 実装が十分に賢ければ Hybrid 並列は行う必要がなくなる．また，共有メモリ並列はオーバーヘッドが大きいため、小さい問題を低い並列度で解く場合は苦労してコードを Hybrid 化しても現状よりも遅くなることもある．

Hybrid 並列の効果が高くなるのは、並列度が高く（Hybrid 化で通信レイテンシの影響を抑えられる）、コアあたりの問題規模が小さい場合，メモリのまた、各プロセスにおけるスレッド並列（OpenMP）の速度を高めることが非常に重要であると考えられる．

以下に Hybrid 並列の長所と短所を列挙する．

長所

1. 高並列時でも高効率を保てる
2. 通信オーバーヘッドが少ない

短所

1. プログラミングが複雑になる

2. 対象コード（低並列度やスレッド生成を頻繁に行う場合) によってはむしろ遅くなることもある

Hybrid 並列を行うための規格としてはプロセス間通信に MPI ではなく PVM を使う方法や，共有メモリ並列に OpenMP ではなく pthread などを使う方法がある．また，IMPACT のように高速なハードウェアとコンパイラによる自動並列処理を用いることにより，プログラムを改変せずに高効率な Hybrid 並列を実現している例もある [12].

3.1.1 Message Passing Interface

MPI はメッセージ通信による並列計算のための API 規格であり，C, C++, Fortran 等，様々な言語から使用が可能である．MPI が定めているのは規格だけであり，多くの実装が存在している．MPI はメッセージ通信並列の規格において事実上の標準となっている．

MPI はライブラリ呼び出しによる並列化であるので，逐次コードを MPI で並列化するには MPI 手続きの呼び出しを行うようにコードを改変しなければならない．

MPI はメッセージ通信による分散メモリ型の並列プログラミングモデルをとっているため，TCP 上のソケットに対して並列プログラムを動作させることが出来る．しかしながら，MPI は共有メモリ型並列計算機においても動作し，この場合でもメモリの局所性が高まるために高いパフォーマンスを得ることがある．

MPI-2 からはスレッド並列のためのサブルーチンが追加された.

3.1.2 OpenMP

OpenMP は、共有メモリ型並列計算のための規格である．C, C++, Fortran に対応した規格が存在する．MPI はメッセージ通信の命令をライブラリの関数として呼び出さなければならないが， OpenMP は並列化のためのディレクティブを挿入することで並列化を行う。OpenMP が利用できない場合はこのディレクティブを無視できるため，逐次コードからの改変は非常に少なくすむというメリットがある．

OpenMP による並列化は指示しない場合，ループを均等に分割して行うため，MPI に比べてデータの局所性が低くなる傾向がある．すなわち，あるスレッドが必要としているデータが別のスレッドに渡されている可能性があり，性能低下の原因となっている．局所性を上げるためには適切な指示のディレクティブを挿入したり，データ構造や処理順序を変更する必要がある．また，共有メモリ上での並列であるため並列化効率はコンパイラの最適化の影響を非常に強く受ける．

標準化された OpenMP では分散メモリ型計算機において並列計算を行うことはできない Intel の拡張規格である Cluster OpenMP を用いると OpenMP の手法と新たなディレクティブを用いることによって，分散メモリ計算機上で並列計算を行うことができる [7].

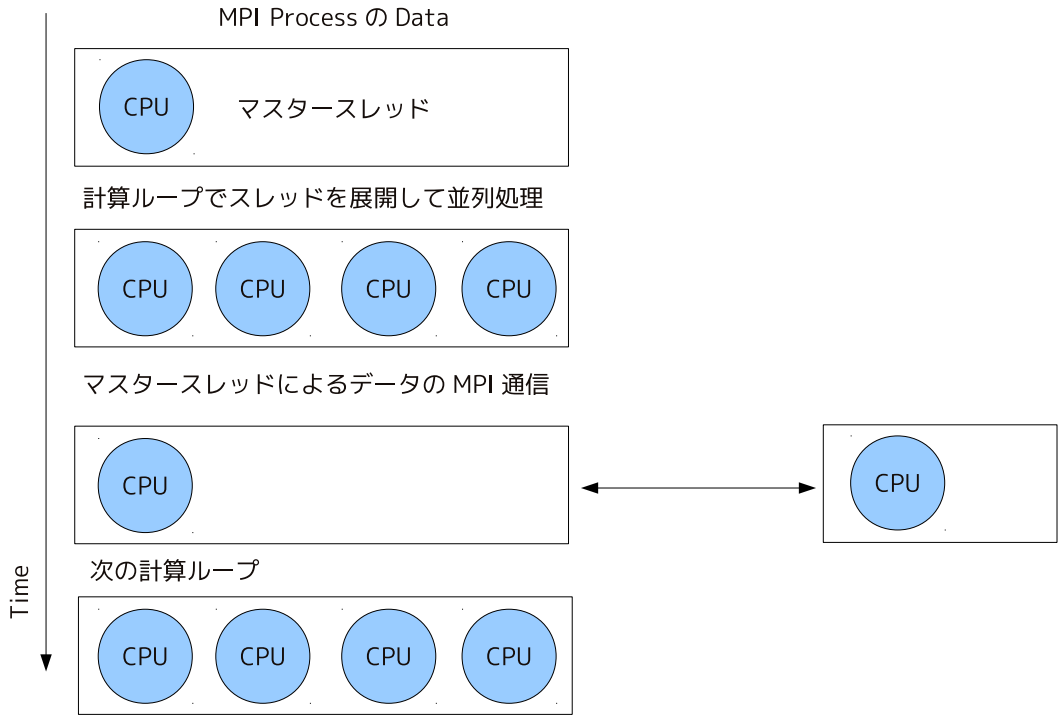

図 3.1: マスターオンリーモデル

3.2 Hybrid プログラミングモデル

アプリケーションを MPI と OpenMP の Hybrid 並列化する場合，その手法として Hager ら (2009) は以下の 2 つの分類を与えている [8]．つまり，MPI による通信をスレッド中で行う場合と行わない場合である．

3.2.1 マスターオンリーモデル

Hybrid 並列を実現するプログラミングモデルでは，領域分割による粗粒度の並列を MPI で行い，プログラミング中のループに OpenMP のディレクティブを挿入することで並列化する、というのが最も実現しやすい．

領域分割で並列度を上げる場合、どうしてもデータ通信のコストが計算に比べて多くなる。そこで、領域分割の数を抑え、その分の並列度をプログラム中の DO ループを分割することに振り分ける。通信の部分ではマスタースレッドにのみが逐次計算を行う（3.1)。

プログラミングのコストは少なく、既存の並列 MPI コードを簡単にハイブリッド化できる。

以下にコードの概要を示す．

```
do i = 1, N
  !スレッドを生成して並列処理
  !$OMP parallel do
   do j = 1
  ...

   end do
  !暗黙の同期とスレッドのクローズ

  ...

  ! MPI 通信を行う
  call MPI_send()
  call MPI_Recv()

end do
```

3.2.2 スレッドオーバーラップ

通信時間隠蔽

計算用のスレッドと通信用のスレッドに分け、計算と通信をオーバーラップさせる手法. 通信用のスレッドではMPIによる領域間通信のデータを扱い，同時にそのデータと関係のない計算を計算用のスレッドで実行する． MPIプロセス間通信時間を実行し待機している間に，別のスレッドが計算をすることにより，待機時間を隠蔽することができる．

このような通信時間隠蔽の手法はHybrid並列だけではなく，MPIの非ブロッキング通信によって行われる非常に一般的な手法である．この手法の場合は複数のスレッドを使って通信を行うことでネットワーク帯域を十分に使うことができると考えられている．しかし，通信データと計算データを分けて計算する場合コードが複雑になるため本研究では扱わない．以下に通信時間隠蔽スレッドオーバーラップの手法の概要を示す.

```
!$OMP parallel
do i = 1, N
  if (omp_get_num_thread() .eq. comm_thread_num) then
     ! communication threads
     call MPI_Send(data_X)
     call MPI_Recv(data_X)
  else
     ! do computation without data_X
     ...
  end if
     ! Wait all threads
     !$OMP barrier

     ! do computation with data_x
     ...
end do
!$OMP end parallel
```

MPI + OpenMP SPMD

スレッドオーバーラップ法の変形である．MPI による SPMD プロセスの中で，さらに計算手続き全体を OpenMP によるスレッドで複数実行させ，スレッドが同一のコードに対してループの異なる部分を処理していくという，OpenMP SPMD 型の並列化手法．マスターオンリーモデルに比べてスレッド生成回数を低く抑えることができ、OpenMP プログラムを高速に実行させることができる [4]．スレッドを複数動作させてその中から MPI を実行する．ここでは 単一のスレッドだけが MPI 呼び出しを実行し，その他のスレッド待機するものを考える．処理の概要を図 3.2 に示す.

OpenMP SPMD において MPI 実行する場合，スレッド内から MPI 手続きにアクセスすることになり，MPI 手続き中の一貫したメモリ内容を破壊してしまう恐れがある。これを避けるにはスレッド安全な MPI ライブラリが必要であり，MPI の初期化に MPI_Init_thread を用いて MPI ライブラリにスレッド安全性レベル指定しなければならない．指定できるスレッド安全性は，スレッド内で MPI 手続きを呼ぶことの出来ない MPI_THREAD_SINGLE, マスタースレッドのみが MPI 手続きを呼ぶ，MPI_THREAD_FUNNELED, 単一のスレッドが MPI 手続きを呼ぶ MPI_THREAD_SERIALIZED, 複数のスレッドが MPI 手続きを呼ぶ MPI_THREAD_MULTIPLE の 4 つがある．

ここでは単一のスレッドだけが MPI を呼び出して実行するようにする．このようにするとデータ構造の改変が必要ない．そのようにするためには

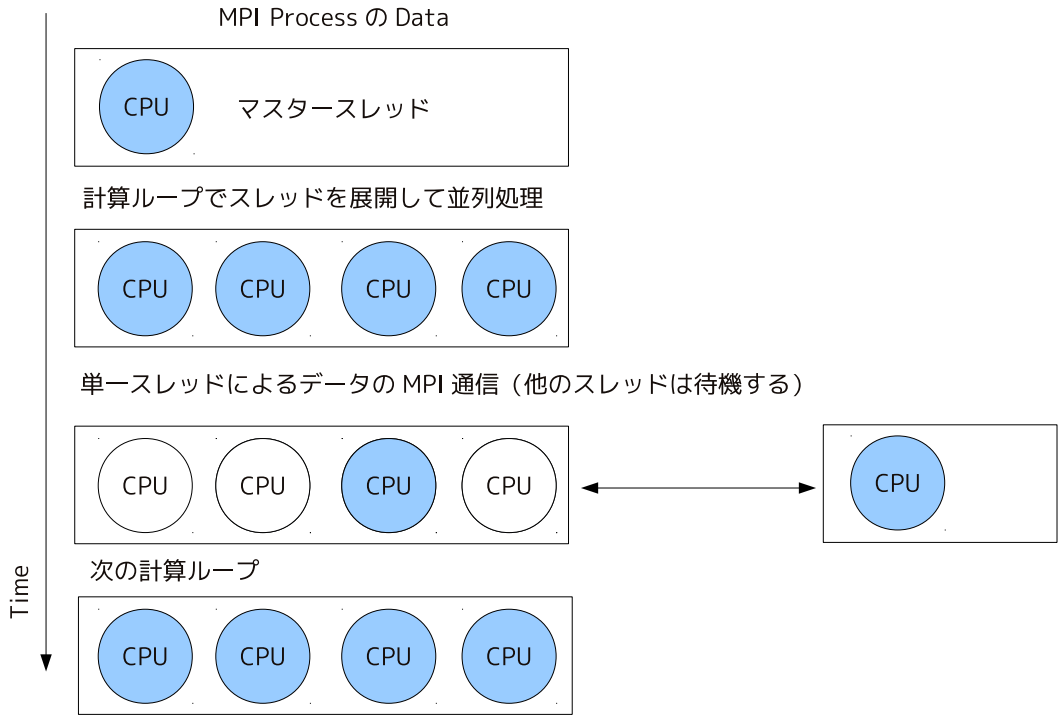

図 3.2: MPI + OpenMP SPMD モデル

- master ディレクティブを使ってマススタースレッドに MPI を実行させる方法.
- single ディレクティブを使って方法

の 2 つの手段があるが，master ディレクティブは同期を行わないため MPI 通信の内容をメモリ上に反映させるために即座に barrier ディレクティブでスレッド間の同期をとらなければならない.

MPI の規格では MPI_THREAD_FUNNELED までのサポートを実装者に義務付けている．これ以上のスレッド安全性はそれぞれの実装者に任せられている.

Cray XT5 で使うことのできる MPI 実装である MPICH では, MPI_THREAD_MULTIPLE までのスレッド安全性の全てをサポートしている。しかしながら、MULTIPLE を指定しては性能低下が起こるおそれがあるため、通常のコンパイルにおいては SERIALIZED までしか使えない。MULTIPLE を有効するには mpich_threadm ライブラリをリンクしなければならない.

本研究での OpenMP SPMD の手法として，OpenMP 並列化ディレクティブ（PARALLEL）をループの前に置き，

にこの手法の概要を示す.

```
call MPI_Init_thread(..)

!$OMP parallel

  ! do computation

  ...


  !$OMP single
  ! do communication
  call MPI_Send()
  call MPI_Recv()
  !$OMP end single

  ! do computation

  ...

!$OMP end parallel
```

第4章　Cray XT5

この章ではターゲットマシンとなる Cray XT5 について述べる．

Cray XT5 は Cray Research Inc. が 2007 年に開発した MIMD 型の大規模並列計算機である．本学における構成では，1 ノードあたりに動作周波数 2.4 GHz の AMD Opeteron Quad core (Shanghai) プロセッサを 2 ソケット持ち，トータルで 8 コアを 16GB のメモリと共に用いることができる．Quadcore Opeteron は 4 コアの各コアローカルにデータ用と命令用に 64KB ずつ，L2 を 512KB，各コア共通の 6MB の L3 キャッシュを持つ．ノード間を結ぶ内部ネットワーク は L3 キャッシュに Hyper Transport 経由で SeaStar2+ チップにより 3 次元トーラス形状に接続されている（図 4.1）．本学においてては合計で 256 ノードを利用可能であり，最大 2048 並列，総メモリ 4TB，理論性能 19.6 TFlops の計算が可能である（表 4）．このように CRAY XT5 は階層的なアーキテクチャを持った並列計算機である．

表 4.1: Cray XT5 のマシン仕様

CPU	AMD Opeteron 2400Hz Quadcore 2sockets per Node
Cache	L1 64KB data + 64KB instruction, L2 512KB per core, L3 shared 6MB per Socket
Memory	16 GB registered ECC DDR2 SDRAM per Node
Interconnect	Cray Seastar2+, 3D torus interconnect
Number of Nodes	256
Compiler	PGI Compiler 8.0.2
MPI Libary	MPICH-2 1.0.6

4.0.3　Hybrid ジョブの実行

Cray XT5 にはコンパイル・デバッグなどに使うログインノードと，計算用に使う計算ノードがある．計算ノードは大規模計算を効率的に行えるようにカスタマイズされた GNU/Linux ベースの OS が使われている．

計算ノードで実行を行うには aprun コマンド用いる．以下のコマンドが Cray XT5 で Hybrid 並列ジョブを実行する場合に使われるものである．-n PROCS でプロセス数を-d

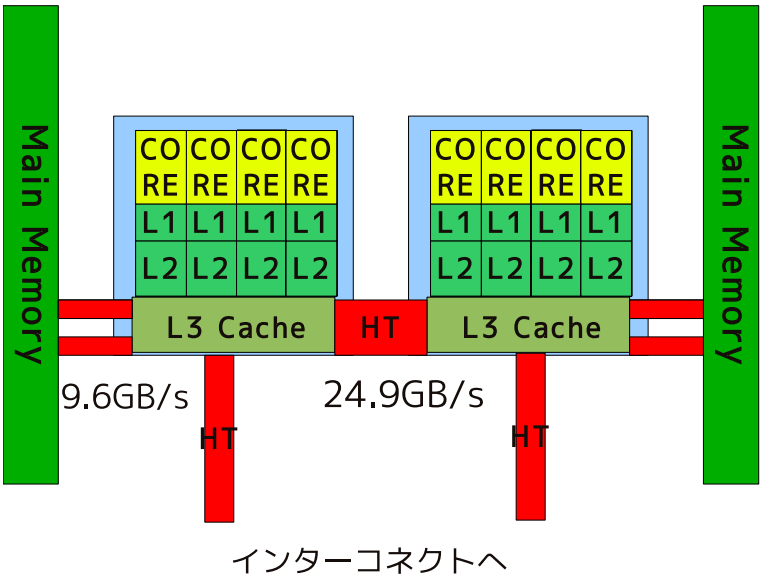

図 4.1: Cray XT5 のノード

THR でスレッド数を指定する．-N オプションはノードあたりのプロセスの数である．全てのプロセッサに 1 プロセスあるいは 1 スレッドを割り当てれば最も CPU 資源を使うことができるため，例えば，1 ノードあたりに 8 スレッドを割り当てたいときは，-d 8 -N 1 とする．-S オプションはソケットあたりのプロセス数を指定できる．

```
export OMP_NUM_THREAD=THR
aprun -n PROCS -d THR -N {1, 2, 4, 8} -S {1,2,4}
```

通常の GNU/Linux をベースとした SMP Cluster では，プロセスに NUMA メモリの割り当てるポリシーを決定するのに numctl を用いるが，NUMA メモリの Cray XT5 には aprun で NUMA メモリの割り当てをサポートしている．

第5章　計算性能ベンチマーク

Cray XT5 の性能を並列計算ベンチマークによって測定し，Cray XT5 のアプリケーション上での実効性能を調査する．ここで用いるベンチマークの計算は並列数値流体アプリケーションで主に扱われる計算を多く含んでいる．

5.1　OpenMP ベンチマーク

ノード内における OpneMP のベンチマークとして，ループサイズに対するスケーラビリティを調べるために Stream benchmark を，OpenMP の各ディレクティブによるオーバーヘッドを調べるために EPCC を用いる．

ここでの計算は PGI コンパイラ 8.0.2 を用い，最適化オプションは-O3 とした．OpenMP が必要な場面では最適化オプションに -mp=nonuma を付加し，OpenMP を有効にして計算した．

5.1.1　Stream Benchmarks

Stream Benchmarks は CPU メモリ間のバンド幅を計測するためのソフトウェアパッケージである [13]．ここでは OpenMP ベンチマークによるスレッド数に対するメモリバンド幅を計測した．Stream Benchmarks においてバンド幅が計測できる演算として，コピー (Copy)，スケール (Scale)，加算 (Add)，三つ組 (Triad) があるが，ここでは最も計算量が多く，かつ数値流体計算コードに現れやすい Triad を用いて計測した．Triad とは数値 C に scalar を掛け，B を加えて，A に格納するという 3 回の倍精度実数型の演算である．

以下が Triad のコード概要である．

```
real(8) :: A(N), B(N), C(N)
t = mysecod()
!$OMP parallel do
do i = 1, N
  A(i) = B(i) + scalar*C(i)
end do
t = mysecond() - t
```

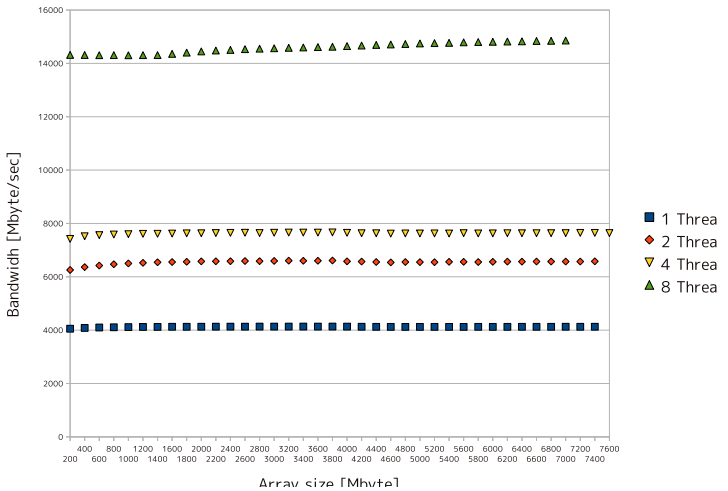

図 5.1: OpenMP Triad

```
throughput = ARRAY * 3 * 8 / t
```

計測において確保メモリのサイズを 200MB から 8GB まで 200MB ずつ変化させた計算を 100 回行って最良値をとった．すべての計測において，100 回の計算における値の分散の最大値は 1 スレッド 8GB のときの 1.27 であった．スレッド数 1, 2, 4, 8 における Cray XT5 の計算結果は図 5.1 のようになる．すべてのスレッド 200MB から 8GB までフラットな結果となる．これは Cray XT5 の OpenMP のループサイズに対するスケーラビリティが高いことを示している．

5.1.2 ECPP Microbenchmarks

ECPP は OpenMP のコンストラクタの違いによるオーバーヘッドを測るためのベンチマーク集である [11].

ここでは図 5.2 に 1 ノードでの EPCC によるベンチマーク結果を示す．ここでは縦軸にスレッド数を 1, 2, 4, 8 と変えて取り，それぞれの OpenMP ディレクティブ (REDUCTION, PARALLEL, PARALLEL DO, DO, BARRIER, SINGLE) による処理のオーバーヘッド時間を図示している．グラフの値の分散はすべて $10^{-}2$ 以下である．

この結果から Cray XT5 上での OpenMP のディレクティブによるオーバヘッドはとくに REDUCTION 構文が最も高いが，これ全てのスレッド間でデータ処理をしていくという REDUCTION 構文の性質を考えれば自然である．次に，PARALLEL 構文と PARALLEL DO 構文のスレッド生成構文があるが，全てのスレッド数で両者同じ値をとる．DO 構文がその半分強のオーバーヘッドしかないことを考えると PARALLEL DO を何度も行ってスレッド生成と消滅を繰り返すよりも，PARALLEL 領域を一度設定し，DO 構文で並列

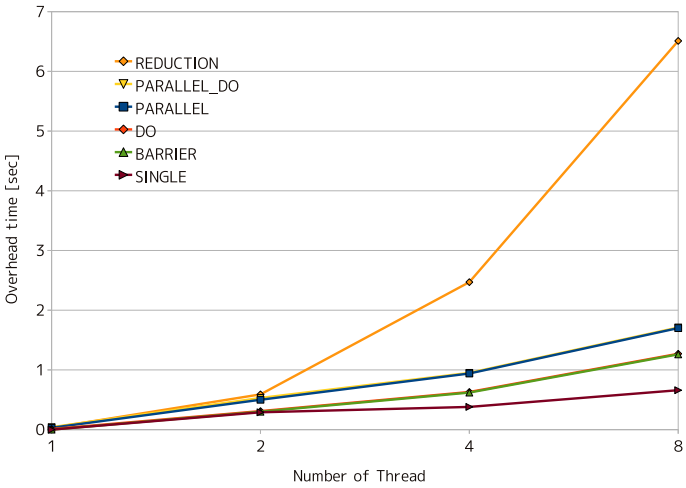

図 5.2: EPCC: OpenMP Overhead

処理を行ったほうが性能を得やすいことを示している．またスレッドの全体の同期を待つ BARRInER 構文よりも単一スレッドで実行してから同期をする SINGLE 構文のほうがオーバーヘッドが少ないのは奇妙である．

5.2 MPI ベンチマーク

Cray XT5 のノード間における MPI ベンチマークを行う．これには Intel MPI benchmark を用いる．

5.2.1 IMB

IMB (Intel MPI Bennchmark) とは Intel の提供している MPI パッケージのベンチマーク集である

ここでメッセージサイズを変えながら 2 つの MPI プロセス間でメッセージの交換を行う Pingpong と Pingpong プログラムの概要を図 5.3 に示す．これは 2 つのプロセスを用いる．長さ x byte のメッセージをこのプロセス間で交換する．プロセス 0 がプロセス 1 に MPI_Send でメッセージを送出し，これをプロセス 1 が MPI_Recv で受信する．メッセージを受け取ったら今度は逆にプロセス 1 がプロセス 0 にメッセージを送信する．プロセス 0 がメッセージを送ってから受け取ったまでの間の時間を t sec として，これを半分に割った値 $t/2$ が一回のメッセージ交換時間となる．したがって，ノード間の Pingpong によるバンド幅は $x/(t/2)$ byte/sec となる．

IMB Pingpong をメッセージサイズを変えてコア間，ソケット間，ノード間で実行した結果を図 5.4 に示す（x 軸は対数グラフになっている）．すぐにわかる通りノード間のバ

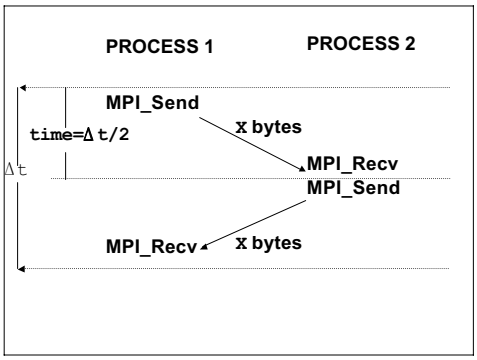

Figure 1: PingPong pattern

図 5.3: IMB Pingpong の概要 []

ンド幅最大が片方向通信バンド幅の理論値 3.2GB/s に対して，約 1.5GB/s と非常に低い．これは Cray XT5 のネットワーク性能が MPI_Send / MPI_Send のような 1 対 1 通信に対してあまりよくないことを示している．

また，ネットワークパフォーマンスで重要なのは通信の立ち上げ時間である．図 5.5 に示すのはノード間，ソケット間，コア間における MPI 通信のレイテンシである．すぐわかるようにソケット間とコア間に比べてレイテンシは非常に大きく，11 倍にも達する．1000 を越えるような高並列になるとロードバランシングが難しくなるのはこのためであり，Hybrid 並列により並列度を落として効率をあげられることがわかる．

5.3 NAS Parallel Benchmark, Multi-Zone Versions

NAS Parallel Benchmark (NPB) [2] は NASA Ames Research Center の NASA Adcanced Supercomputing (NAS) 部門が開発した科学技術計算のベンチマークパッケージである．NPB は実装において MPI，Java, High Performance Fortran, OpenMP などいくつか派生がある．本稿では，そのうちハイブリッド並列化を施したバージョンである Multi-Zone Versions (NPB MZ)[3] を用いる．

NPB MZ には Fortran90 で記述された 3 つのベンチマークアプリケーション LU, SP, BT がある．全て 3 次元空間における非定常圧縮性 Navier-Stokes 方程式を LU は対称 SOR 法，SP と BT は ADI 法を用いて解くものである．計算サイズは小さい順から S, W, A, B, C, D, E, F の 8 つのクラスが用意されている．以下にそれぞれのアプリケーションの概略を示す．

LU 上下三角行列を対称 SOR 法を用いて解く．最大並列度が 16 までなのでハイブリッド並列の効果が現れにくく，本報告では用いない．

SP スカラー 5 重対角行列を ADI 法を用いて解く．

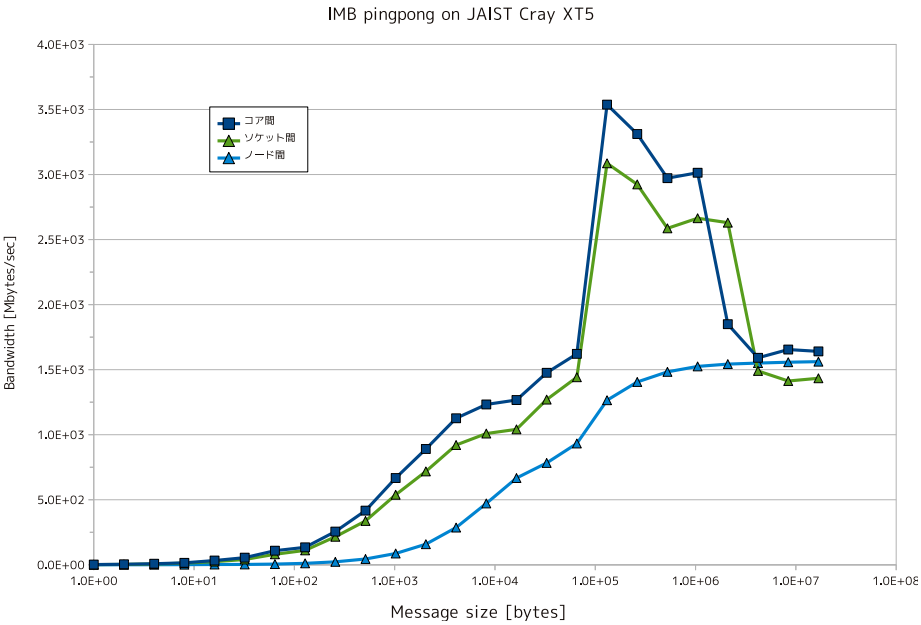

図 5.4: IMB Pingpong の結果

BT ブロック 3 重対角行列を ADI 法を用いて解く．メッシュの分割サイズが一様ではないので負荷分散が難しくなる．

Multi-Zone Version はオリジナルの NPB と比べて並列度の条件が緩やかである．これは並列化の手法がオリジナルと異なっているからである．NPB MZ では計算対象となる 3 次元直方体を x 方向と y 方向に荒く分割し，分割単位を Zone とする 5.6．時間ステップごとに各 Zone に計算を割り当て，得られた境界データを隣接する Zone 間で交換することにより並列化を行っている．このような手法の結果，オリジナルの NPB に比べてメッシュのアスペクト比は異なるが，総メッシュ数はほぼ同じである（クラス S は除く）．

5.3.1 計算条件

NPB MZ のバージョンは 3.2 を用い，コンパイラは PGI Fortran コンパイラがベースにある Cray XT5 のコンパイラ ftn を用い，コンパイラオプションは “-O3 -tp shanghai-64 -r8” で，ハイブリッド並列を有効にする場合は “-mp=nonuma” を付加して OpenMP の宣言を有効にした．

計算サイズはクラス C, D, E とし，ハイブリッド並列を行わない Pure MPI，1 プロセスあたり 4 つスレッドを用いる Hybrid (x4), 1 プロセスあたり 8 つのスレッドを用いる Hybrid (x8) をそれぞれ測定した．

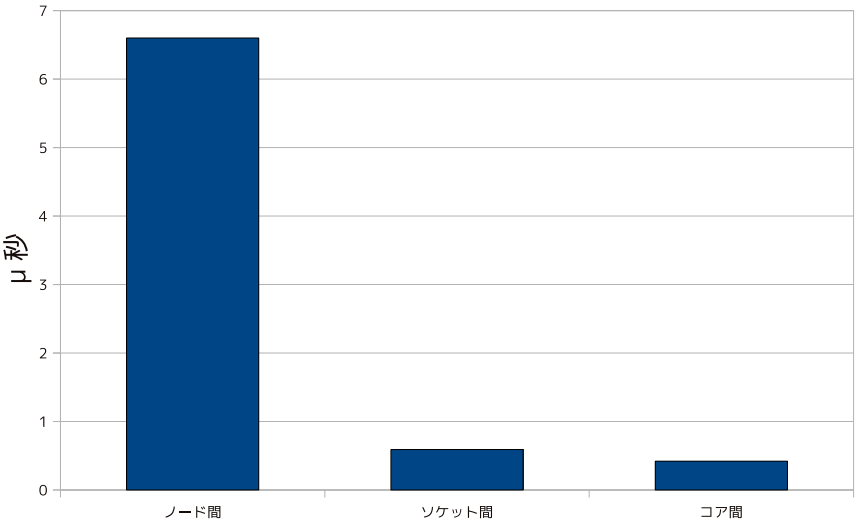

図 5.5: ネットワークレイテンシ

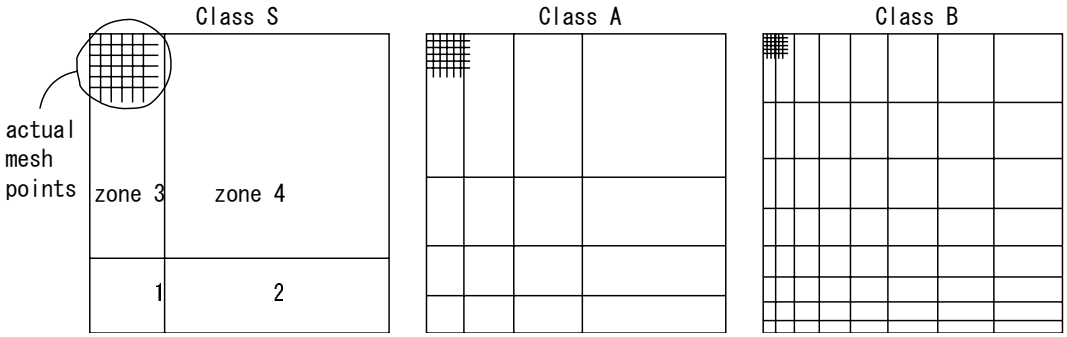

図 5.6: 領域のゾーン分割 [6]

5.3.2 結果と考察

NPB MZ を Cray XT5 上で実行した結果を述べる．グラフの縦軸は経過時間 [sec] ，横軸はコア数であり，プロセス数とスレッド数の積で表されている．

計算クラスのメッシュサイズは C が $480 \times 320 \times 28$ の 総必要メモリ約 800MB，D が $1632 \times 1216 \times 34$ の約 12.8GB, E が $4224 \times 3456 \times 92$ の約 251GB となっている．また，NPB のプロファイラ機能を用い，それぞれの計算プロセスにおける通信時間と計算時間を区別し，もっとも待機時間の長いプロセス，すなわち全体の実行時間とほぼ等しいプロセスを表示した．ここでの待機時間とは領域分割の境界値交換を行うルーチン exch_qbc.f 中の MPI_Isend/MPI_Irecv を呼び出しから MPI_Wait によるメッセージ通信の終了を待つ時間のことである．

アプリケーション SP の結果はクラス C 図 5.7, クラス D 図 5.8, クラス E 図 5.9 にそれぞれなった．クラス D を除いて， PE の数に関わらず，Pure 並列は常に Hybrid 並列よりも高い性能を出している．並列度が上がってもこの傾向は変わらない．アプリケーション SP は格子メッシュを均等に分割する問題であり，MPI 通信のオーバーヘッドは並列度が上がってもほとんど高くない．図 5.10 はクラス E の 2048 並列時における通信時間を各ノードごとにとったものであるが， ほぼ均一であり (低いところと高いところに 2 極化しているのはノード間通信とノード内通信の性能差であると考えられる)，Hybrid 並列による通信時間低減の効果はほとんどないため Pure 並列のほうが良い性能を出している．クラス D でのみ Hybrid 並列が優れているのはこの問題規模の場合，最適化がうまく効き，OpenMP の並列化が高速化したためではないかと思われる．コンパイラの最適化フラグを低いものにしたところ，クラス D もその他のクラスと同様に常に Pure 並列のほうが早くなった．

アプリケーション BT の結果はクラス C 図 5.7, クラス D 図 5.8, クラス E 図 5.9 にそれぞれなった．全ての並列化において並列度が上がるとともに性能は落ちていく．128 並列までは Pure と Hybrid の性能差はほとんど見られず，わずかにピュア並列のほうがよい．しかし，512 並列以上になるとピュア並列は大きく性能を下げるのに対し，ハイブリッド並列は 1024 並列でも高い並列化性能を保っている．興味深いのは 2048 並列の場合

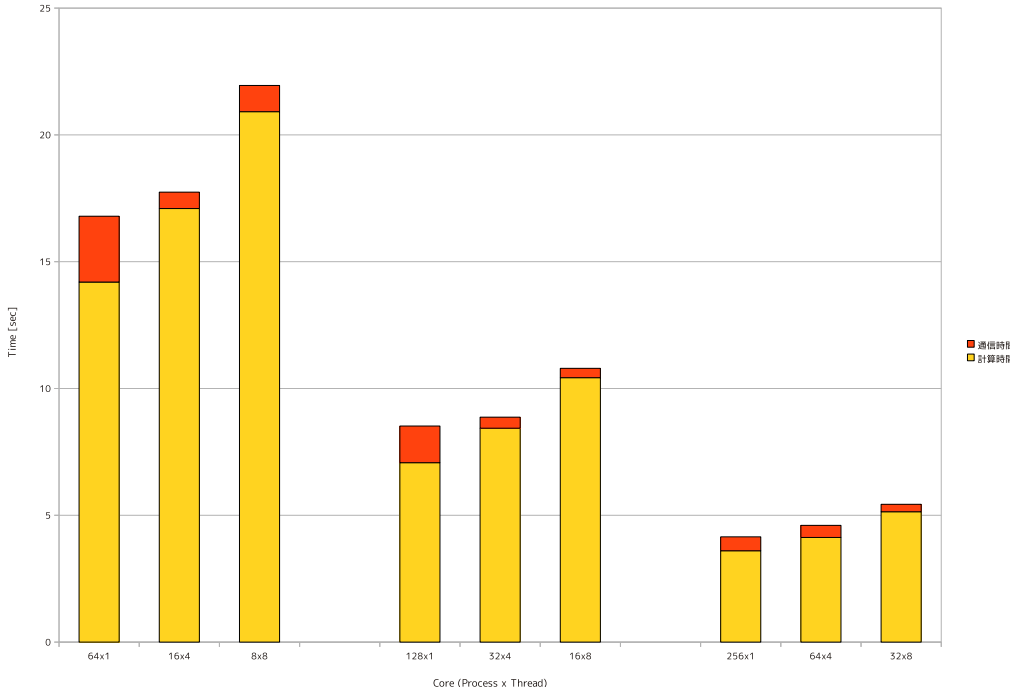

図 5.7: SP (CLASS C) における計算時間と通信時間

である．計算クラス D においては MPI のみで 2048 並列の計算を行うことはできない．したがって，この並列度ではハイブリッド並列のみの結果しかない．このとき 1024 並列までは緩やかに性能を保っていた 4 スレッドハイブリッド並列の計算は 2048 並列で大きく性能を落としている．

以上から，Hybrid 並列モデルにおいてはノードあたり 4 コアのソケット内並列を行うものが最も効率が高いと考えられる．

5.4 まとめ

以上で得られたベンチマークの結果をまとめる．Cray XT5 はノード内での OpenMP の性能がよく，大規模なメモリに対しても高い並列性能を維持できる．一方，ノード間の MPI 通信性能はあまり高くない．また，ノード間，コア間の MPI レイテンシは 11 倍も差があり，大規模並列になった場合これが大きなオーバーヘッドにになる可能性がある．

NPB ベンチマークににおいてはアプリケーション BT のような，ロードバランシングが悪く並列度が上がると通信時間過多になるような問題では Hybrid 並列は非常に有用であることがわかる．さらに，アプリケーション SP のようなロードバランシングがよく高並列時に通信時間過多にならないものであっても，クラス D の結果のように特定の問題規模によって OpenMP の並列化効率があがり Hybrid 並列が Pure 並列よりも優れる場

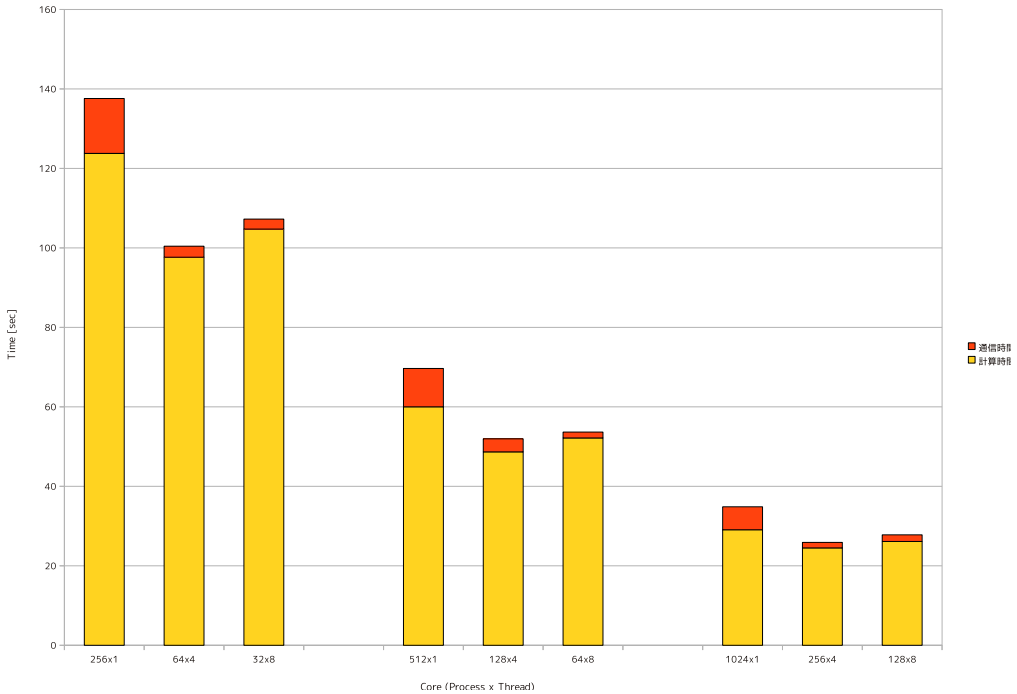

図 5.8: SP (CLASS D) における計算時間と通信時間

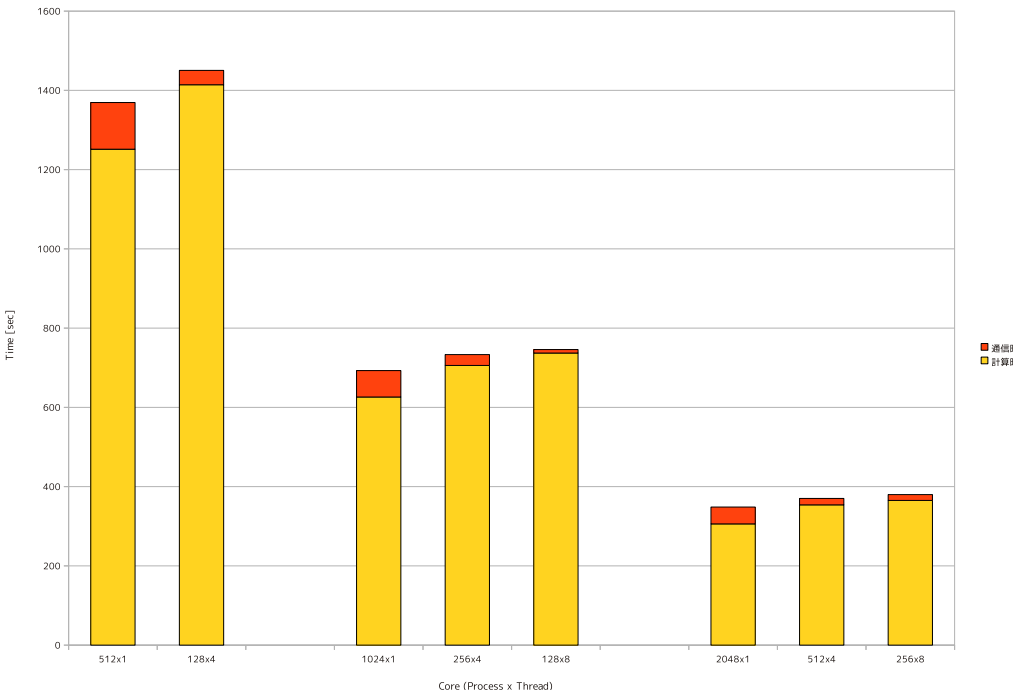

図 5.9: SP (CLASS E) における計算時間と通信時間

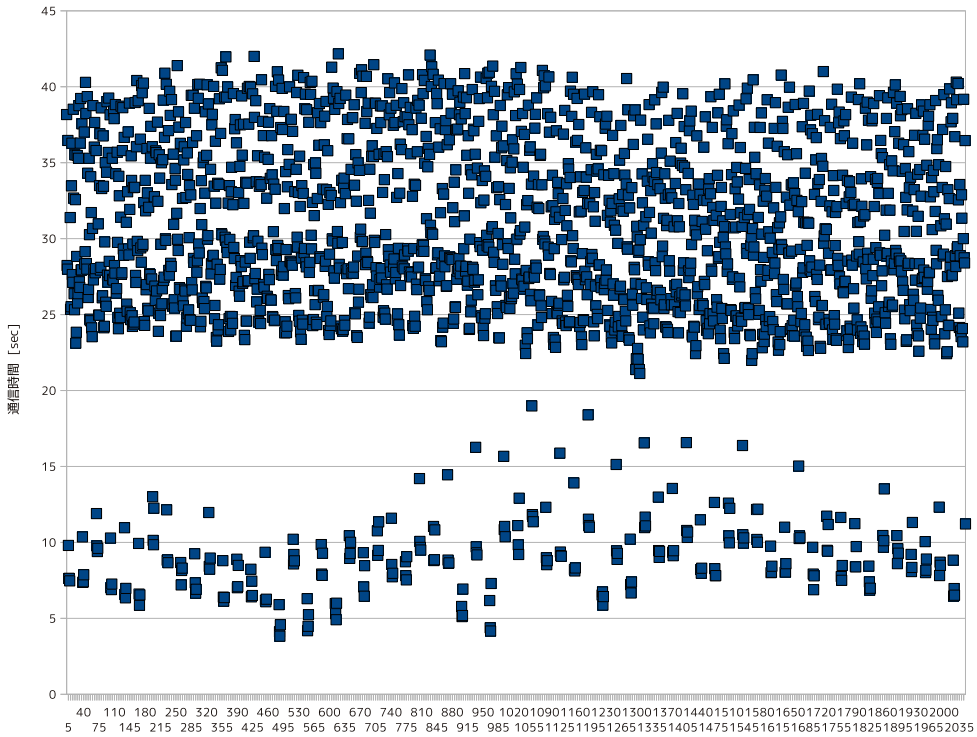

図 5.10: SP (CLASS E) 2048 並列における各ノードの通信時間

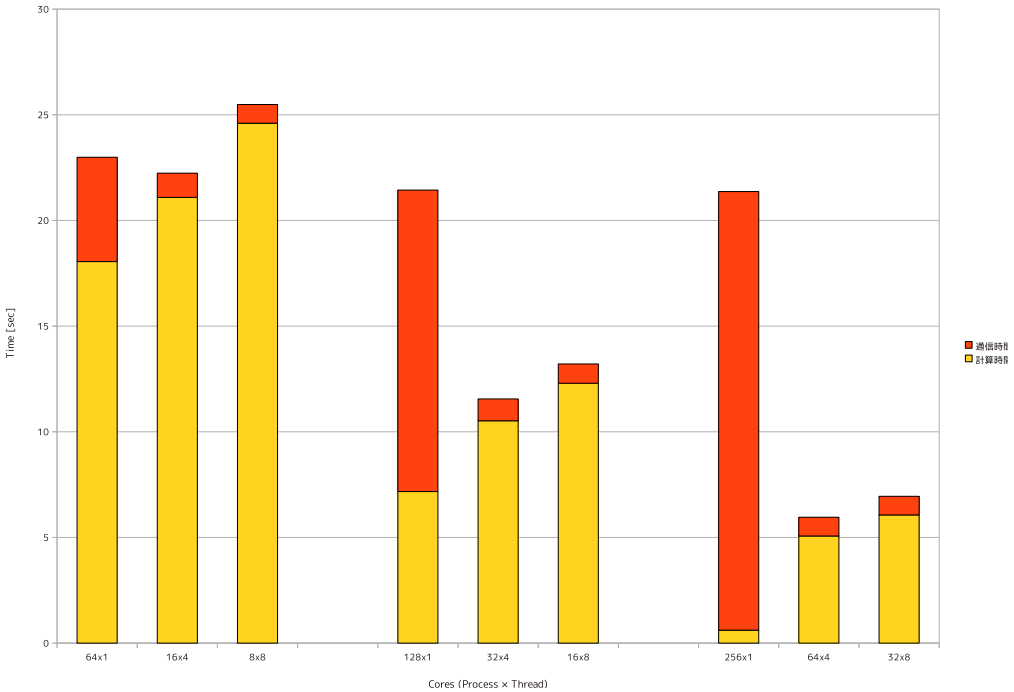

図 5.11: BT (CLASS C) における計算時間と通信時間

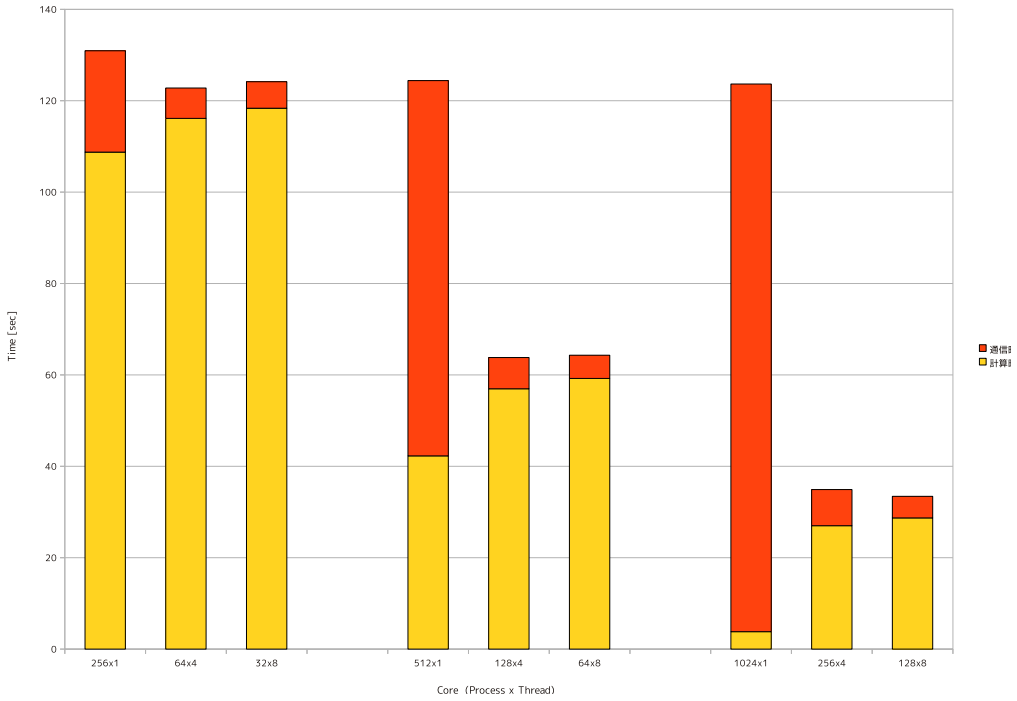

図 5.12: BT (CLASS D) における計算時間と通信時間

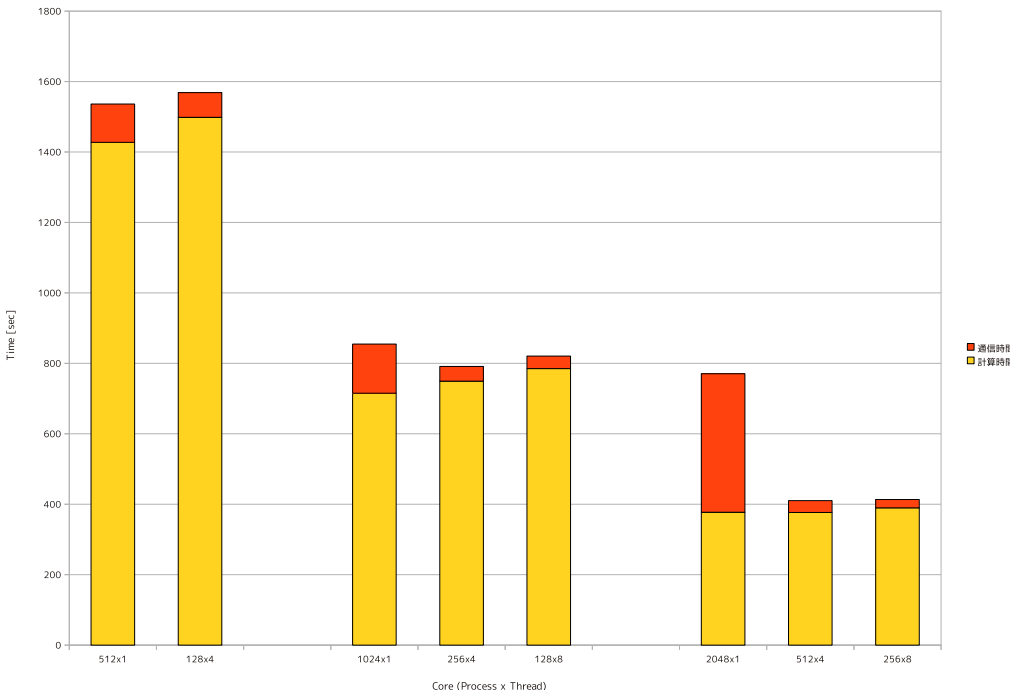

図 5.13: BT (CLASS E) における計算時間と通信時間

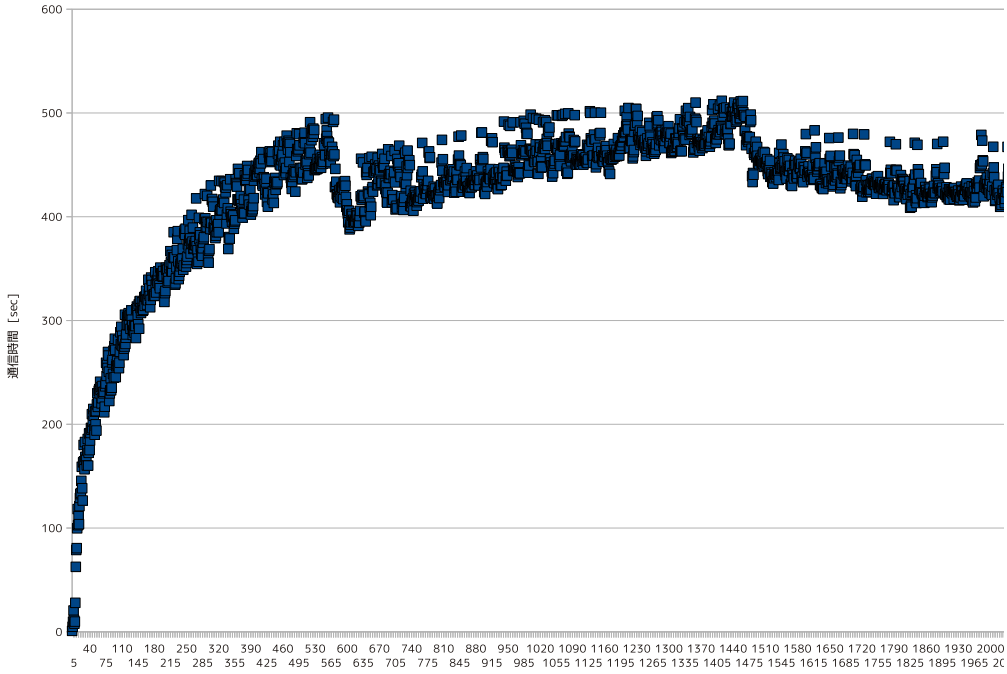

図 5.14: BT (CLASS E) 2048 並列における各ノードの通信時間

合があった.

NPB の全ての結果において Hybrid 化してプロセス数を減らした場合，通信時間は減少した．したがって，OpenMP による高速化率がプロセス数の減少よりも大きい場合，Hybrid 並列は Pure 並列よりも高い性能が出せるはずである．OpenMP の並列化効率を妨げるものは OpenMP ディレクティブのオーバーヘッドがある．EPCC ベンチマークの結果より，PARARELL DO を何度も使うよりも PARARELL 構文の中で DO 構文を用いて OpenMP SPMD のように処理するものが効率的であると考えられる.

また，Hybrid 並列のスレッド数コア間スレッドを生成する 4 スレッド Hybrid（チップ内 Hybrid）が最も性能がよいと考えられる.

第6章　Hybrid並列数値流体アプリケーション

実際の数値流体アプリケーションでは MAC 法や SMAC 法といったスキームによりラプラス方程式のソルバが非常によく使われる．

ここでは Hybrid 並列化された数値流体並列アプリケーションとして，ラプラス方程式を対象としたポテンシャル問題を有限要素法で離散化し，得られた線型方程式を Hybrid 並列化された並列共役勾配法ソルバで解く．アプリケーションを Cray XT5 で実行し，結果を調べた．なお有限要素解析コードとして，[5] のコードをベースとして使用しており，Cray XT5 上で大規模領域において動作するように動的メモリを使用し，任意の領域において実行可能かつ Hybrid 並列で動作するように構築した．

6.1　対象問題

本アプリケーションが対象とする問題は 2 次元ポテンシャル問題である．ここで行われる計算は原点を左下にとった $\Omega = [0, 1]$ の正方領域上でのスカラー ϕ に対するポテンシャル問題は以下のようになる．

$$\nabla^2 \phi = \frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial^2 y^2} = 0$$

境界条件として，底面 $(y = 0)$ に $\phi(x, 0) = sin(\pi x)$ を与え，それ以外の境界面では自然境界条件

$$\left.\frac{\partial \phi}{\partial n}\right|_{\Omega} = 0$$

を設定する．

なおこの問題には解析解があり，検証が可能である．この問題の解析解は

$$\phi(x, y) = \frac{\sin(\pi x)\sinh(\pi(1 - y))}{\sinh(\pi)}$$

となる．

6.2 有限要素法

この節では方程式を離散化する方法について解説する．一般に偏微分方程式を計算機で直接解くことはできない．その代わり，問題領域を有限個の格子点で分割し，その上で代数方程式を構成し近似的に解くことになる．ここでは有限要素法 (Finite Element Method) を用いて方程式を離散化する．

6.2.1 重み付き残差法

与えられた方程式の近似解を ϕ とする．

残差 $\nabla^2\phi$ に対して，重み関数を ψ として計算領域 Ω 上積分し，残差が 0 になればよい．

$$\int_\Omega \psi \nabla\phi d\Omega = 0 \tag{6.1}$$

グリーンの公式を用いて変形すると

$$\oint_{\partial\Omega} \psi \frac{\partial\phi}{\partial n} dl - \int_\Omega \nabla\psi\nabla\psi\nabla\phi d\Omega = 0. \tag{6.2}$$

ディリクレ境界 Γ_D 上で $\psi = 0$ と仮定すると，

$$\oint_{\Gamma_N} \psi u_n dl - \int_\Omega \nabla\psi\nabla\phi d\Omega = 0 \tag{6.3}$$

となる．この方程式を離散化する．

計算領域を複数の三角形に分割し，この三角形上で ϕ, ψ が線形に分布していると仮定して離散化を行う．

6.3 共役勾配法

得られた連立一次方程式を解く．このような線型方程式の解法としては，Gauss の消去法や Skyline 法のような直接解法，あるいは Jacobi 法や逐次過緩和法 (Succesive Over Relaxation Method) ，共役勾配法のような反復解法がある．反復解法は直接解法に比べて，一般に計算領域を非常に節約できるという利点があり，また多くの場合，非常に速く収束する．このため現在の大規模数値解析では反復解法が多く用いられている．

本研究では連立一次方程式のソルバとして前処理のない共役勾配法を用いる．

6.3.1 原理

共役勾配法の原理を以下に示す.

求めたい線形方程式を

$$A\mathbf{x} = \mathbf{b}$$

とする．A を対称正定値行列であると仮定する．ここで,

$$\phi(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T Ax - x^T\mathbf{b}$$

となる $\phi(\mathbf{x})$ を定義すると，$\nabla\phi(\mathbf{x}) = A\mathbf{x} - \mathbf{x}$ となる.

線形方程式の解を求める問題は，関数 ϕ の最小値を探す問題に還元される．なぜなら，$\hat{\mathbf{x}}$ が方程式の解であったとき，$\mathbf{x} = \hat{\mathbf{x}} + \mathbf{h}$ での ϕ の値は

$$\phi(\mathbf{x}) = \phi(\hat{\mathbf{x}}) + \mathbf{h}^T(A\hat{\mathbf{x}} - \mathbf{b}) + \frac{1}{2}\mathbf{h}^T A\mathbf{h}$$

となる．第 2 項は $\hat{\mathbf{x}}$ が解であるから $\mathbf{0}$ であり，第 3 項は A が正定値であることから非負となる．したがって，任意の $\mathbf{h}$ に対して，$\phi(\hat{\mathbf{x}} + \mathbf{h}) \geq \phi(\hat{\mathbf{x}})$ となる.

$\phi(x)$ の最小値を $x = x_0$ から始めて，探索する．このとき，得られた解の候補から，その全ての候補に対して共役となるベクトルを選ぶことで高速な解の収束が可能となっている.

以下に共役勾配法の擬似コードによるアルゴリズムを示す.

$\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$
$\mathbf{p}_0 = \mathbf{r}_0$
do k = 0, kmax
    $\mathbf{q}_k = A\mathbf{p}_k$
    $\alpha = (\mathbf{r}_k, \mathbf{r}_k)$ / (p_k, q_k)
    $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k p_k$
    $\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k s_k$
    $rtr = (\mathbf{r}_{k+1}, \mathbf{r}_{k+1})$
    if $(rtr < \varepsilon)$ exit
    $\beta_k = (\mathbf{r}_{k+1}, \mathbf{r}_{k+1})$ / $(\mathbf{r}_k, \mathbf{r}_k)$
    $\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \beta_k p_k$
end do

6.4 並列化

この節では得られた線形ソルバを並列化する方法について述べる.

6.4.1 領域分割

数値流体計算においては反復によって計算を解く場合，緩和法などで大規模問題の並列処理するために計算領域を分割し，それぞれを独立に計算することで問題を処理する手法が一般的である．このとき領域間データの依存性はある程度無視されるため，並列計算で処理する場合と逐次計算で処理する場合の反復回数は，並列計算を行った方が多くなる．

分割された領域を適切に分割する．分割の方法は単純な短冊状の分割も可能であるが，計算領域全体が複雑な形状をしている場合，単純に分割したときの分割領域間のサイズがまちまちになってしまい性能低下の原因となることがある．

今回の計算においてはメッシュの分割に自動分割エンジンであるミネソタ大学の開発したMETISを用いる[9]．分割領域間に，のりしろとなる領域としてゴーストノードを設定し，計算ループを回す度に，ゴーストノードと領域境界点の値をコピーし，領域ごとに新たな境界値を設定する．ゴーストノードの境界点との値のコピーはMPIの2点間通信を用いて行われる．本研究のコードでは，MPI_Isend, MPI_Irecv と MPI_Waitall の非ブロッキング通信を用い，通信時間とMPIメッセージ送出バッファを構築する計算時間の重ね合わせが行われている．

緩和法のループ完了を判定する際の誤差の値は各領域間で最も大きい値を用いて判断せねばならない．なぜならある領域での誤差がループを脱出する許容値より低かったとしても，別の領域ではそうとは限らないためである．このような計算にはMPIの集合通信であるMPI_Allreduceを用いて誤差の値をすべての領域上で同期させる．

MPI通信による並列化はコード上に概要で示すと次のようになる．

$\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$
$\mathbf{p}_0 = \mathbf{r}_0$
do k = 0, kmax
　$\mathbf{q}_k = A\mathbf{p}_k$
　!ゴーストノードにおける q_k の値の交換
　$\alpha = (\mathbf{r}_k, \mathbf{r}_k) \ / \ (p_k, q_k)$
　!MPI領域間で α の値をAllreduceで足し合わせこれを新しいalphaの値とする
　$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k p_k$
　$\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k s_k$
　$rtr = (\mathbf{r}_{k+1}, \mathbf{r}_{k+1})$
　!MPI領域間で rtr の値をAllreduceで足し合わせこれを新しいrtrの値とする
　if $(rtr < \varepsilon)$ exit
　$\beta_k = (\mathbf{r}_{k+1}, \mathbf{r}_{k+1}) \ / \ (\mathbf{r}_k, \mathbf{r}_k)$
　$\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \beta_k p_k$
end do

6.4.2 ループレベル分割

プログラム中の DO ループを OpenMP を使って並列化する．

OpenMP での並列化の手法はプログラム中に並列化ディレクティブを挿入することである．CG 法のアルゴリズムには初期化と内積計算の部分がそうであるが，第 2 章で述べた 2 つのプログラミングモデルによって並列化部分が異なる．これらは以下のように表現できる．

Hybrid マスターオンリーモデル CG 法

```
...
do k = 0, kmax
    !$OMP parallel do
    !ここでスレッドを開く
    q_k = A p_k
    ! ここでスレッドは閉じられる
    !ゴーストノードにおける q_k の値の交換
    !$OMP parallel do
    ! スレッドを開く
    α = (r_k, r_k) / (p_k, q_k)
    ! ここでスレッドは閉じられる
    ...
end do
```

HybridMPI+OpenMP SPMD モデル

```
...
!$OMP parallel do
! 反復部分全体に渡ってスレッドを生成する
....
do k = 0, kmax
    !$OMP do
    !各々のスレッドが分割されたループを処理する
    q_k = A p_k
    !$OMP end do
    !$OMP single
    ! MPI 呼び出しの前に同期してから単一スレッドで実行
    ゴーストノードにおける q_k の値の交換 (MPI 通信)
    !$OMP end single
    !再び複数スレッドで実行
    ...
end do
!$OMP end parallel
!ここでスレッドを閉じる
```

6.5 結果と考察

この問題の計算結果を図示 6.1 する．Pure 並列での結果と Hybrid 並列の結果は全ての節点で 10^{-4} の範囲で一致しており，この Hybrid 化実装による解の誤りはないと考えられる．

数値計算は以下の条件で行った．2 つのサイズ SMALL (1024 節点) と LARGE (2048 節点) について，Pure 並列，コア間 Hybrid 並列をマスターオンリーモデルと MPI+OpenMP SPMD モデルでベンチマークを 3 回実行し最良値を比較した．

実験条件

1. SMALL 1024^2 節点領域
2. LARGE 2048^2 節点領域
3. PGI Fortran 8.0.2
4. 最適化オプション -O3 （OpenMP が必要な場合 -mp=nonuma を付加）
5. 反復終了条件：残差ベクトルのノルムが 10^{-12} 以下

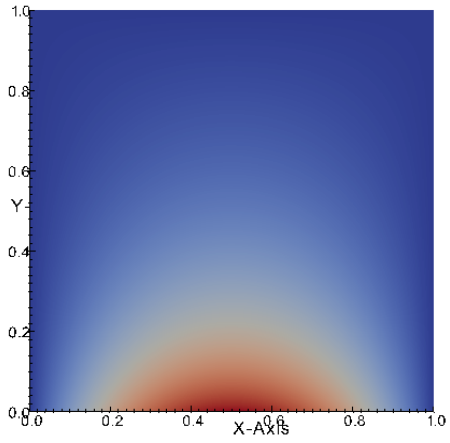

図 6.1: ポテンシャル問題

SMALL での結果は図 6.3 になる．並列度が上がると Hybrid 並列が優勢になり 256 並列では Pure MPI 並列に勝っている．これは小さいサイズの問題であるために並列度があがると通信過多となり Hybrid 並列が並列度を落とせたからであると考えられる．また, MPI+OpenMP SMPD モデルはこの場合もっとも優れた結果を出している．

LARGE での結果は図 6.4 になる．これは全ての並列度で Pure 並列が勝っているが並列度が上がるにつれてその差は縮まっている．

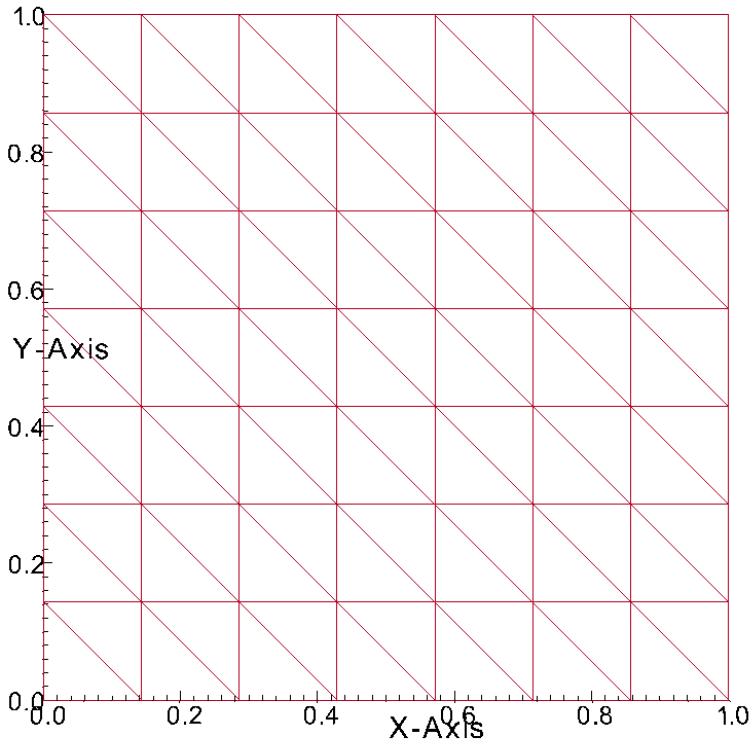

図 6.2: 計算領域の要素分割

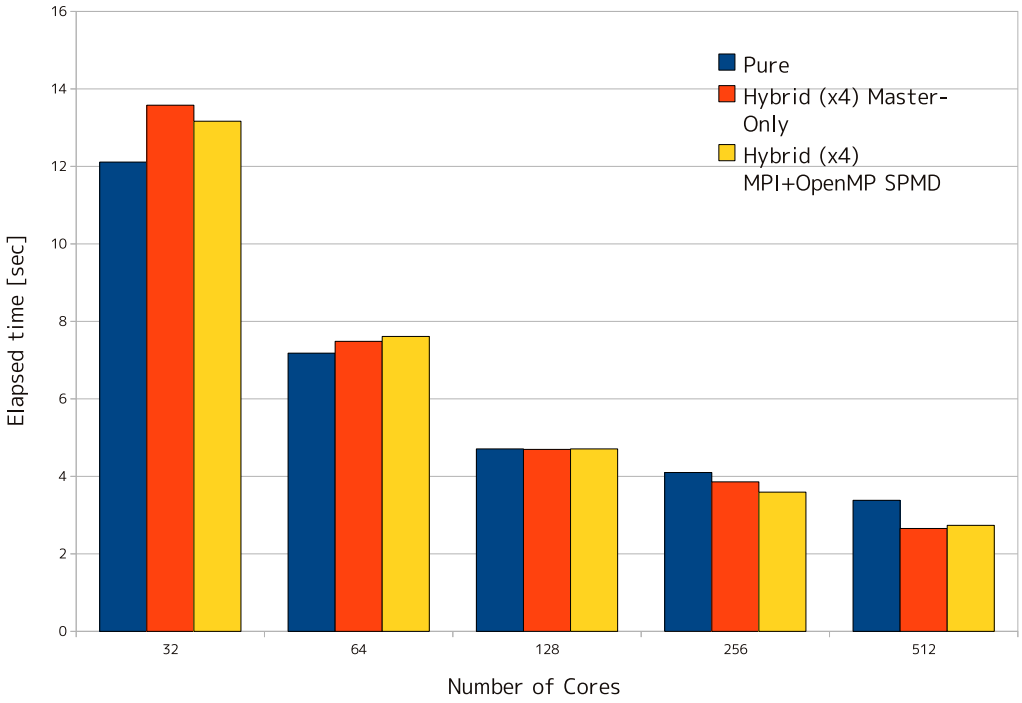

図 6.3: SMALL (1024^2 節点) における計算結果

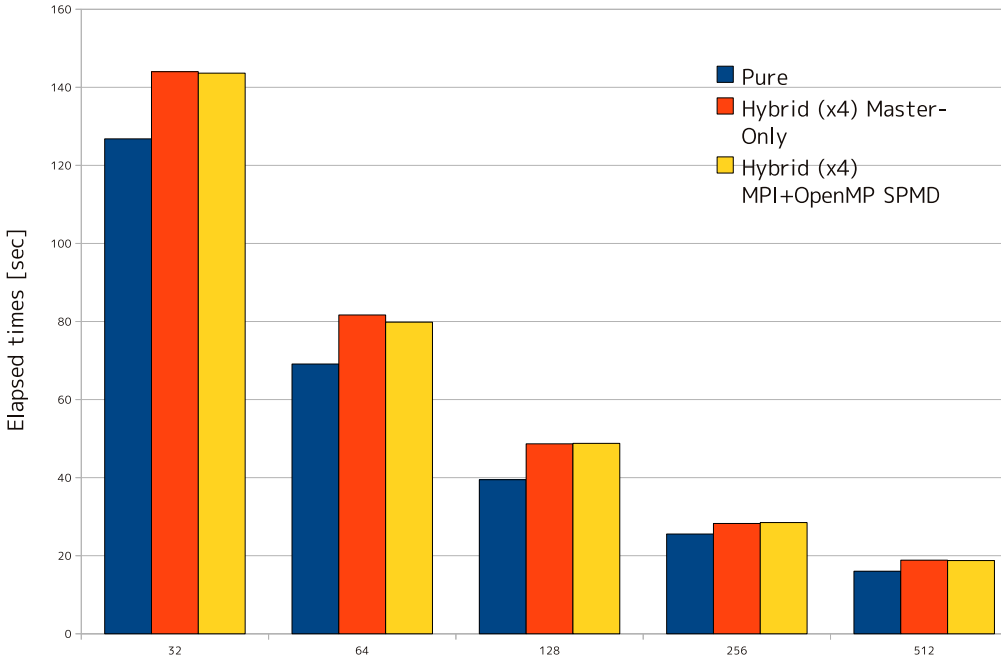

図 6.4: LARGE (2048^2 節点) における計算結果

第7章　結論

7.1　結論

本稿では, SMP クラスタマシン上で有効であると言われている Hybrid 並列について, 大規模な SMP クラスタマシンである Cray XT5 の性能を調査し，Hybrid 並列の有効性を調べ，その結果，MPI+OpenMP SPMD モデルの Hybrid 並列化手法を用い，反復法により有限要素法アプリケーションを構築し，性能を調査した.

その結果，以下のことがわかった.

- Hybrid 並列が Pure 並列よりも性能が高くなるのは，高並列時に負荷分散がうまくいかずアプリケーションの通信時間が計算時間よりも大きくなる場合であった.
- 2 次元ポテンシャルソルバを Hybrid 並列化した数値流体アプリケーションとして用い，Hybrid 並列プログラミングのマスターオンリーモデルと MPI+OpenMP SPMD を比較した結果，共に Pure 並列と比べて同程度からやや劣った結果を得たが，並列度が上がると Hybrid 並列との差がなくなっていくという結果を得た.

7.2　今後の課題

- ccNUMA アーキテクチャにおける OpenMP の高速化を行う手法として，赤黒 SOR 法のようにループをデータ依存性のないように分割して，スレッドデータの局所性をあげる手法が知られている．これを利用すると Hybrid 並列の効率を高められる可能性がある.
- 現在の OpenMP における共有メモリプログラミングでは共有メモリのデータ同期に単純なバリアしか使えない．スレッドを用いた共有メモリ上での 1 対 1 同期を行うことで OpenMP 並列の性能が向上すると考えられる.

謝辞

研究を常にご指導いただいた松澤照男教授に深く感謝します．適切な助言で研究に協力していただいたクレイ・ジャパン・インクの木下武志様，西村成司様に感謝します．研究のための計算機資源の提供とサポートをしていたいだた JAIST 計算機センターの皆様に感謝します．また，松澤研究室の方々には多くの指導と助言をいただきました．感謝します．最後に，大学院での研究生活を支えてくれた私の家族に感謝します．

参考文献

[1] Franck Cappllo, Daniel Etiemble “MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks”, Supercomputing ’00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM), IEEE Computer Society, Washington, DC, USA, 2000

[2] D. Bailey, E.Barscz, J. Barton, D.Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederikson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, S. Weeratunga. “The NAS Parallel Benchmarks” NAS Technical Report RNR-94-007, NASA Ames Research Center, Moffett Field, CA, 1994

[3] R.F. Van Der Wijngaart, H. Jin. “NAS parallel benchmarks, multizone versions”, NAS Technical Report NAS-03-010, NASA Ames Research Center, Moffett Field, CA, 2003.

[4] 牛島 省. “OpenMP による並列プログラミングと数値計算法”, 丸善, 2006

[5] 樫山 和男, 牛島 省, 西村 直志, 日本計算工学会編. “並列計算法入門”, 丸善, 2003

[6] Haoqiang Jin, Rob F. Van der Wijngaart. “Performance Characteristics of the Multi-Zone NAS Parallel Benchmarks.” ipdps, vol. 1, pp.6b, 18th International Parallel and Distributed Processing Symposium (IPDPS’04) - Papers, 2004

[7] Cluster OpenMP* for Intel Compilers. `http://software.intel.com/en-us/articles/cluster-openmp-for-intel-compilers/`

[8] Georg Hager, Gabriele Jost, Rolf Rabenseiner. “Communication Characteristics and Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-core SMP Nodes,” Cray User Group 2009 Proceedings, 2009

[9] METIS - Family of Multilevel Partitioning Algorithms. `http://glaros.dtc.umn.edu/gkhome/views/metis`

[10] 中尾哲也, “マルチコアクラスタ向けチップレベルハイブリッド並列化に関する研究”, 北陸先端科学技術大学院大学 修士論文, 2009

[11] J.M. Bull, D. O'Neill. "A Microbenchmark Suite for OpenMP 2.0." In Proceedings of the Third European Workshop on OpenMP (EWOMP'01), 2001.

[12] Motoi Okuda. "Roadmaps and visions II - Fujitsu's vision for high performance computing," Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing, November 11-17, 2006 ACM, 2006

[13] John D. McCalpin. "Memory Bandwidth and Machine Balance in Current High Performance Computers," IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, December 1995.