修 士 論 文

共有メモリ型並列計算機による
流体問題解析の並列計算

指導教官　松澤 照男 教授

北陸先端科学技術大学院大学
情報科学研究科 情報システム学専攻

黒川 原佳

1999 年 2 月 15 日

要 旨

本稿では、共有メモリ型並列計算機において、効率的な数値計算を行うための方法と実装方法を提案し、実際問題特に流体問題解析にそれらの方法を適用し、本稿において提案する方法の効率を検討する。

目 次

第 1 章

はじめに

1.1 研究の背景

近年共有メモリの並列計算機[1]、特に SMP($SymmetricMulti-Processor$) 型の計算機[2]の普及は著しい。特にシステムの構成が容易な小規模及び中規模なシステムは、目を見張るものがある。小規模な SMP システムの構成は、メモリ-PE 間をバス接続することで、並列計算機システムの中では比較的容易に作成できる。そのため、PC や WS などの低価格製品でも製作が可能となる。また中規模なシステムでは、安易にバス接続することで、バスボトルネックが発生するが、メモリ-PE 間の接続をバスではなく、クロスバースイッチに置き換えることでこの問題は改善されるが、製作コストは多少上昇する。ただ現状では、メモリアクセス速度などの問題から、大規模な並列システムの構成が困難である。

しかし、ハード ウェアの普及は目覚しいものの、ユーザーレベルにおける効率的なプログラム作成の方法は必ずしも確立されていない。SMP などの共有メモリシステムには、コンパイラが自動的にプログラムコード を解析し、分割可能なループを判断してもっとも単純な並列化を行なう自動並列化コンパイラが備わっている場合が多い。これは、ループを分割し並列化を行なうという初期的な並列手法で、ループ内にデータの依存関係が存在する場合、自動的には並列化が行なわれない。そのため、ある程度ユーザー自身が並列化を行なう必要がある。通常並列システムにおいては、MPI(Message Passing Interface) など通信ライブラリを用いる並列計算プログラムの方法が確立されつつあるが、SMP システムなどの共有メモリシステムにおいては通信をする必要がない。そのため共有メモリシ

[1]以後共有メモリシステムとする。
[2]以後 SMP システムとする。

ステムでも様々なプログラムモデルが提案されている。その一つがスレッドであるが、まだ、共有メモリ並列プログラムのスタンダードとなる方法として確立されていない。

共有メモリシステムにおいて共有メモリプログラムモデルを用いる利点は多い。MPIなどの通信ライブラリでは、通信とデータインデックスを考える必要があり、プログラミングを考えた場合非常に面倒である。共有メモリシステムでは共有データへのアクセスの同期を考えるのみである。分散メモリ型並列計算機[3]では、通信においてデータアクセスの同期を取っていることと同義である。MPIなど通信ライブラリを用いる通信とメモリアクセス同期だけのコストを較べると、同期の方がコストが小さい。また、プログラミングも通信とインデックスの両方を考えるより、同期だけを考える方が容易である。以上のように、共有メモリシステムでのプログラミングは、分散メモリシステムに比べ非常に容易であると考えられる。

また、同期にも色々な方法が考えられる。使用するプロセッサ全体で同期を取るバリア同期が一般的である。しかし、バリア同期ではプロセッサ全体での同期であるため同期のコストがプロセッサ数が増えるにつれて高くなる。本稿では、データに依存関係を持つ場合、データアクセスに対して同期を取る局所的な同期を提案する。この場合同期は1対1であるため、プロセッサの増加に比例して増えることはない。

1.2 研究目的

以上のように、普及が著しい高性能な共有メモリ型の計算機を数値計算特に数値流体力学 (CFD : Computational Fluid Dynamics) のような科学技術計算に有効に利用したいと考えた場合、現状のままでは利用し易く、計算効率の高い並列計算機とは言えない状況である。

数値計算の分野において、楕円型方程式問題に関する解法として一般的に用いられるのは、SOR法やCG法などの緩和法であるが、分散メモリシステムでは、データ依存関係を多少無視し、計算領域を分割してしまいある程度独立に計算を進めることで対処しているが、逐次計算より反復回数が増大する。

データ依存性が強い楕円型方程式問題や負荷量が計算領域内でまちまちであるような問題に、分散メモリシステムを用いることは、通信時間が多くなり高並列時の弊害となっているが、共有メモリシステムとスレッドによる同期を用いるプログラムでは、これらの問題を容易に解決できるため、利用する利点は多いと考えられる。

[3]以後分散メモリシステムとする。

本稿では、共有メモリシステムでの数値解析、特に流体問題解析での利用方法を示し、ハードウェアの普及に見合あった効率的な利用方法を提案する。そして、その実装と実際問題への適用を行なう。

第 2 章

Symmetric Multi-Processor System

2.1 Shared Memory Model

2.1.1 UMA

共有メモリシステムのメモリモデルは、大分して二種に分類される。まず一つは UMA(Uniform Memory Access Model : 均一メモリアクセスモデル) である。図 2.1 に示す通り、全ての PE(Processing Element) [1]からすべてのメモリへアクセスできる。 すなわち、すべての PE は共通のメモリアドレス空間を持ち、全てのアドレスに対して同一時間でアクセスできる。

この方法は、既存の逐次システムの延長線上と捉えることができるため、単一 PE で動作するアプリケーションの移植や自動並列化コンパイラによるコード生成が比較的容易である。

2.1.2 NUMA

もう一つは、NUMA(Non-Uniform Memory Acesss : 不均一メモリアクセスモデル) である。NUMA とは、各 PE からメモリへのアクセス時間がアドレスによってことなるメモリモデルである。図 2.2 に示す通り、概念的には各 PE が、ローカルにメモリを持ち、そのメモリを大域のメモリ管理機構によって、共有メモリ的に扱っている。よって、共有メモリであるが、自 PE のメモリへのアクセスに対する時間より、他 PE のメモリへのアクセスにはより時間が掛かる。

[1]本稿では、PE を CPU とそのプライベートキャッシュを含めたものとする。

2.2 Architechtuer

2.2.1 RISC based SMP

本稿では、共有メモリ型並列計算機を用いた。特に RISC プロセッサを用いた SMP システムと呼ばれる形式の並列計算機である。 特徴としては、MIMD(Multiple Instruction Multiple Data) であり UMA である ということと、システムに実装されている全てのプロセッサが同等に機能し、全てのプロセッサが同一方法でメモリにアクセスする。すなわちメモリと全てのプロセッサは等距離である。また、システム中で 1 つのプロセッサがその他のプロセッサに対してマスタとして振舞うことはない、ということが上げられる。このシステムは、サーバとして利用されることが多く、数個のプロセスを実行させた場合、OS がプロセスを空き PE に自動的に割り当てる。このため通常の利用では、ユーザは並列システムと意識することはほとんどない。

しかし、欠点として以下のようなことが上げられる。

- プロセッサーメモリ間のバスの細さによる、バス転送混雑時のシステム全体の性能低下。
- RISC ベースであるため、パフォーマンスに対するキャッシュの影響が非常に高くなる。キャッシュ内データ使用不可による大規模演算での処理能力低下。

これら欠点は改善可能なものである。まず一つめは後述する Interconnect の方法や専用バスの採用で、大部分が改善される。また、二つめは、RISC based SMP システムに適したアルゴリズムを適用することである程度の改善が見込まれる。

2.2.2 Interconnect

SMP システムのメモリ接続方法にも、バス接続 (図 2.3) とスイッチ接続 (図 2.4) が存在する。SMP システムは、比較的容易に製作できるため小規模な（プロセッサ台数が少ない）計算機が数多く見かけられる。一般に、このような安価な SMP システムには、バス接続の物が使用される。すべての PE が単一のバスを介してメモリにアクセスするため、PE 数が増加すると、1PE のメモリのアクセスのバンド幅が低下するという現象が見られる。その現象を解決するため、PE-メモリ間のデータ転送をスイッチ (クロスバーネットワーク) に置き換える。この方法では、PE 数の増加によるメモリアクセスのボトルネックを抑えることが出来る。この方法を用いることで中規模程度の並列システムの製作が可

能となり、更に高速な接続が出来れば、大規模なシステムの構成も可能となる。しかし、スイッチ接続はバス接続に較べて高価なシステムとなる。

2.2.3 PVP

本稿では、キャッシュを持つ SMP システムでの実行結果との比較対象として、キャッシュを持たない (少い)PVP(Parallel Vector Processor) システムを用いている。PVP システムとはベクトル型プロセッサを並列に接続したもので、記憶モデルとして分散、分散共有、共有という方式がある。共有メモリでは、複数のメモリ・ポートによってプロセッサーメモリ間を接続しているため、非常に高速なメモリアクセスを実現している。このアーキテクチャではキャッシュがほとんど必要ないほどのメモリアクセス速度が実現され、特にキャッシュが効きにくい大規模問題では高い性能を発揮する。

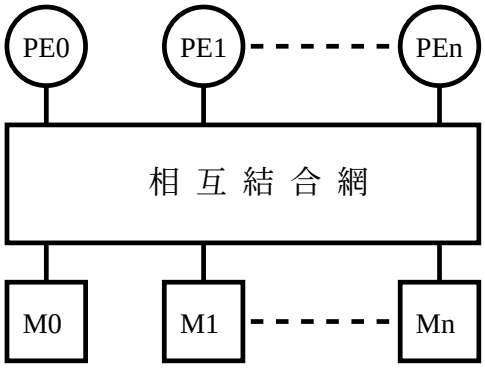

図 2.1: UMA

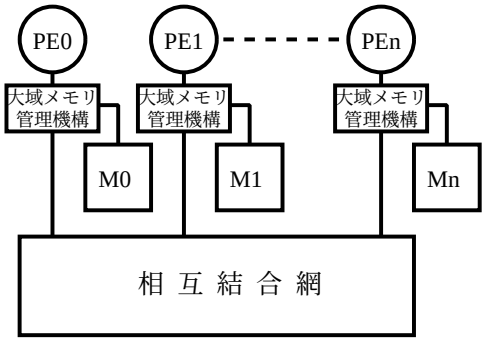

図 2.2: NUMA

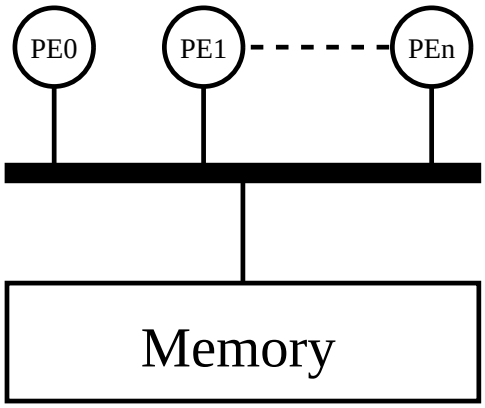

図 2.3: Bus Interconnect

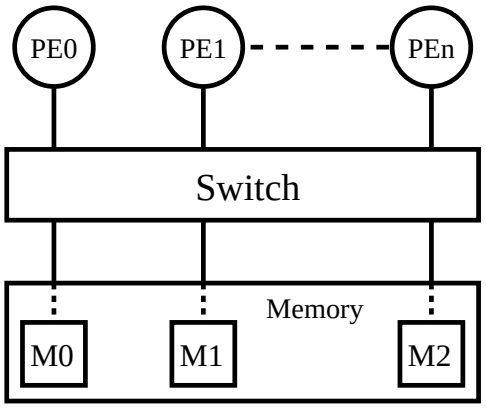

図 2.4: Switch Interconnect

2.3 Starfire

本稿では 富士通製 S-7/7000U model 500 (以後 Starfire とする) (SUN Microsystems 製 SUN Ultra Enterprise 10000 OEM 製品) を用いた。本システムは、現在の市場における最大規模の UNIX(Solaris) ベースの SMP システムである。Starfire の主要事項は以下の通り、

- 16~64 個迄の 250MHz UltraSPARC モジュール (1MB 外部キャッシュ付) をサポート
- 最大 16 枚のシステムボード (各ボード には以下を搭載可能)
  - 最大 4 つの UltraSPARC モジュール
  - 1 枚の I/O モジュール (最大 4 枚の SBus カード をサポート)
  - 1 枚のメモリモジュール (最大 4GB 搭載可能)
- 高いインターコネクト帯域幅 (最高 12.8GB/sec^2)
- 高速な浮動小数点演算性能 (ピーク時最高 32GFLOPS、実効 25GFLOPS 以上)
- I/O の柔軟性 (最大 32 個の独立 SBus)
- 高い I/O 帯域幅 (SBus 帯域幅ピーク時最高 6.4GB/sec)

本システムの Interconnect は、図 2.5 に示す通り、スイッチ接続になっており、プロセッサーとメモリの間にクロスバースイッチをおき、データはクロスバースイッチで、アドレスはバスで接続しさらなる高速化を計ったシステムである。アドレスがバス接続であるのは、キャッシュのコヒーレンシーをとる必要があるためである。本稿で用いたシステム仕様は以下の表 2.1 の通りである。

表 2.1: Machine Specification

CPU	UltraSPARC-II(250MHz)
available PE	32
Second Cache	1MB
Main Memory	4GB
Peak Flops	16GFlops
Peak Bandwidth	6.4GB/s

2.4 Cray-J90

本稿では、上記 Starfire での性能評価の比較として、SGI 社製 CRAY-J916se/8-4096 でも同一コードを実行しその効果を調べた。

本機は、PVP システムのミッドレンジベクトルシステムである。主要事項は以下の通り、

- 1 ～ 16 個迄のクロック周期 10 ナノ秒のオリジナル 64bit プロセッサ
- 最大 4GB の大容量共有メモリ (物理的にも論理的にも共有)
- 最大 51.2GB/s のメモリバンド幅
- 最大 3.2GFlops の浮動小数点演算

本稿で用いたシステム仕様は以下の表 2.2 の通りである。

表 2.2: Machine Specification

CPU	オリジナル
available PE	8
Scalar Cache	128 words
Main Memory	4GB
Peak Flops	1.6GFlops
Peak Memory Bandwidth	12.8GB/s

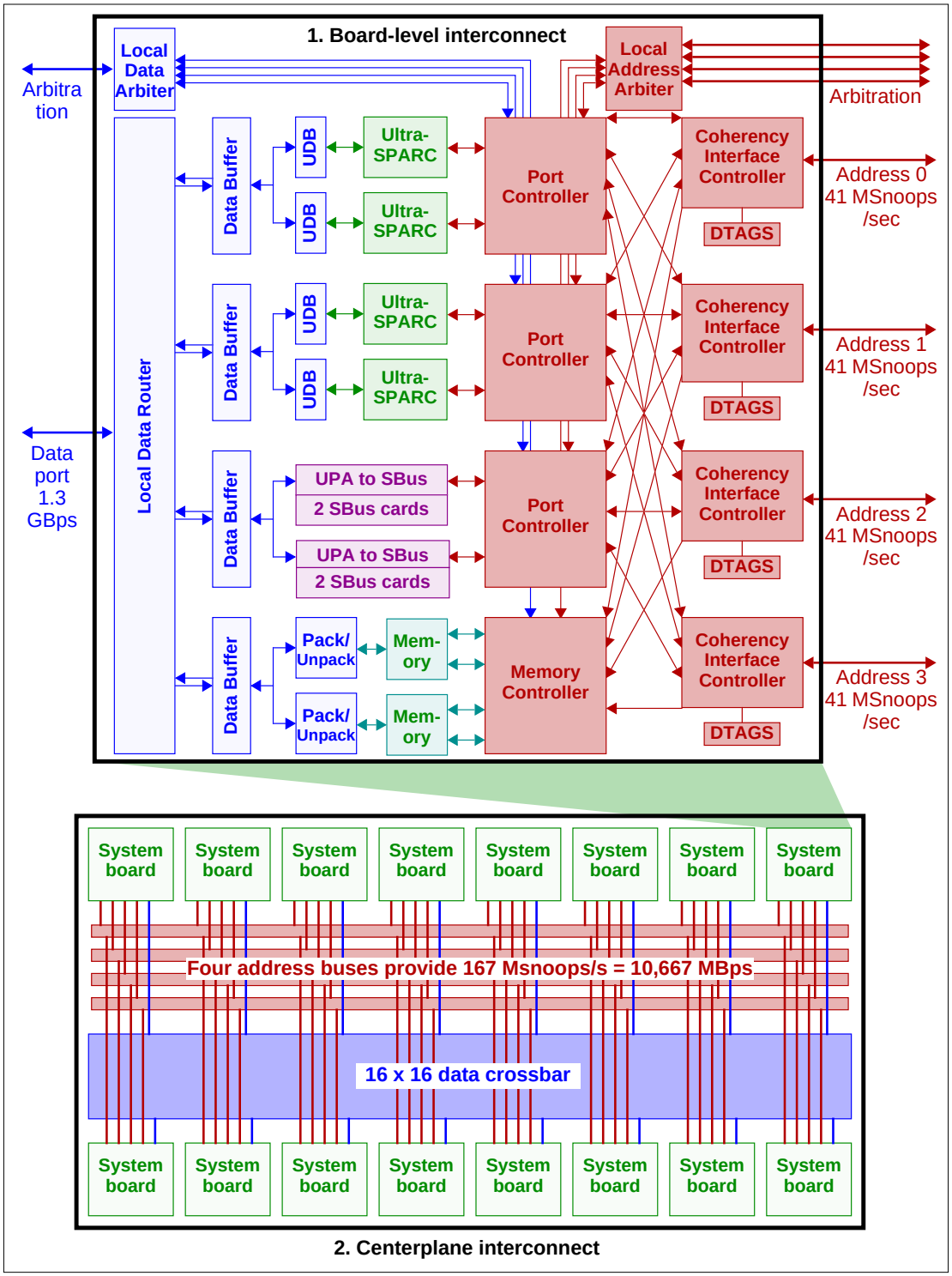

図 2.5: Starfire Interconnect[4]

第 3 章

Shared Memory Programming

3.1 Automatic Parallel Programing

近年のコンパイラ技術の発達によって、共有メモリシステムにおいてシーケンシャルなプログラムを分析し、プログラムの部分部分が等価であり並列実行に支障がないかどうかを自動的に決定する。このような技法には単純なものから複雑なものまで多様に存在するが、実際にループの繰り返しを任意の順序で実行できるかどうかを決めるのは非常に扱いが難しいことが多い。そのため、実際のプログラムに適応するのはそれほど容易なことではない。

単純な例として、次の 2 つのループを考える。

Example

```
for(i=0 ; i<1000 ; i++)
   a[i] = b[i] * c[i];

for(i=2 ; i<100 ; i++)
   a[i] = a[i-2] + b[i];
```

最初のループは、それぞれの繰り返しが他の全ての繰り返しから独立しているので並列化可能である。1 から 999 の間の i の全ての値に関して、式が評価される限り、評価の順番は重要ではない。反対に 2 番目のループは式 a[i-2] の正しい値を得て、その後に a[i] を求める必要があり、並列化は容易でない。

この問題をある程度解決する方法として、プログラマが陽に並列処理を指定してプログラムを記述する方法がある。ただ、これはあくまでもループ分割の制御文程度のもので並列プログラムとは言えないものである。制御文を用いた場合、コンパイラは制御文を優先し、実際データ依存関係が存在する場合でも並列化を行ないデータレースなどが発生する。このため制御文を挿入した場合のデータの整合性の維持はプログラマにある。

以上のように、データ間の依存がないプログラムでは、自動並列コンパイラを用いることによって、十分な並列効率が得られるものと考えられる。上述の Starfire および J90 においても自動並列コンパイラを実装しているが、Starfire ではスレッドが生成され、J90 では、プロセス[1]が複数個生成される。

3.2 スレッドプログラム

スレッドプログラムとは共有メモリプログラムモデルを用いて並列処理を行なうプログラムである。共有メモリシステムにおいて、スレッドを用いる利点は多く、プロセスを用いるよりリソースを節約できるためプロセスを複数生成するよりスレッドを生成する方が効率がよい。

現在スレッド標準は、UNIX インターナショナル (UI) が System V インターフェース定義の一部として採用したものと POSIX(Portable Operating System Interface) の P スレッドが存在する。これらは、UI スレッドと P スレッドでプリミティブと提供されている機能が少し異なるだけで、ほとんど同じものである。また、P スレッドでも POSIX1.003.1b のリアルタイムのようにプロセス間での同期を行なうのスレッドライブラリが存在している。[2]

3.2.1 プロセス

スレッドを取り上げる前にプロセスについて簡単に触れる。プロセスは、多くの実装系 (アーキテクチャおよび OS) では、図 3.1、3.2 に示すようにそれぞれのプロセスごとに独立したアドレス空間が割り当てられる。そして、そのアドレス空間は、ユーザ空間とカーネル空間に分けることが出来る。ユーザ空間には、ユーザが実行すべきプログラムコード、グローバル変数、スタック領域などが確保され、カーネル空間には OS が使用する情

[1]ただし、本稿では、J90 で生成されるプロセスも便宜上スレッドと呼ぶ。
[2]J90 でのスレッドライブラリはプロセス間同期のものである。

報が確保される。ここで重要なのは各プロセスは独立にアドレス空間を保持していることである。

単一 CPU でのマルチタスクでは、このように独立した処理の実態である各プロセスに対して、CPU 時間の割当を行ない、プロセスを適時切替えていくことによって、複数のプロセスを実行して行く。SMP システムでは、負荷は OS によって自動分散されて空き PE に分散される。そして、プロセス数が PE 数を上回った場合には、CPU 時間を競合する者に時間分割でそれぞれ割り当てる。MPP(Massively Parallel Processor) ベースの並列計算機も同様で各 PE にプロセスを 1 つづつ実行してプロセス間の通信を行なっている[3]。また、共有メモリシステムでのスレッドライブラリの中には実行形態がプロセスのものがあり、実装系によって様々であるが、別々のプロセス間で共有のデータを保持している。

3.2.2 マルチスレッドのプロセス

スレッドとは、プロセスに割り当てられた資源を利用して、実際の処理を行なう「実行経路」のことである。マルチスレッドをサポートしていない UNIX 系 OS でも、1 プロセスにつき 1 スレッドのみが存在し、1 つだけの処理を行なうという概念を用いる。マルチスレッドでは、1 つのプロセス中に複数のスレッドを持つことが出来る。その場合のスレッドはプロセス内部では図 3.3 に示すようになっており、プログラムコードやグローバルデータなどは、各スレッドで共有されることになる。そのため、各スレッドは共有されたプログラムコードとグローバルデータなどを用いてそれぞれ独立に処理を実行することが出来る。ただし、1PE のみの動作である場合、実行可能状態にあるスレッドは一つだけで、その他は優先順位を持って待ち行列内の中で待ち状態に入る。待ち行列から実行可能状態にスレッドを移すスケジューリング (CPU 割り当て) は、OS が管理し、そのスケジューリングの方針の決定は重要なことである。

3.2.3 スケジューラ

最近のプリエンプティブ OS は、一定の時間が経過すると、CPU の制御を OS のスケジューラに戻し、順次各プロセス又は各スレッドに CPU の使用権を与えていく方式を取っている。このため、各スレッドは少量の CPU 時間を割り当てられ、OS は更に少量の時間でスケジューリングを行なう。スケジューラに戻された CPU 制御は各スレッドに割り

[3] 1 PE 1 プロセスであるため切替えはほとんど考える必要はないない。

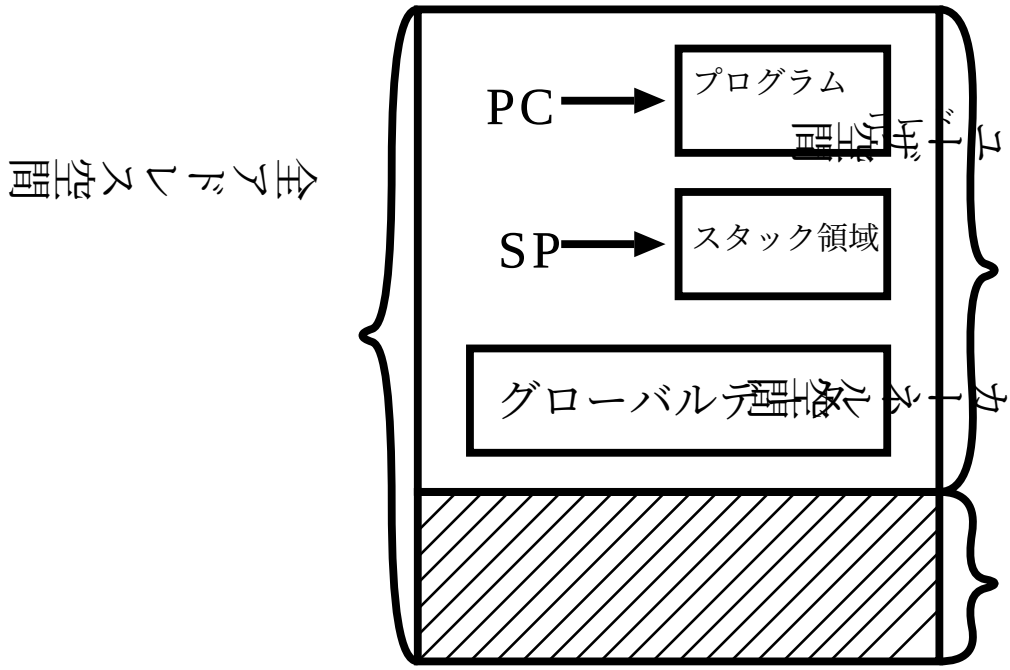

図 3.1: Process

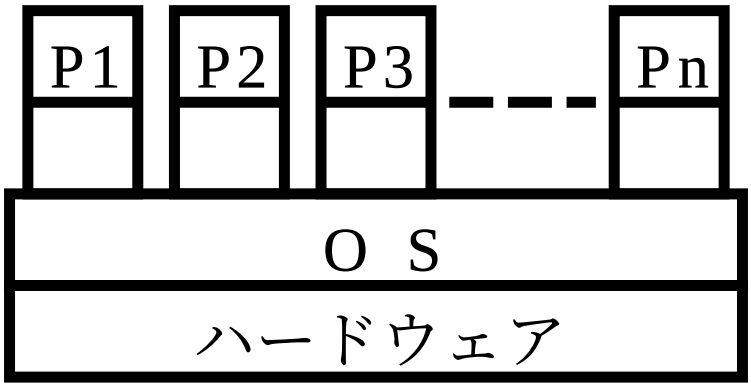

図 3.2: Multi Process

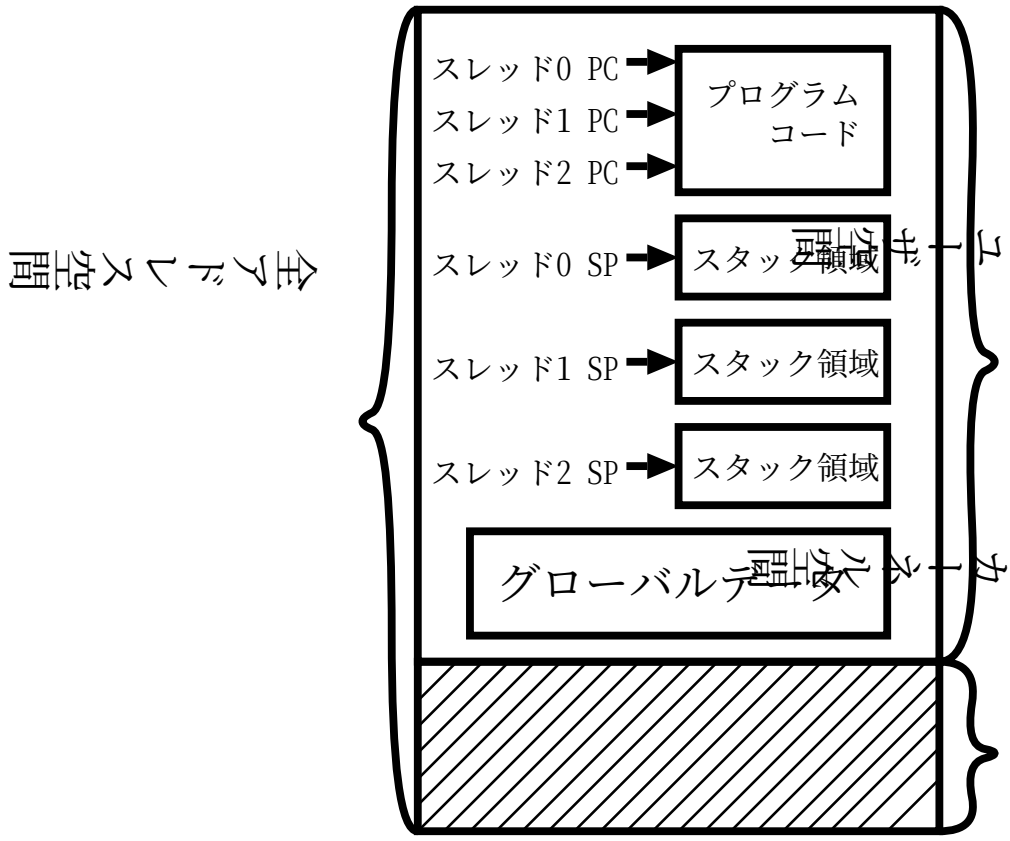

図 3.3: Thread Process

付けられた時間が経過しているかどうかを調べ、経過していない場合はそのスレッドの実行を継続し、経過している場合は待ち行列内先頭のそのスレッドと同等以上の優先順位を持つスレッドに新たに CPU 時間を割り当てて、明け渡す。この方式は、タイムスライシング (Time Slicing) と呼ばれ、実際のスレッドの切替えをコンテキストスイッチ (Context Switch) 呼ぶ。コンテキストスイッチは、それが行なわれる時点で CPU 割り当てを持つスレッドと次の割当を待つスレッド両方に依存する。

コンテキストスイッチには二つのタイプが存在する。一つめは同一プロセス内のスレッド同士がコンテキストスイッチを行なう場合 (a)。二つめは異なるプロセス間のスレッド同士、すなわちプロセスのコンテキストスイッチがある (b)。それら二つの違いは以下のようになる

(a) 全てのスレッド情報 (Program code, stack,etc.) がすべて同一アドレス空間に存在するため、スレッド情報復元のための参照が容易。

(b) スレッド情報が異なるアドレス空間に存在するため、アドレス変換を行なった後にス

レッド情報を復元するため非常にコストが掛かる。

以上のように、マルチプロセスとマルチスレッドの両方で実現可能なプログラムであれば、マルチスレッドで実現した方が効率的である。しかし、いずれの場合もコンテキストスイッチが発生し、全体パフォーマンスは上がらない。科学技術計算を行なう上では、単一CPUでマルチスレッドを用いても単一スレッドでの計算時間より、遅くなってしまう場合がある。

3.2.4 スレッド利用の利点

High Performance Computingの分野でマルチスレッドプログラムが、絶対的な優位さを主張できる場合は、複数の実行状態を持つモデル、つまり、1つのシステムに複数個のプロセッサを持っている場合である。それは、SMPシステムやその他の共有メモリシステムである。このようなシステムでは実行状態が複数個存在し、その分だけ実行待ちスレッドを多数処理できる。また、並列システムの有効利用を考えた場合、通信ライブラリを用いたプログラムの並列とマルチスレッドの同期を用いた並列では、同期時間の方が格段に小さくなる。また、共有メモリシステムにおいて、通信ライブラリを用いるものは、大体の場合プロセスを用いたプログラムとなる。そのため、上述のコンテキストスイッチの動作もスレッドに較べ低速となる。そのため、マルチスレッドプログラムでは、本来並列に向かないアルゴリズム内部の並列性を抽出し、SMPシステム上でスレッドを用いることで、より高速に処理を行なえる可能性がある。

また、プログラミングにおいてもMPPなどでは通信を考慮したプログラミングを行なう必要がある。それにともなって、データを各PEにおいて独自に持つことになり、データのインデックスの付け替えなどの面倒な作業がある。科学技術計算では、データ配列が大量に利用されるため、この問題が解決出来るのは非常に意味が大きい。そして、スレッドプログラムでは、通信は考慮する必要がないため通信部分のプログラミングのコストを省くことが出来る。また、データも共有データとして扱えるためインデックスの付け換えを気にする必要がない。以上のように科学技術計算をSMPシステムとマルチスレッドプログラムで行なうことは、非常に有効であると考えられる。

第 4 章

Benchmark

本稿で使用する SMP システム (Starfire) の基本性能を把握するため、いつくかのベンチマークを実施した。以後 MPI を用いて並列化した場合には、使用する PE を PE 数とし、スレッドを用いて並列化した場合にはスレッド数とする。

4.1 MPI

ここでは、本稿で用いた通信ライブラリ (MPI) の基本的な性能を示す。本来共有メモリシステムにおいて通信を行なう必要はない。しかし、現在並列計算機で用いられているプログラムの並列化手法では MPI が標準として用いられている場合がほとんどである。そのため、共有メモリ型の並列計算機においても MPI を用いたプログラムを実行できるように環境を整えた。本稿で用いた通信ライブラリは、ANL(Argonne Natioanl Laboratory) が公開している MPICH 1.1.0 を Starfire 上に構築した。

4.1.1 計測対象

本稿で行なった通信ベンチマークは次の二つである。

PingPong: 任意の長さ N のメッセージをマスターノードからスレーブノードに送る。スレーブノードは配列に受けとるとすぐにマスターノードに送り返して、マスターノードが受けとるまでの時間

Broadcast: 任意の長さ N のメッセージの同報通信に要する時間

以上の計測を行なう。メッセージの長さ N は、1 ～ 10000 とし、通信要素は倍精度とした。また、同報通信では、PE 数を 2 ～ 16 までとし、要素の精度は PingPong と同じである。図 4.1 に PingPong ベンチマークの結果を示し、図 4.2 に Broadcast ベンチマークの結果を示す。

4.1.2 結果と考察

Pingpong ベンチマーク (図 4.1) では、データサイズに対してある程度スケーラブルである。ただ、データサイズの節目に時間の段差が存在する。これは、メモリの読み出し、書き出しブロック長とデータサイズに関係するものと考えられるが、詳細は不明である。

Broadcast ベンチマーク (図 4.2) では、データサイズおよび PE 数に対してともにスケーラブルである。ただ、共有メモリを踏まえた MPI の実装が行なわれているとするならば、この結果は得られないだろう。なぜなら、共有であるため共有データにブロードキャストするデータをおいておけば、他の PE から同時読み出し出来るため、PE に対してスケーラブルになることはないと考えられる。

Starfire での、通信性能の計測は、メモリアクセスの能力と考えても良い。SMP システムでのメッセージパッシングを行なうことは、ある側面を考えると、余り意味のないことのようにも見える。しかし、標準通信ライブラリというプログラム手段が存在する限り、その標準ライブラリを用いたプログラムが多く存在するというのも事実である。

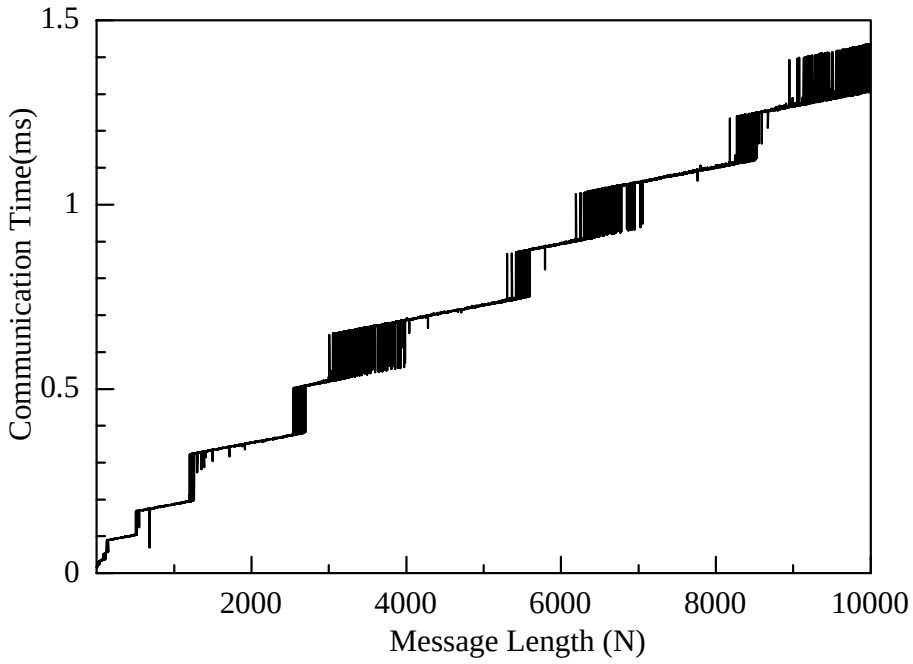

図 4.1: Pingpong Benchmark

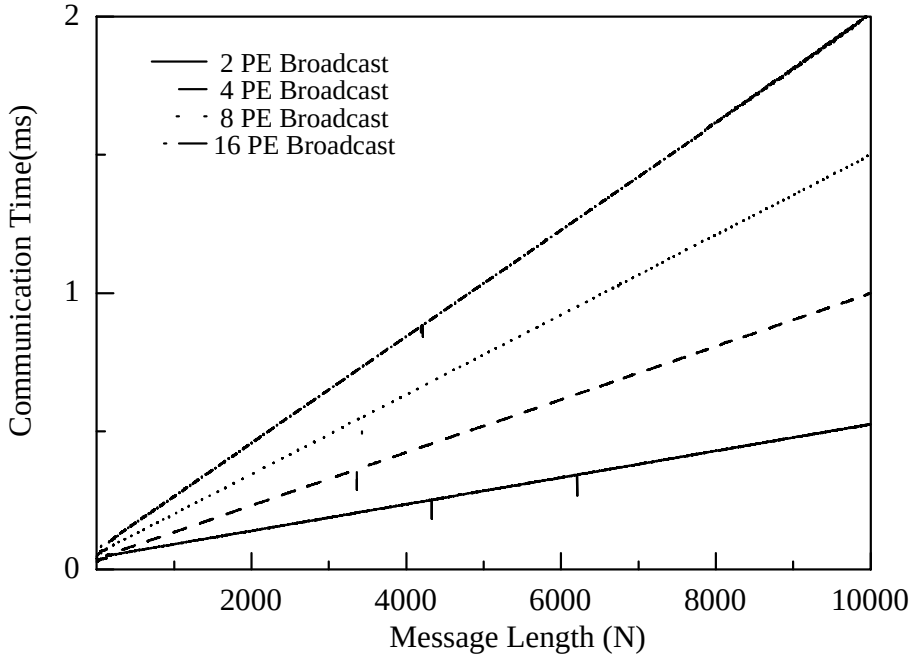

図 4.2: Broadcast Benchmark

4.2 行列乗算

並列計算機の基本的な能力を調べるために、行列乗算を行なった。評価に用いたプログラミング方法は 3 種類である。まず一つめはシーケンシャルな行列乗算プログラムを自動並列化コンパイラを用いて並列化したもの、二つ目は MPI を用いたものもう一つはスレッドを用いたものである。

4.2.1 計算対象

プログラム自体は、通常の行列乗算プログラムであり、コードレベルの最適化などは行なっていない。以下に用いたコードを示す。問題の大きさ N は、2048 配列要素は、倍精度の乱数とした。ただし、MPI の場合は通信用のメモリ空間を確保するため、32 PE での結果が得られなかった。実行結果を図 4.3 と図 4.4 に示す。

ikj 型

```
for(i=0 ; i<N ; ++i){
   for(j=0 ; j<N ; ++j)
      a[i][j]=0.0;
   for(k=0 ; k<N ; ++k){
      for(j=0 ; j<N ; ++j)
         a[i][j] = a[i][j]+b[i][k]*c[k][j];
   }
}
```

スレッド および MPI での領域分割方法は、最外ループ i による使用 PE 数による単純分割である。また、スレッドの場合での時間計測は、スレッドの生成破壊までも含めたものである。そのため、スレッドの生成破壊に要する時間の計測結果を図 4.5 と J90 におけるスレッドの生成破壊時間 (図 4.6) も合わせて表示する。

4.2.2 性能評価

性能評価の方法として、2 つのデータを示す。

- 計算時間 (Computation Time)

- Flops(Floating Operation per Second)

4.2.3 結果と考察

図 4.3 と図 4.4 を見ると、どの並列化手法を用いても同程度の並列化が行なわれているが、自動並列化コンパイラを用いた場合の結果が良い。

行列乗算プログラムは、基本的にデータの依存関係が存在しない。そのため、自動並列化コンパイラを用いた場合でも十分なパフォーマンスが得られていることが分かる。また、ソースコードに何も手を加えてないということも考えると、非常にプログラミングコストが低くなっている。

次にスレッドライブラリを用いたの場合を見てみる。この結果は、スレッドの生成、破壊のオペレーションを含んだ結果である。しかし、生成、破壊の時間的なオーダーから考えて、自動並列と較べて遅くなる理由とは考えにくい。よって、スレッドライブラリを用いた場合の自動並列化コンパイラを用いた場合との並列処理時間の低下は、それ以外の原因であると思われる。しかし、自動並列化コンパイラと較べ、スレッドプログラムは並列の自由度が高い。そのため、スレッドプログラムおよび SMP システムを意識したプログラミングを行なえば、自動並列化コンパイラを用いた場合より、高速に実行できると考えられる。

また、MPI ライブラリを用いた場合を見てみると、上記の二つの場合と較べて、実行時間が落ちているのは、明らかに通信時間だと考えられる。MPI の実装方法は、左行列を PE 数分で分割し分割分を各 PE に送信、右行列を全 PE にブロードキャストという方式をとっているため、MPI ベンチマークの Broadcast でも明らかなように、PE 数にスケーラブルに効いてきていると考えられる。ただし、この場合もスレッドの場合と同様に MPI を意識したプログラミングをすることによって、高速実行は可能と思われる。残念に思われることこの並列化手法では、MPI での 32 PE 使用時の実行結果が得られなかったことである。後述するが、SMP システムにおいて、プロセス数がシステム全体の PE 数を使用する場合、パフォーマンスの減少が見られる場合が多いがその現象が捉えられなかった。

スレッド生成・破壊時間について見てみると、図 4.5 に示す Starfire の場合、スレッド生成・破壊時間はスケーラブルであり、非常に小さな時間で済んでいる。16 スレッドを生成・破壊しても数百マイクロ秒程度である。次に図 4.6 に示す J90 の場合では、スレッドライブラリを用いて、生成されるのは実際には プロセス であるため、Starfire の場合に較べて、非常に時間が掛かっている。

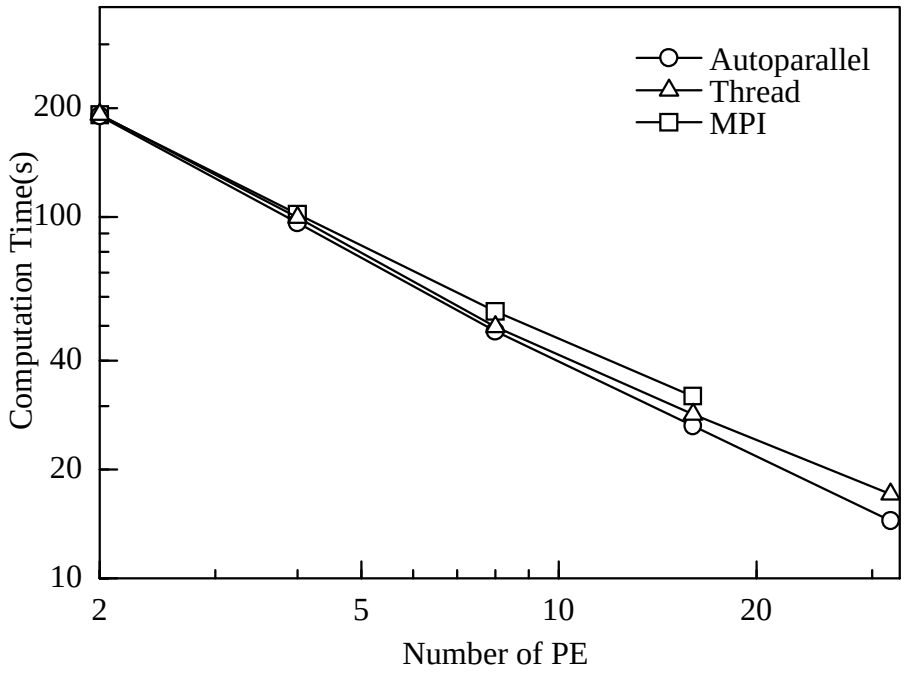

図 4.3: Computation Time

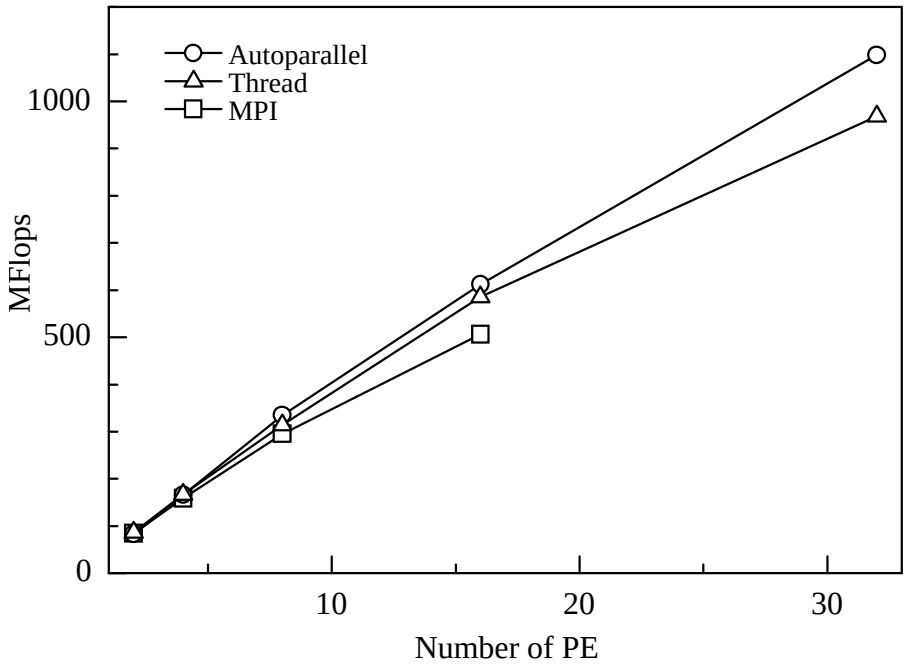

図 4.4: Flops

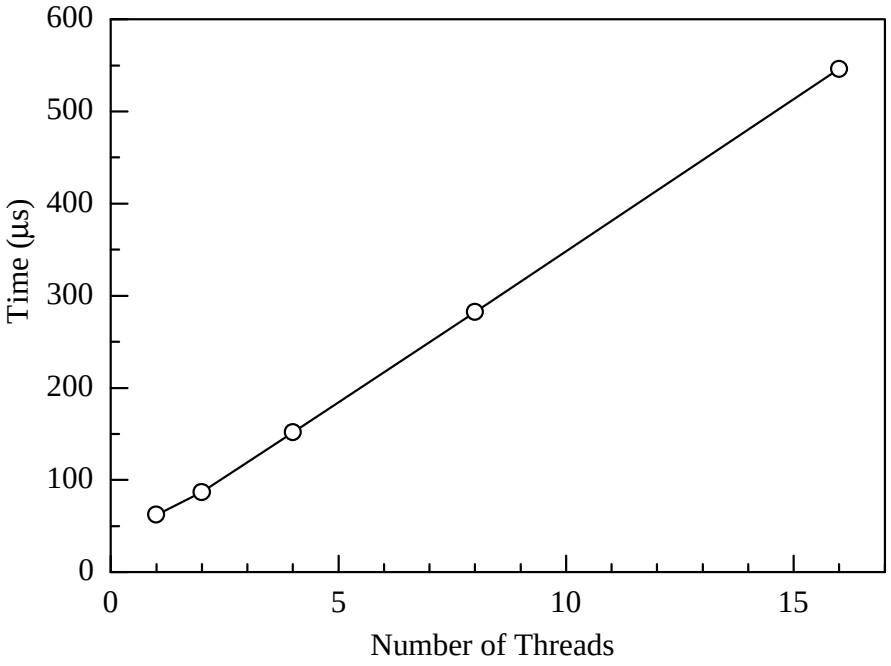

図 4.5: Thread Create & Destory Time

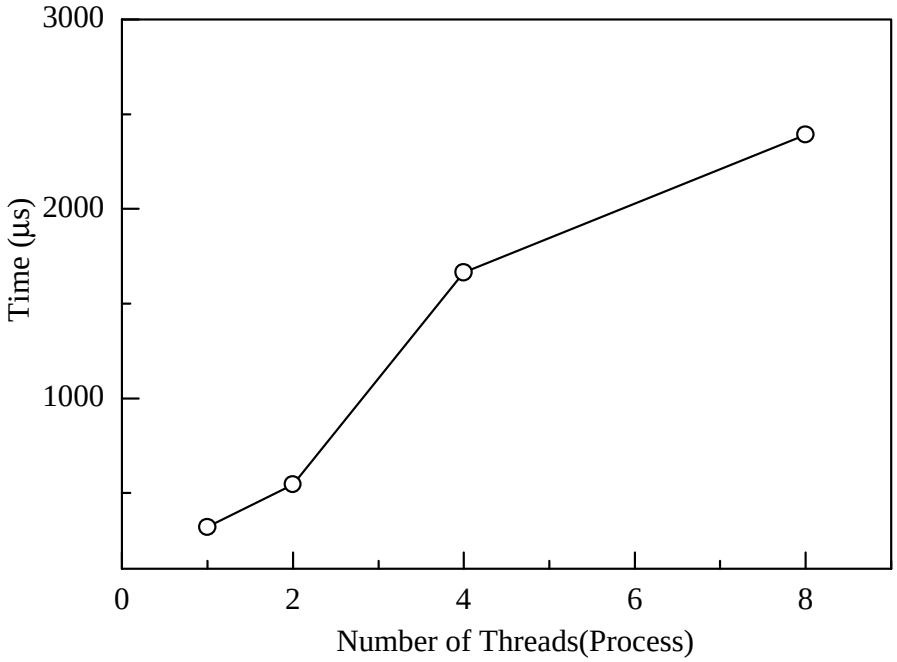

図 4.6: Thread(Process) Create & Destory Time

4.3 熱伝導問題

4.3.1 計算対象

並列計算機用のベンチマークプログラムとして、二次元正方領域の定常熱伝導問題の計算を行う。記述言語は Fortran、メッセージ通信ライブラリは MPI を用いている。

計算対象は、二次元の正方領域における定常熱伝導問題を用いる。内部に一様発熱 Q を仮定し、境界条件として温度 $T_0 = 0$ を与える。計算を行なう熱伝導方程式は以下のようになる。

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + Q = 0$$

この方程式を二次中心差分で離散化すると、

$$T_p = \frac{\Delta y^2 (T_e + T_w) + \Delta x^2 (T_n + T_s) + \Delta x^2 \Delta y^2 Q}{2(\Delta x^2 + \Delta y^2)}$$

が得られ、この式を収束計算することとなる。

計算では、図 4.7 に示す正方領域の計算対象に等間隔直交格子を生成し、格子点上に温度 T を定義する。

並列計算では、この正方領域を縦分割し、隣接した領域を持つプロセッサ間で、収束計算に必要な値を受渡しする (図 4.8)。

4.3.2 性能評価

性能評価の方法として、3 つのデータを示す。

- 計算時間 (computation time/1000step) 図 4.9

$$Time = \frac{computation\ time}{1000 step}$$

- 速度向上率 (speedup ratio) 図 4.10

$$speedup\ ratio = \frac{calculation\ time\ for\ 1PE}{calculation\ time}$$

- 効率 (efficiency) 図 4.11

$$efficiency = \frac{speedup\ ratio}{number\ of\ PE}$$

以上の計測を分割数がそれぞれ 129, 257, 513, 1025 の場合で行なった。ただし、分割数は上記の値であるが、グリッド点で値を評価するため、実際には (1) 130 × 130 (2) 258 × 258 (3) 514 × 514 (4) 1026 × 1026 の四種類の節点数の場合を計測した。また、温度 T の計算精度は倍精度である。

4.3.3 結果と考察

計測結果、(1) では、ほとんど並列の効果が得られず、逆に PE が増えると効率が極端に下がっていく。(2) では、4 PE 使用時までは、ある程度の並列の効果が得られているがそれ以上の PE を用いた場合、効率は下がっていく。(3) と (4) では、一様に並列化の効果が見られるが、(3) では、8 PE 以上 (4) では、16 PE 以上で並列の効果が現れていない。

まず、計算時間を見た場合に最も注目しなければならないことは、1 PE の場合、(1) と (2) では、計算量が約 4 倍であり、計算時間も約 4 倍になっている。しかし、(2) と (3) を見た場合、計算量は約 4 倍で、計算時間は約 20 倍へと増大し、(3) と (4) では、7 倍になる。この原因を考えた時、最も大きな理由であると思われるのは、プライベートキャッシュの影響である。温度 T を格納する配列のメモリ量を考えると、表 4.1 のようになる。

表 4.1: Memory size

	メモリ量 (KB)
(1) 130 × 130	132
(2) 258 × 258	520
(3) 514 × 514	2064
(4) 1026 × 1026	8224

Starfire の CPU 毎に搭載されているプライベートキャッシュは、1MB であり、1 PE で計算する場合、(3) でキャッシュ容量が飽和している。このため、上記のような現象が起こったものと考えられる。その他にも原因はあると思われるが、今計測で得られたデータからは以上のことが明確に推測できる。

本稿で用いた領域分割法による SOR 法は、領域内データのみを演算するため、データ参照に局所性が生まれる。この結果 (3)、(4) の場合において、並列加速率および並列化効率が、理論値を越える値を引き出している。これは (3) の 2, 4, 8 PE と (4) の 2, 4, 8, 16

PE の部分でその部分での 1 PE 当たりのメモリ利用量を見た場合、表 4.2 のようになる。ただし、領域の両端の部分は境界条件部分の領域を多く含むため、少しメモリ使用量が多くなるが、表は全体の平均値である。

表 4.2: Memory size per 1PE

PE 数	2	4	8	16
(3) 514 × 514(KB)	1032	516	258	129
(4) 1026 × 1026(KB)	4112	2056	1028	514

(3) では、使用メモリ量が 1024 KB を下回るに連れて、並列加速率、並列化効率ともに上がっていくが、8 PE をピークに下がっていく。これは、計算量と通信コストのトレードオフがあるものと考えられる。(1) (2) の場合は、計算領域が小さいため、通信が多くなるため、これらの場合では並列化によってほとんど計算時間が短縮しないか又は逆に増加している。このベンチマークでは通信が多くなるような場合、良い結果が得られていない。

次に (4) では、メモリ使用量は表 4.2である。16 PE まで計算領域を分割しないとキャッシュに収まる容量にはならない。しかし、4 PE を越えた当たりから、(3) よりも並列加速率および効率が良い結果が得られている。ただし、1 PE の測定時間が (3) の場合と較べて更に遅くなっているため、数値上 (3) よりもそくなっているにすぎない。そして、メモリからのデータの読み込み量が多いため、使用する PE 数が増えることで、全体のメモリアクセスバンド幅が増加していることが、効率が上がる一因であると推測される。

また、32 PE 使用時に、どの場合も計測時間、並列加速率、並列化効率ともに悪くなっている。この原因は幾つか考えられるが、原因の特定までには至っていない。原因として、32 PE 搭載の SMP システムにおいて、32 PE 使用することは、その内の 1 PE で動いていると思われる OS 関連プロセスと 1 PE の実行権をとり合う結果となり、32 PE 中の 1 PE の動作の遅延が全体に影響してる。そして、大きな原因として、通信によるメモリアクセスと計算によるメモリアクセスそして、前述こコンテキストスイッチなどの複合的な原因があると考えられる。また、分割数が小さな場合では、通信と計算量とのトレードオフが問題と考えられる。

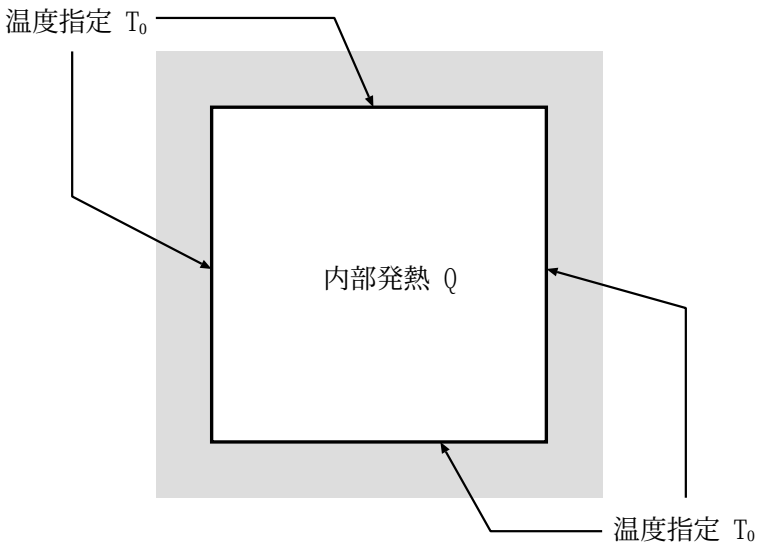

図 4.7: Computation Model

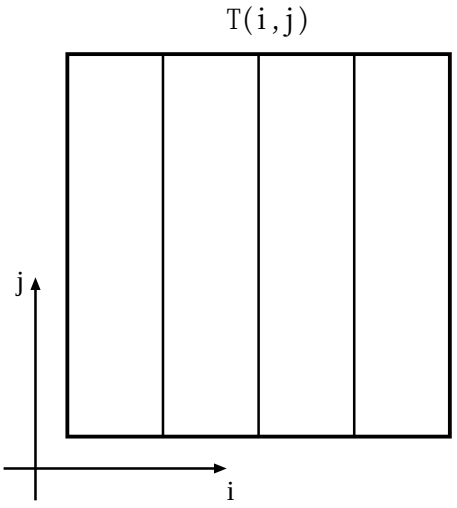

図 4.8: Domain decomposition

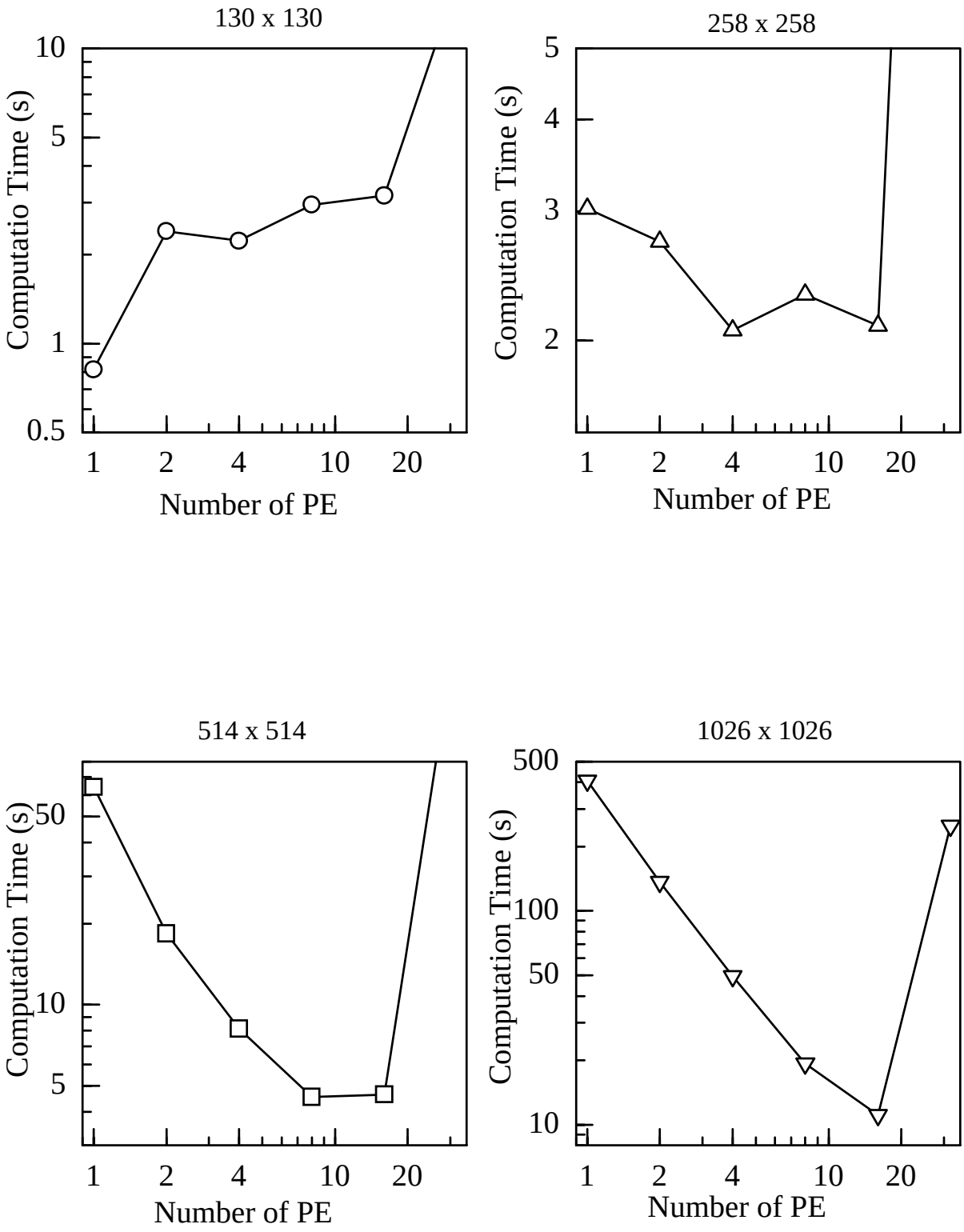

図 4.9: Computation Time per 1000step

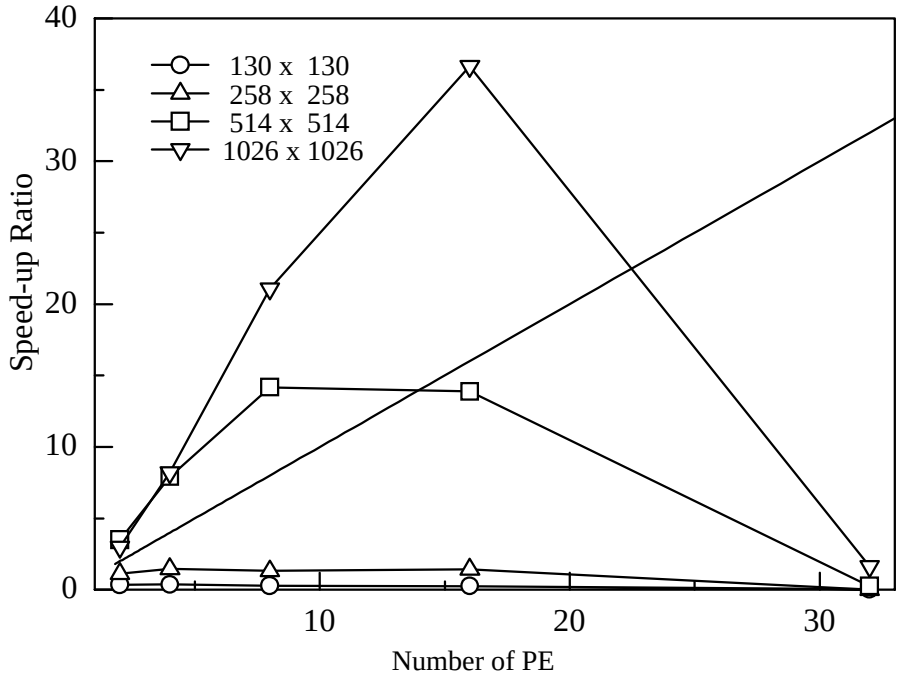

図 4.10: Speedup Ratio

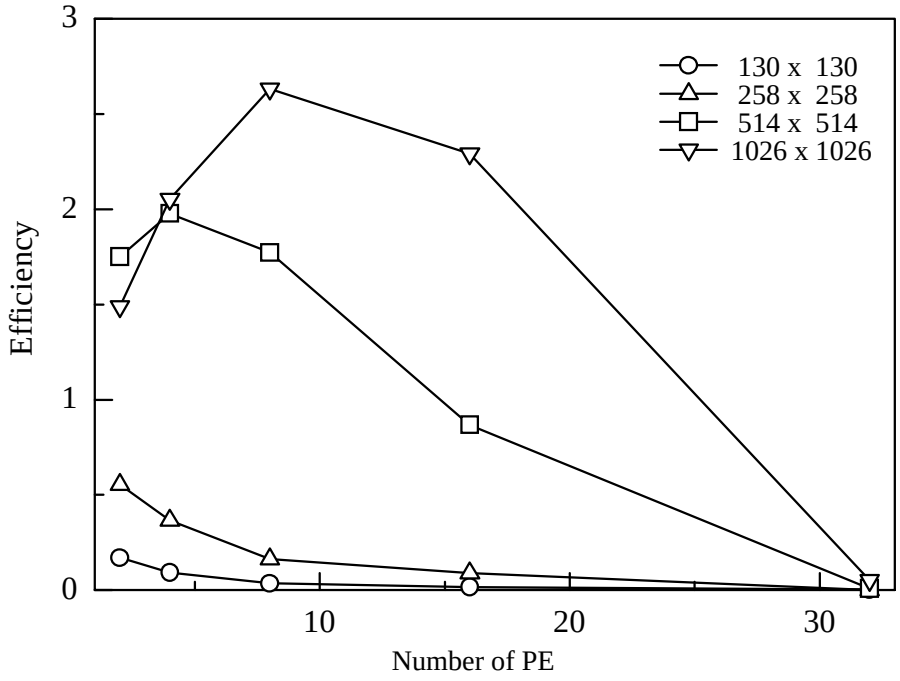

図 4.11: Efficiency

4.4 Nas Parallel Benchmark

Nas Parallel Benchmark(NPB) は、NASA Ames Research Center で作成された、熱流体関連の並列数値計算ベンチマークである。ソースコードは Fortran77 と MPI を用いて記述された並列版と Fortran77 のみで記述された逐次版が存在し、両方ともインターネット上で公開されている[1]。また、様々なメーカの計算機システムで実行された結果についても同様に公開されている。(CRAY-J90 の結果も存在する。) NPB は 5 つのカーネルプログラムと 3 つの CFD アプリケーションプログラムから構成されている。

本稿では、その中から CFD アプリケーションである以下の三つを並列版と逐次版のもの計測を行なった。。逐次版のものは自動並列コンパイラを用い並列化したものを用いた。

LU: 三次元圧縮性 Navier-Stokes 方程式を 5×5 のブロック上下三角行列方程式を簡易化 SSOR 法で解いている。この解法は、小さなデータを多く交換する。MPI を用いた場合、解法上使用する PE 数は 2 の累乗である必要がある。

- gridsize $= 102 \times 102 \times 102$
- itreration $= 250$
- Floating point operation $= 4988275 \times 10^5$

SP: 三次元圧縮性 Navier-Stokes 方程式を類似した構造の非優位対角なスカラ五重方程式で解いている。この解法は、大きなデータを少い回数で交換する。MPI を用いた場合、PE 数は平方数にする必要がある。

- gridsize $= 102 \times 102 \times 102$
- itreration $= 400$
- Floating point operation $= 7021896 \times 10^5$

BT: SP と同様の方程式を非優位対角な 5×5 のブロックサイズのブロック三重対角方程式で解いている。SP と同様に使用する PE 数は平方数である。

- gridsize $= 102 \times 102 \times 102$
- itreration $= 200$
- Floating point operation $= 3550125 \times 10^5$

[1] http://science.nas.nasa.gov/Software/NPB

4.4.1 性能評価

性能評価の方法として、NPB が示す 2 つのデータを評価として用いる。

- 計算時間 (calculation time)
- Mop/s(Million operation per second) 浮動小数点演算数を計算時間で割ったもの。
- 上記の結果について、逐次版を図 4.12 に、並列版を図 4.13 に示す。

4.4.2 結果と考察

結果を見ると、逐次版は並列の効果がほとんど得られていない。しかし、並列版では、効率良く並列化が行なわれている。逐次版では、24 PE 前後からは、逆に効率が悪くなっている。また、並列版でも LU の 32PE 使用時は並列の効果が落ちている。

この問題の規模は間違いなく大規模問題に属する計算量である。そのため、上述のキャッシュなどの影響がほとんどないと考えられる。この問題で注目したいのは、自動並列化コンパイラの実用問題への適用に対する問題と大規模問題に SMP システムを適用した場合の効率についてである。

逐次版で自動並列化コンパイラを用いた場合、ここでも 1 PE 使用時が非常に遅いことが言える。この状況を見る限り SMP システム特に Starfire では、1 PE の計算を動かすべきではない。また、自動並列の場合 PE 数の増加が 効率の上昇には繋がっていない。NPB ベンチマークでは、データの依存関係が存在する演算がほとんどであり、自動並列で並列化が可能な領域が限られてくるため、並列化できる部分のみの並列化効率は上がっていると考えられるが、並列度が上がると逆に遅くなる部分も見られ、全体の並列化率はほとんど上がっていない。

次に並列版の結果を見ると、最適化されたコードらしく、かなり良い並列化が行なわれている。そして、計算量が多いため通信の影響がほとんど出ていない。このことから Starfire は、ユーザー自ら並列化したコードを効率良く並列で実行する能力を持っていると思われる。また、ここでも 32 PE 使用時に並列効率が下がるような現象が見られているが、計算量が多いためか前記ほどではない。

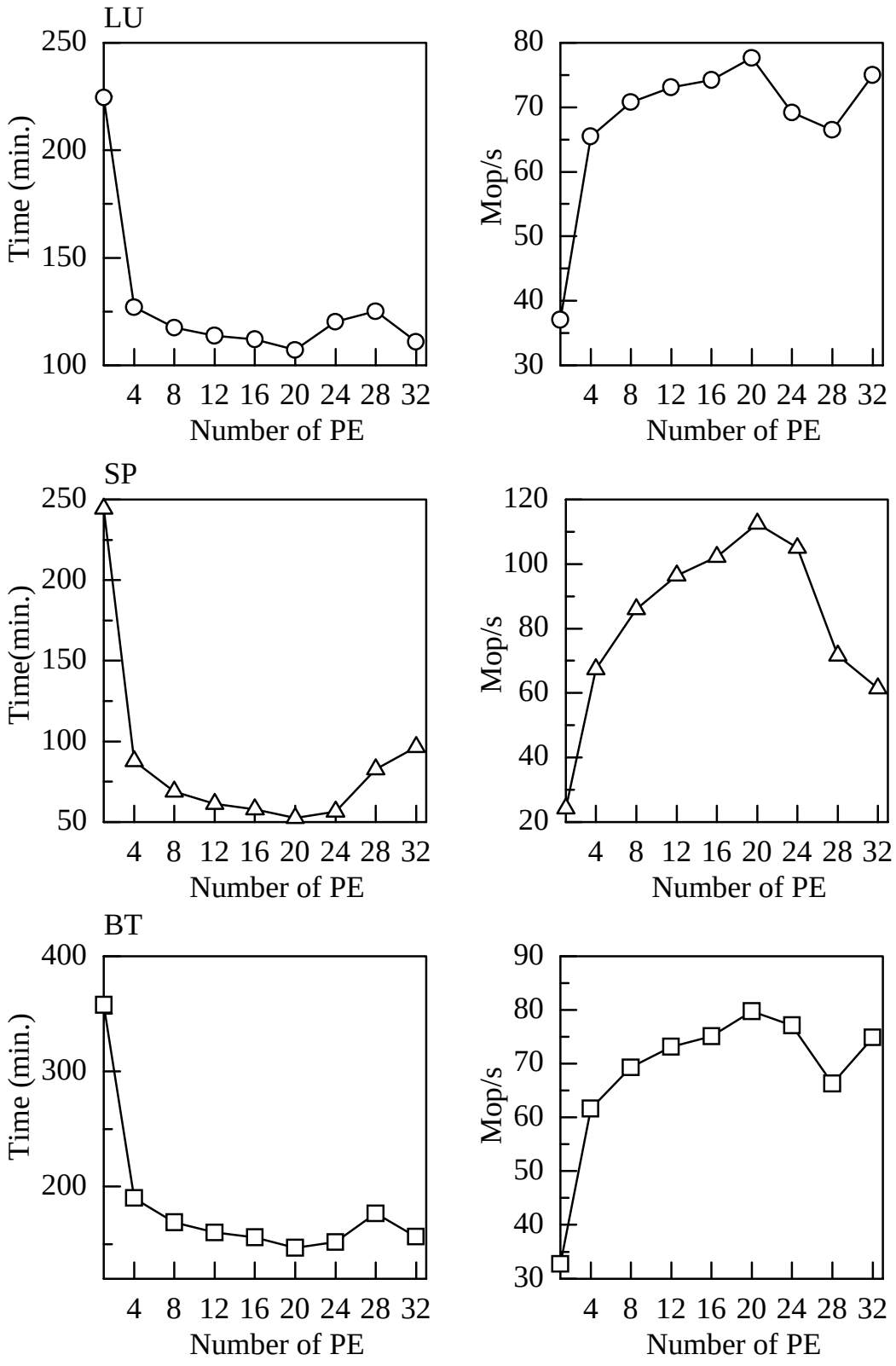

図 4.12: NPB-serial upper:LU, middle:SP, bottom:BT

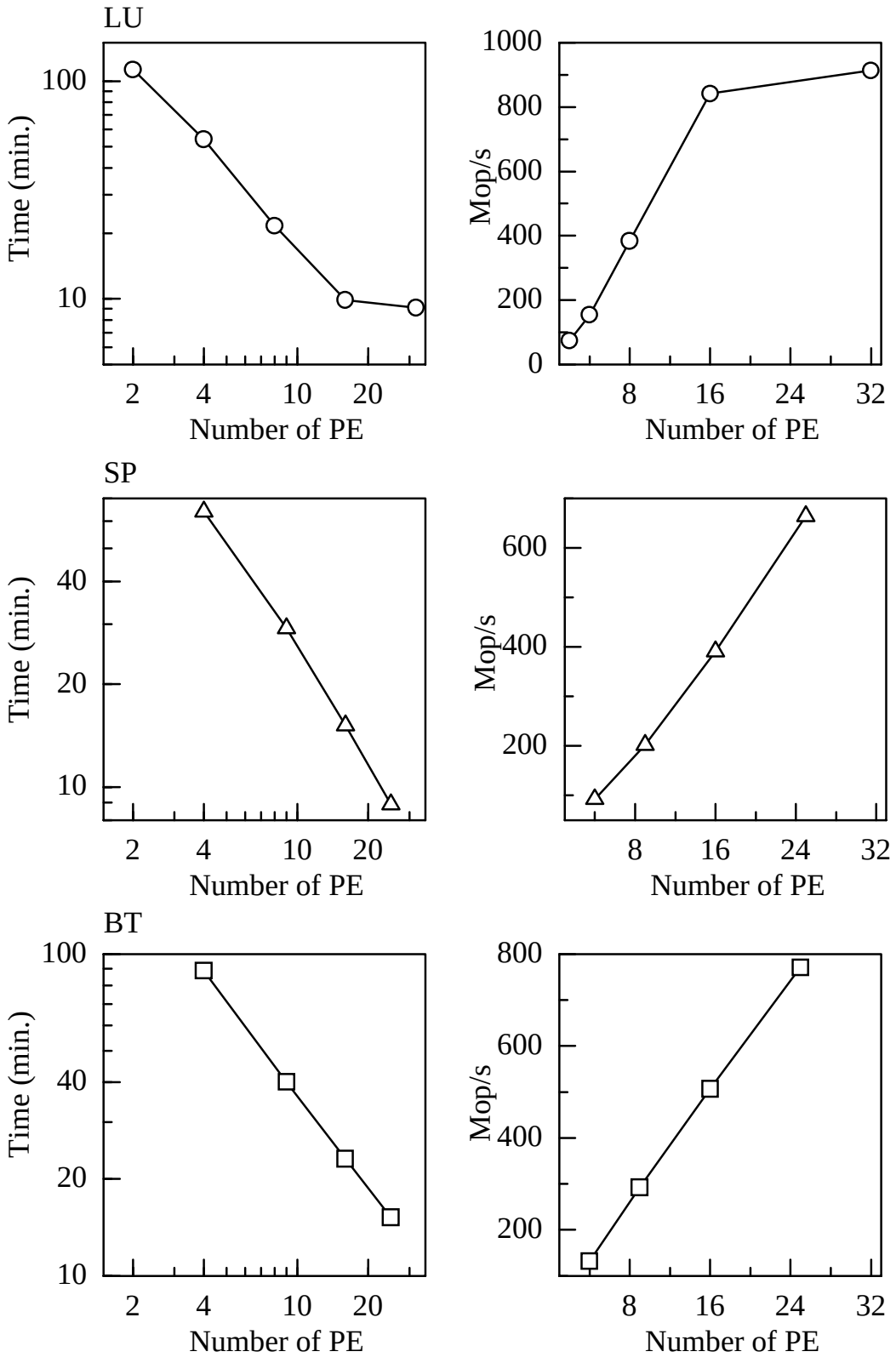

図 4.13: NPB-parallel upper:LU, middle:SP, bottom:BT

第 5 章

同　期

分散メモリ型並列計算機ではメッセージパッシングを用い、データ並列やパイプライン並列処理などを記述することは容易であるが、共有メモリ型並列計算機では、まだプログラミングの方法が整備されておらず、実問題を解析する前に実装すべきプログラムがある。本章以降では、Starfire については、16 スレッドまでの評価を行ない、J90 では 8 スレッドまでの評価を行なう。

5.1　同期機構

共有メモリ機構を持つ計算機システムにおいて並列プログラミングを行なう場合、排他制御、同期制御が非常に大きな役割を果たす。しかし、Pthread ライブラリでは、排他制御プリミティブは提供しているが、バリア同期などの大域的な同期や MPI ライブラリでの $send, recv$ に相当する 1 対 1 のデータアクセスの同期は提供されていない。これらを実装することは、共有メモリ型並列計算機を利用するために不可欠であり、アルゴリズムの実装においてプログラミングのコストを下げることが出来る。

通常スレッドプログラムの同期には、大域的に同期を取るバリア同期が用いられる。共有メモリのバリア同期には、すでに多くの文献が存在する。バリア同期には計算フェーズを揃えて、ネットワークやその他資源の不必要な混乱を解消することができるというメリットがある。しかし、大域的な同期であるためのコストが大きい。そのため、建部ら [3] は共有メモリ計算機での局所的な同期機構を提案している。このような同期方法を用いた場合、パイプライン並列処理を用いた計算を行う場合、バリア同期などの大域同期を用いる必要はなく、これら、同期機構を用いることによって、メッセージ・パッシングを用いるよりも高速に効率的に問題を解析することが可能であると考えられる。

5.2 バリア同期

バリア同期とは、各スレッド又はプロセス間ですべての実行処理が終了するまでバリアポイントで同期待ちをすることである。各動作のフェーズを合わせるために、あるスレッドの実行処理が終了すると同時にその他のスレッドに対して強制同期を掛ける EUREKA 型同期制御ではない。バリア同期の概念図を図 5.1 に示す。塗りつぶされていない帯が実行処理中、黒塗りの帯が待機中を示している。

共有メモリ型並列計算機でのバリア同期の方法は、すでに多数の文献が存在し、共有メモリ型並列計算機におけるバリア同期に関しては、Mellor-Crummey ら [6] の論文でアーキテクチャと効率的なバリア同期の機構について述べられている。キャッシュコヒーレントマルチプロセッサの場合、集中カウンタ方式、あるいは 4 分木と集中極性フラグを用いた方式が良く、キャッシュコヒーレントを持たない、又はディレクトリベースのキャッシュコヒーレントを持つマルチプロセッサの場合は、dissemination を用いた方式がよいと報告されている。また、バリア同期の手法として最も単純な exclusive を用いた場合の結果も合わせて表示する。

本稿では、exclusive、dissemination、tree の三種類を Starfire 上に、exclusive、dissemination の二種類を J90 上に実装した。Starfire でのバリア同期による遅延時間を図 5.4 に、J90 での遅延時間を図 5.5 にそれぞれ示す。遅延時間が少ない順に、Starfire では、 tree、dissemination、exclusive となり、J90 では、dissemination、exclusive となっている。

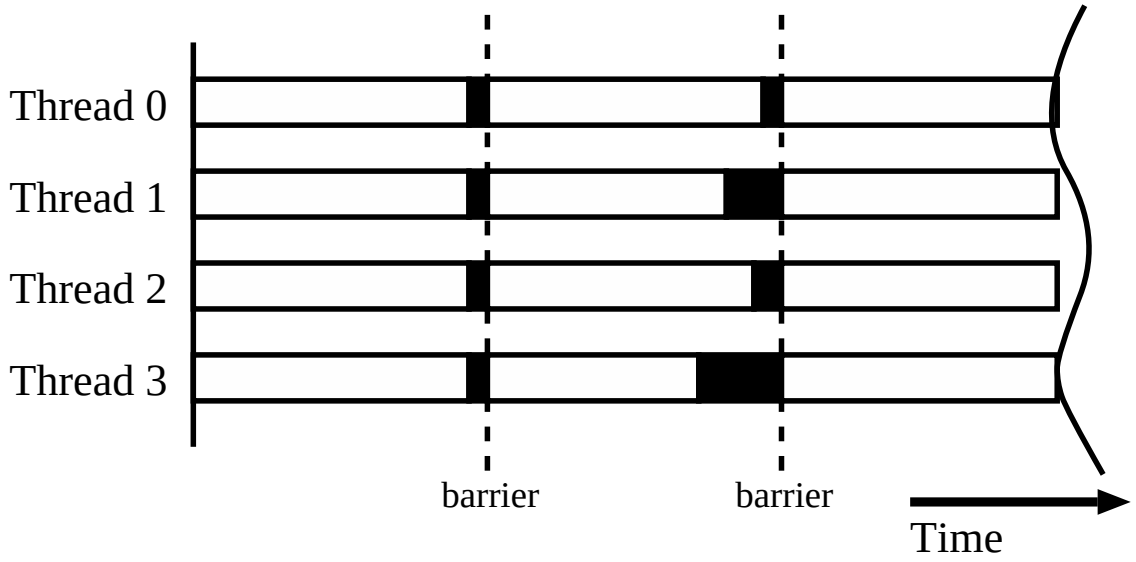

図 5.1: barrier synchronization facility

5.2.1 Exclusive Access to Shared Variable

最も簡単なバリア同期の実装方法は、各スレッドの合流時であるバリア地点で共有カウンタのアクセスを排他書き込みにし、カウントを減少させることである。バリア地点に達したスレッドは、排他書き込みを開放し、カウンタが 0 になるまで動作待機する。このアルゴリズムには、問題点が存在する。

- カウンタに対する書き込みが集中した場合、同時実行には制限が掛かる。もし、n スレッドが同時にアクセスしたならば、O(n) の時間が掛かることになる。
- カウンタの再初期化のタイミングに問題ある。カウンタが目的の値に達したことを全てのスレッドに確実に知らせる必要がある。しかし、スレッドが進行するため、その内のどれかが前の処理中に取り残されるかも知れない。

このバリア同期の概略図を図 5.2 に示す。

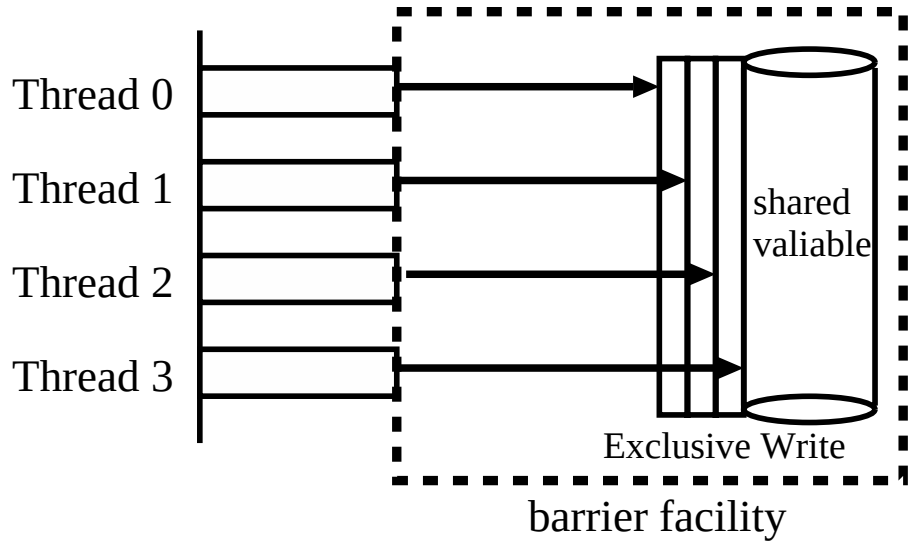

図 5.2: exclusive barrier

5.2.2 Dissemination Barrier

このアルゴリズムは Han ら [10] によって示され、$log_2 n$ の循環で、n スレッドのバリア同期を実現している。

i 循環の間にスレッド T は、スレッド $2^i + t(mod\ n)$ に、共有変数を通じて情報を伝達する。スレッドが i 循環の間に送られてくる情報を受けとるために待ち、$i+1$ 循環でも情報が送られてくるのを待つ。その後、循環のすべての情報を受け入れた場合、すべてのスレッドはちょうど $log_2 n$ の循環で全ての他のスレッドの情報を受けとることになる。

このバリア手法では、書き込みにおいて排他処理が存在しない。そのため、Exclusive Access to Shared Variable のように、全てのスレッドがバリア地点に同時に入った場合でも、O($log_2 n$) の時間でバリア同期が成立する。また、再初期化問題はこの同期手法にも存在する。そのため、ダブルバッファリングを用いこの問題を解決している。

このバリア同期の概略図を図 5.3 に示す。

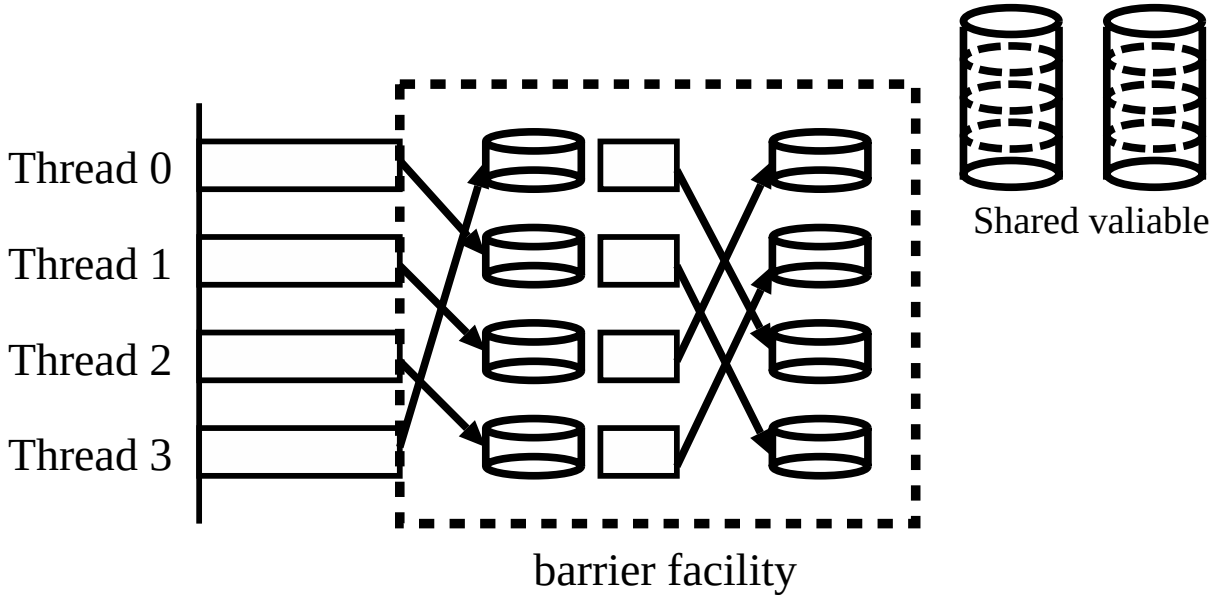

図 5.3: dissemination barrier

5.2.3 Tree Barrier

本稿で使用した Tree Barrier は、Yew ら [7] の提案した The Software Combining Tree Barrier を Mellor-Crummey ら [6] が改良した方法を用いた。まず、Yew らは、同期変数アクセスのための回線の争奪を減少させるために、結合ツリー (Combining Tree) のデータ構造を提案した。このデータ構造では、多段相互結合網で結合されたハードウェアのように、ソフトウェア結合ツリーは単一参照のように同じ共有変数に多重参照が行なえる。

各プロセッサは、1 グループがツリー構造のリーフに割り当てられ、各グループに分割される。各プロセッサはリーフの状態をアップデートする。そうするためには、グループ内の最後のスレッドがその発見を司るとするならば、子集団のアップデートを反映して、親をアップデートする。大体の場合この操作は、一番遅くバリア地点に到着したスレッドが行なう。

- 各プロセッサはリーフで、リーフの変数を減算を始める。
- 各ノードに最後に到達したプロセッサが次のレベルを続ける。
- ツリーのルートに達するプロセッサはバリアのロックフラグをアップデートするとすぐに、各プロセッサは各ノードのパスを逆に進行し、同胞をブロックを解除していく。

このバリア手法で、メモリ競合をかなり減少させることが可能であるが、このスレッドブロッキングの方法では、スピンブロッキングに用いるメモリ領域が静的に決定できない。ブロードキャストベースのキャッシュ一貫性プロトコルのシステムでは、各スレッドはそれらがスピンブロックしているツリーノードのコピーを得れるかも知れない。しかし、プロセッサは遠隔地に存在するスピンブロック変数を参照してブロッキングすることになる。このことは、相互接続ネットワークの不必要な回線取得に繋がる。

そこで、スレッドがそれぞれユニークな位置にスピンブロック変数を持たせることで、おそらくローカルに変数アクセスが可能になる。また、このアルゴリズムでは、読み込みを除いて、どんな不可分命令も使用しない。プロセッサーメモリの接続には、必要最小限の命令しか渡されない。

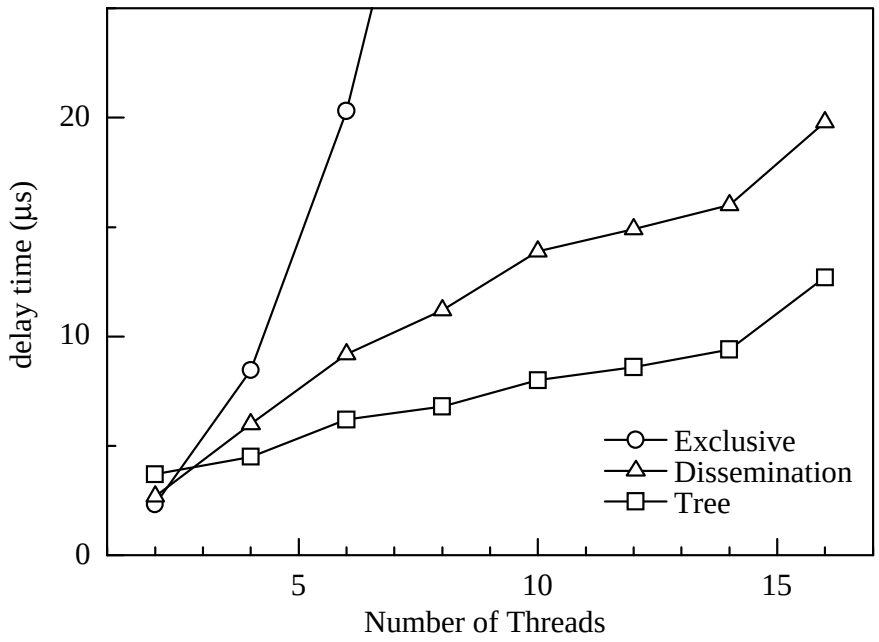

図 5.4: Barrier Delay Time on Starfire

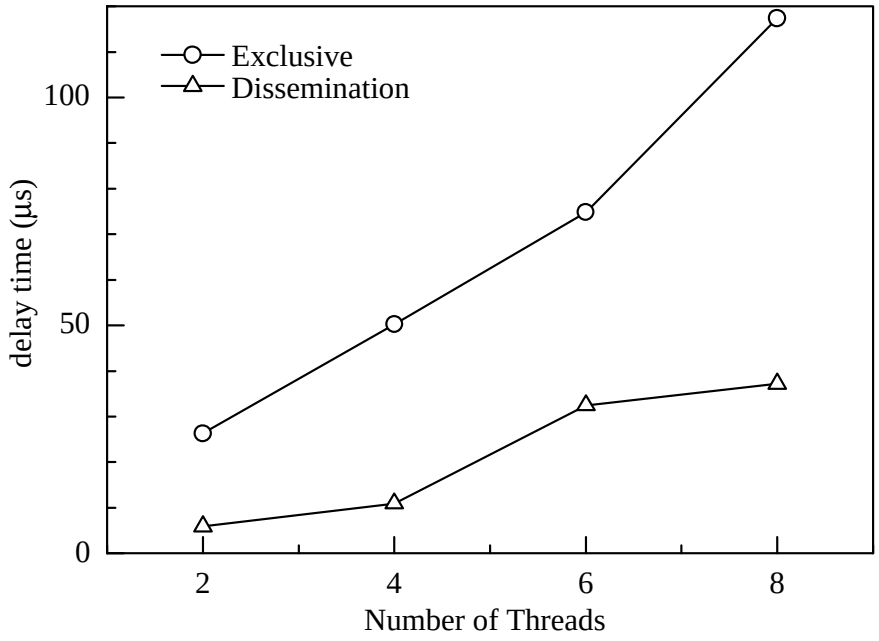

図 5.5: Barrier Delay Time on J90

5.3 局所同期

5.3.1 概要

分散メモリ型並列計算機では、1 対 1 通信では、送信完了と受信完了を待つことによって送信側と受信側の 1 対 1 の局所同期をとることが出来る。更に、送信バッファと受信バッファでは、各々が物理的に別の領域に存在するため、通信終了後であれば、そのデータに対する変更が可能であり、受信通信が起こる前に受信バッファのデータに対して変更は行なわれたとしても、相手に影響を与えることはない。

共有メモリ型並列計算機では、送信・受信バッファを分ける必要がない。また、バッファを設ける必要がない。そのため、バッファ間のコピーなど余計なオーバーヘッドがなくなる。しかし、この場合には、データの移動がないことから、数値の整合性を保つには、そのデータが書き込み可能なのか、読み込み可能なのか、それともデータの変更中なのかを判断する必要がある。

その方法として、局所的な同期を必要とするデータに対して、presence bit と同等の機能をするものを設け、その bit の更新を不可分に行なうことで前述の判断が行なえる。すなわち、bit に、書き込み可、読み込み可、更新中という状態を持たせ、その bit 更新に排他制御を行ない不可分性を持たせる。

5.3.2 実装

前述のように実装には、有限バッファの生産者/消費者問題であるため、セマフォを二つ用いるか、同等の機構を排他制御を用いて実装することになる。しかし、セマフォを用いると効率が悪くなるため、排他制御を用いて実装を行なう。この場合のブロックには条件変数を用いる方法と spin-wait を用いる方法が存在する。しかし、条件変数を用いた場合ブロックするスレッドをスケジューリングから外す (sleeping) よう実装されている場合がある。スレッドをスリープする場合のオーバーヘッドが非常に大きくなる場合がある。これを回避するためにブロックが必要かどうかを繰り返し検査することで回避できる。これは spin blocking と呼ばれる手法である。ただし、この手法はスレッドプライオリティにおいて、ウェイクアップの順番に制限がない場合においてのみ適用できる。局所同期では、同期の対象がすべて 1 対 1 であるため、順番に関しての制約はないため、spin-wait を用いて実装を行なう。

局所同期機構は、変数の初期化関数と rd_lock()、rd_unlock()、wr_lock()、wr_unlock()

という5つの関数で構成されている。これらの関数をリードあるいはライトを他のスレッド動作によって変更させたくない部分をはさむことによって局所同期を実現する。関数の動作の概要を図5.8に示す。

前節のバリア同期では、同期する場合でのリソース資源の効率は良い。しかし、実行スレッド全体に同期を掛けるため、同期の実行コストが高くなり、同期の実行が頻発するような場合や、全スレッド中の1スレッドの遅れが、全体パフォーマンスに影響を及ぼす可能性がある。また、全体同期のみでは、プログラムの記述に柔軟性を欠くことになるが、局所同期を用いることによってそれらはある程度以上解決される。

局所同期のプログラムへの適用例を図5.6に示す。thread_id を 各スレッド番号とすると、wr_lock() と wr_unlock() が MPI_Send() に当たり、rd_lock() と rd_unlock() が MPI_Recv() に当たる。thread_id = 1 は、thread_id = 0 の wr_unlock() が終了するまで、rd_lock() によってスレッドの実行がブロックされて、変数 count にはアクセス出来ない状態にある。このプログラムをバリア同期を用いて記述した例を図5.7に示す。バリア同期では、同期は部分でなく領域に対して行なう。そのため、プログラミングは簡単になる。

また、変数 count は、各スレッドに対して同一アドレスに存在し、同一変数へのオペレーションであるため、分散メモリシステムのようにローカルアドレスに直す必要はない。このことは、配列操作の多い科学技術計算の並列プログラミングにおいて大きな効果が期待できる。

MPI の Pingpong ベンチマークのように、図5.6の実行時間の計測を行ない、その結果を表5.1に示す。分散メモリシステムにおいてこのコードを実装する場合、Send 側で count++ を実行し、その count 変数を MPI_Send() で Recv 側に送り、count– を実行する。その後その値を使用すべき PE に送る必要があるかもしれない。共有メモリ型並列計算機では上述のように、そのような操作は全く必要ない。この結果、J90 は Starfire の15倍程度遅い。また、Starfire では、図4.1に示されるメッセージ通信を用いた場合、通信時間はミリ秒のオーダーとなり、局所同期を用いた場合非常に高速なことが分かる。

表 5.1: Performance of Local Synchronization Facility

	Time(μs)
Starfire	4.86
J90	72.61

```
if(thread_id == 0){
     wr_lock(local_sync);
     count++;
     wr_unlock(local_sync);
} else if(thread_id == 1){
     rd_lock(local_sync);
     count--;
     rd_unlock(local_sync);
}
```

図 5.6: For example Programming for local synchronization

```
if(thread_id == 0){
     count++;
     barrier(bt);
} else if(thread_id == 1){
     barrier(bt);
     count--;
}
```

図 5.7: For example Programming for barrier synchronization

```
typedef struct{
   mutex_t   m;
   valatile int  p_bit;   /*READ_OK, UPDATE, WRITE_OK*/
} st;
```

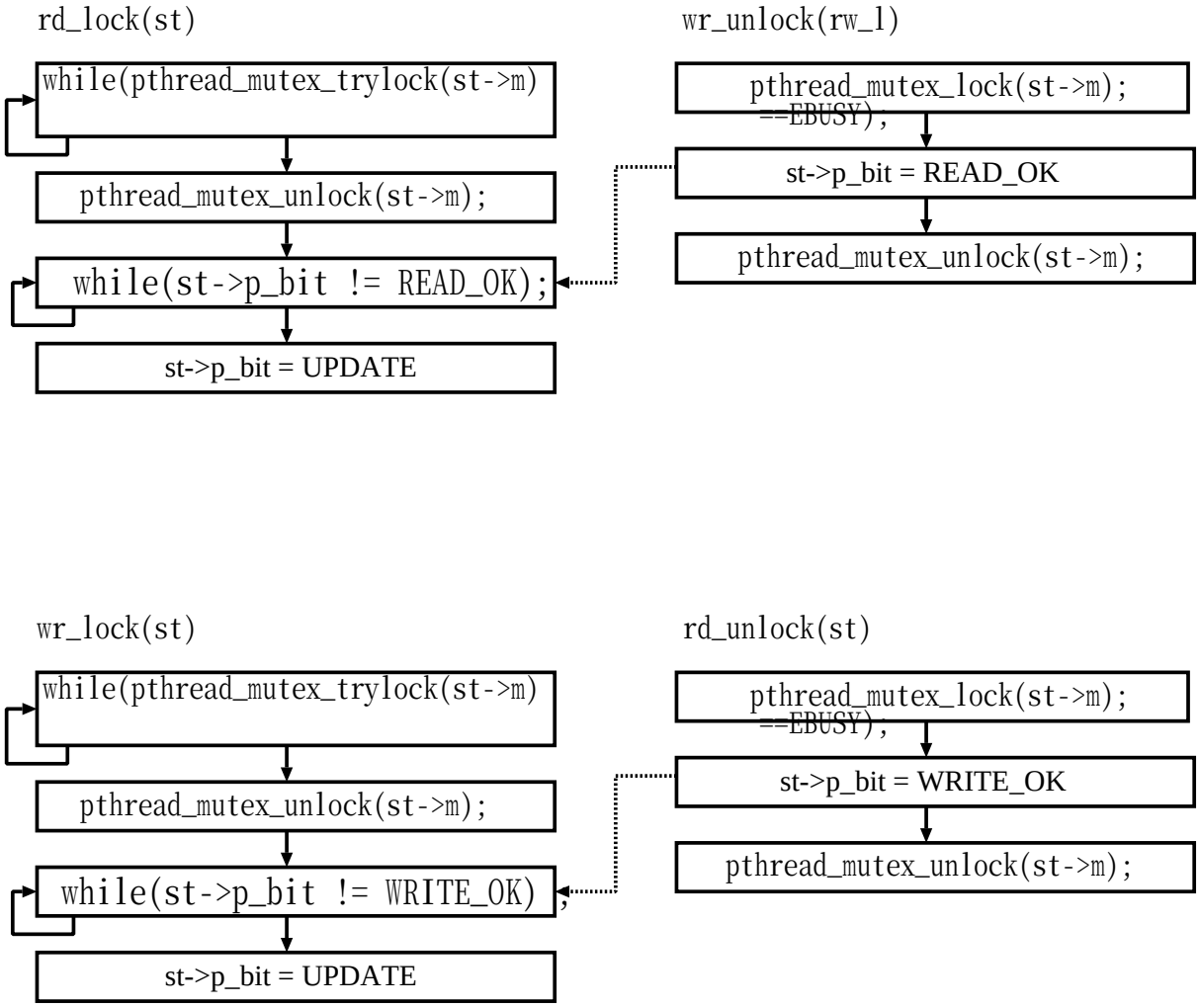

図 5.8: Local synchronization facility

第 6 章

予 備 実 験

流体解析において最も並列化の効果を発揮して欲しい部分は、圧力解を得るために必要な Poisson 方程式の解法部分である。そのため、分散メモリシステムでは、領域分割法に基づいて、反復法 (SOR 法や CG 法) や直接法 (LU 分解) をよる解析を行なう。ただ直接法は数値的な鈍りが生ずるため本稿では反復法を考える。

本稿では、SMP システムおよび共有メモリシステム全体においてパフォーマンス向上が望める方法の検討を行なう。CG 法はベクトル演算が主となり、高速なメモリアクセスの能力が必要となる。つまり、PVP などのベクトル計算機やメモリシステムが強化されたシステムにおいて有効である。また、SMP システムでは、データ参照の局所性を引きだし、キャッシュの再参照性を上げて、メモリ参照能力を疑似的に上昇させ RISC プロセッサの能力を引き出すプログラミングが必要である。そのためには、巨大なベクトル演算を行なわねばならない CG 法のような反復解法は SMP システムでは有効ではない。そこで、SOR 法に基づく解法によって、圧力の Poisson 方程式を高速に解く方法を検討する。

6.1 SOR 法への同期機構の適用

有限差分法で離散化した場合の SOR 法による解法では、第 4 章の熱流体ベンチマークで述べたように、データ参照の局所性を引き出す必要がある。しかし、ただデータ参照の局所性を出すだけであれば、領域分割に基づく SOR 法で十分である。また、データ分割の弊害として収束性の悪化が見られる。しかし、この手法では、データ交換用のバッファを確保する必要があり、共有メモリとスレッドプログラムの有利さを発揮できない。

そのため、SOR 法と解くモデル問題として、図 6.1 のコードの並列化を考える。

```
for(i=1 ; i<N ; ++i)
   for(j=1 ; j<N ; ++j)
      a[i][j] = 0.25 * (a[i-1][j]+a[i][j-1]
                     +  a[i][j+1]+a[i+1][j]);
```

図 6.1: Computation Model

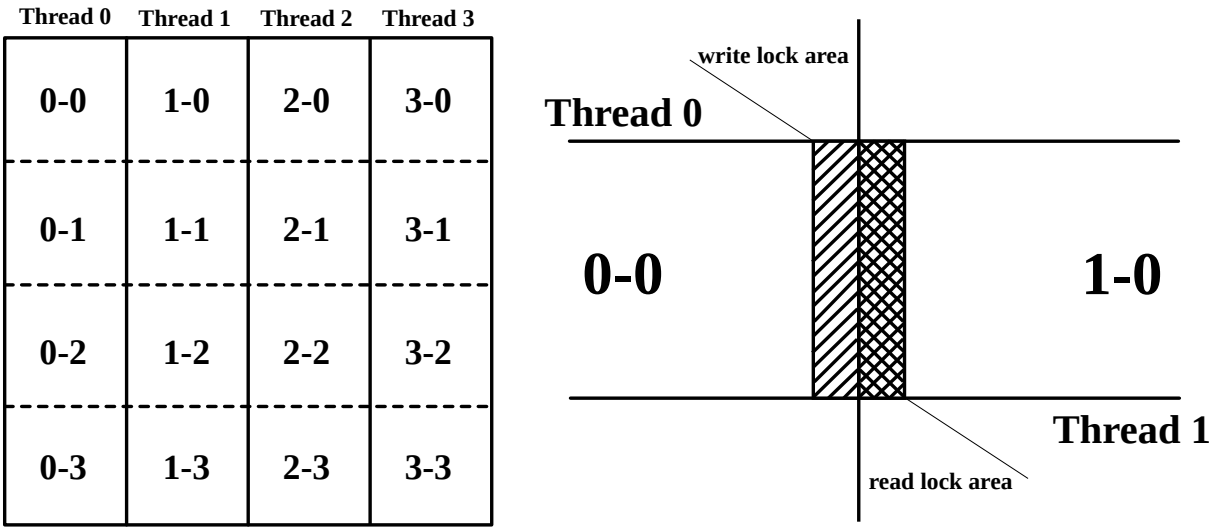

図 6.2: Parallel Computation (local synchronization)

表 6.1: Pipeline Processing

	1	2	3	4	5	6	7	8	·····	·····	n-3	n-2	n-1	n
Thread 0	0-0	0-1	0-2	0-3	0-0	0-1	0-2	0-3	·····	·····	0-3			
Thread 1		1-0	1-1	1-2	1-3	1-0	1-1	1-2	·····	·····	1-2	1-3		
Thread 2			2-0	2-1	2-2	2-3	2-0	2-1	·····	·····	2-1	2-2	2-3	
Thread 3				3-0	3-1	3-2	3-3	3-0	·····	·····	3-0	3-1	3-2	3-3

このモデル問題では、各反復において a[i-1][j] と a[i][j-1] のデータに依存関係が存在する。そこで図 6.2 のように、a 行列の計算領域をブロックに分けて、ブロックをパイプライン処理によって並列化を行なう。

次に表 6.1 に図 6.2 のパイプライン処理の流れを示す。0-0 と 1-0 を基に図の説明を行なう。0-0 と 1-0 において、1-0 が 0-0 の前に処理されない限り、このパイプライン処理は自然順序で処理するのと同等である。そのために、局所同期では、Thread 0 の処理部分の Thread 1 の処理部分に接している領域を wr_lock() と wr_unlock() で囲む。また、Thread 1 の Thread 0 に接している領域を rd_lock() と rd_unlock() で囲む。この操作で、Thread 0 が write lock area を処理し終るまで、 Thread 1 は、read lock area を処理し始めることが出来ない。よって、全体処理と関係なく Thread 0 と Thread 1 の関係において処理が行なえる。すなわち、非同期のパイプライン処理が行なえることになる。

バリア同期を用いた場合は、図 6.3 のようになる。前述と同様に 0-0 と 1-0 を基に図の説明を行なう。0-0 が実行されている間、1-0 が処理されてはならない。1-0 の処理は 0-0 の barrier が発行するのを待つ必要がある。このため、Thread 1, 2, 3 は、処理が始まる前に段階に、barrier を それぞれ 1, 2, 3 回実行する必要がある。逆に処理終了後には Thread 0, 1, 2 が、それぞれ 3, 2, 1 回 barrier を実行する必要がある。

よって、barrier を用いた場合でも、表 6.1 のようにパイプライン処理が実行できる。この場合 barrier を用いているため、全体同期となり、完全同期のパイプライン処理を行なうことになる。

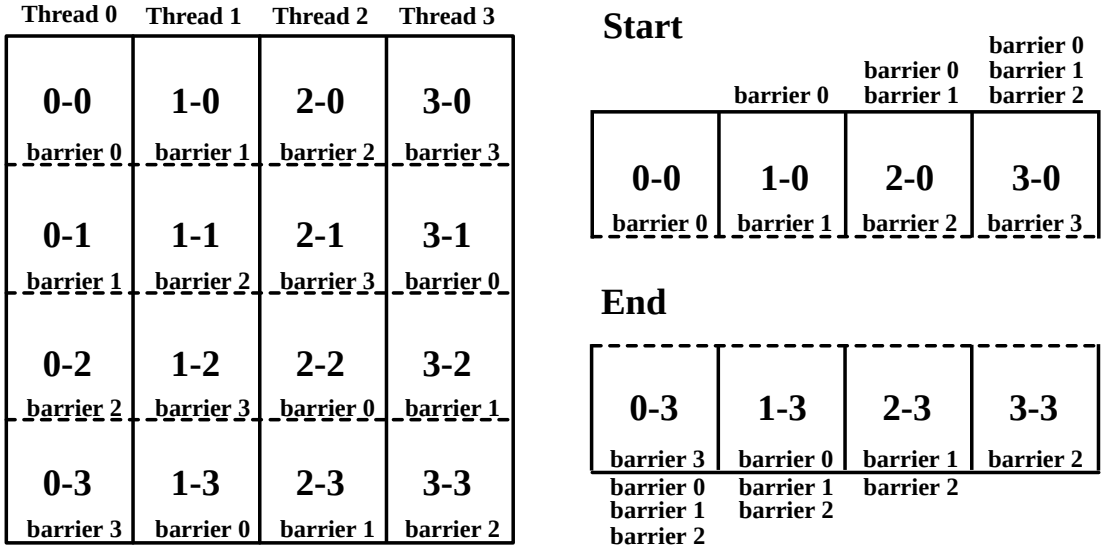

図 6.3: Parallel Computation (barrier synchronization)

6.2 評価方法

並列化の評価には、経過時間 $Time$(T)、並列加速率 $Seedup$、並列化効率 $Efficiency$ で評価した。

$$Speedup = \frac{T_1}{T_N} \qquad\qquad Efficiency = \frac{S_N}{N}$$

ただし、N はスレッド数、T_1 は 1 スレッドによる逐次計算をした計算時間、T_N は N 個のスレッドにより並列計算をした場合の計算時間である。

前記の並列手法を用いて、図 6.1 の並列計算を行なう。配列要素は倍精度乱数を与え、配列の大きさは N=130, 514 とする。

Starfire を用いた場合、使用するスレッド数は 2, 4, 8, 16 とし、1000 ステップの経過時間の計測を行なった。また、同様のパイプライン処理を領域分割したものを MPI ライブラリで実装したものとの結果を合わせて示す。MPI を用いた実装方法は、バリア同期によるものではなく、局所同期的なものを用いた。すなわち、wr_lock(),wr_unlock() で囲んだ領域を計算し、隣の PE に送り、その結果を受けて、rd_lock(),rd_unlock() で囲んだ領域を計算し、その結果を戻す。J90 を用いた場合、使用するスレッド数は 2, 4, 8 とし、並列化手法は前述の局所同期、バリア同期を用いたものを計測した。

図 6.4、6.7 に計算時間 (Time)、図 6.5、6.8 に並列加速率 (Speedup)、図 6.6、6.9 に並列化効率 (Efficiency) をそれぞれ示す。

6.3 結果と考察

6.3.1 Starfire

バリア同期、局所同期ともに良い並列化が行なえた。問題が大きな場合には並列化効率が 1 を越えるものもある。これは前述のように問題が大きな場合 1 スレッドでの実行速度が遅いということとプライベートキャッシュが効率良く働いていると考えられる。

130 × 130 の場合、Exclusive バリアは同期コストが高く、並列度が上がるほど実行効率が下がっていく。また、スレッド数が少い場合、高速なバリア同期を行なった方が局所同期を用いた場合より高速な実行が行なえている。しかし、計算粒度が小さくなっているスレッド数が 16 の時では、局所同期の方が早くなっている。16 スレッド使用時では、バリア同期を用いた場合同期コストも高くなるため、局所同期を用いた方が効率が良くな

る。それは、1 スレッドの計算量が少くなるにつれて、同期コストが計算時間に影響を及ぼしているものと考えられる。局所同期を用いた場合、理想的には同期時間は 0 であるため、高並列化による計算粒度が小さくなった場合でも同期コストによる影響が少い。16 スレッド使用時の Exclusive バリアの場合を除いて、全ての場合で MPI を用いて行なった並列化と較べれば、明らかにスレッドを用いた場合の方が高速であることが分かる。

514 × 514 の場合、MPI を用いる場合とスレッドを用いる場合では、4 スレッドを越えた当たりから MPI の通信の影響が出て、各同期を用いる方法よりも遅くなっている。16 スレッド使用時の Exclusive バリアでは、MPI による結果よりも悪くなっているが、それ以外の同期方法では、ほとんど違いが見られない。バリア同期の同期コストが影響しない計算量では、同期方法の影響がほとんどないと考えられる。そして、この場合に並列化効率が 1 を越える。特に Excusive バリアと MPI の 16 スレッド使用時を除くすべての同期方法で、効率が 1 以上となっている。表 6.2 に、この場合の 1 スレッドのメモリ使用量を示す。メモリ利用量から推測すると、2 スレッド使用時には、まだキャッシュ内に収まるデータ量ではないが、1 スレッドに較べて半分のデータ量になっている。そのため、2 スレッド使用時には、4, 8, 16 スレッド使用時ほど並列効率は高くない。データ量が完全にキャッシュ内に収まっている 4,8,16 スレッド使用時にかなり高い並列化効率が見られる。すなわち、データ参照の局所性によるキャッシュの有効利用が影響しているものと考えられる。また、4 をピークに並列化効率が下がっていくのは、並列粒度の減少による同期コストのためと考えられる。

表 6.2: Memory size per 1 Thread

使用スレッド数	1	2	4	8	16
メモリ使用量 (KB)	2064	1032	516	258	129

6.3.2 J90

J90 では、1 スレッドの実行時間が遅くなることや、プライベートキャッシュの影響を考慮する必要はほとんどない。しかし、数パーセントであるが、並列化効率が 1 を越えるような領域が存在する。

130 × 130 の場合、dissemination バリアを用いた場合が最も早く。また、4 スレッドまでは、Exclusive バリアの方が局所同期を用いたものは、それよりも遅い結果が得られた。

この原因として局所同期は排他処理を多く行なわれている。前章の表 5.1 と図 5.5 に示される通り、局所同期で J90 での結果は Starfire の 18 倍以上遅い結果となっているが、バリア同期のパフォーマンスの違いは、数倍程度である。そのため、J90 の排他処理が遅いということが考えられる。しかし、バリア同期と局所同期には単純な比較において、大きな差位があるが、この問題に適用して得られた差は 10%未満であり、それほど大きな差にはなっていない。

514 × 514 の場合、どの同期方法を用いてもパフォーマンスに影響がない。これは計算時間が多くなり、同期時間の影響が計測時間にあまり影響しなくなったためと考えられるが、わずかに dissemination バリアの効率が良いという結果が出ている。2 スレッドの場合を見てみると、すべての同期方法で効率が 1 を越えている。原因として考えられるのは、2 スレッドを用いることでメモリ転送能力はほぼ二倍になりメモリ読み込み待ちの部分が少くなることと、計算量とメモリ転送能力および同期速度のトレードオフの結果得られたものと推測される。

しかしこの並列手法はベクトルには適していない。今回は Starfire で実行したコードをそのまま実行しているため、積極的なベクトル化を行なっていない。そして、ベクトル化を行なった場合だとこれほどの効率は出ないものと考えられるその理由として、この並列化手法は、ベクトル長を短くするものであり、J90 でのパフォーマンスのこれ以上の向上はあり得ない。

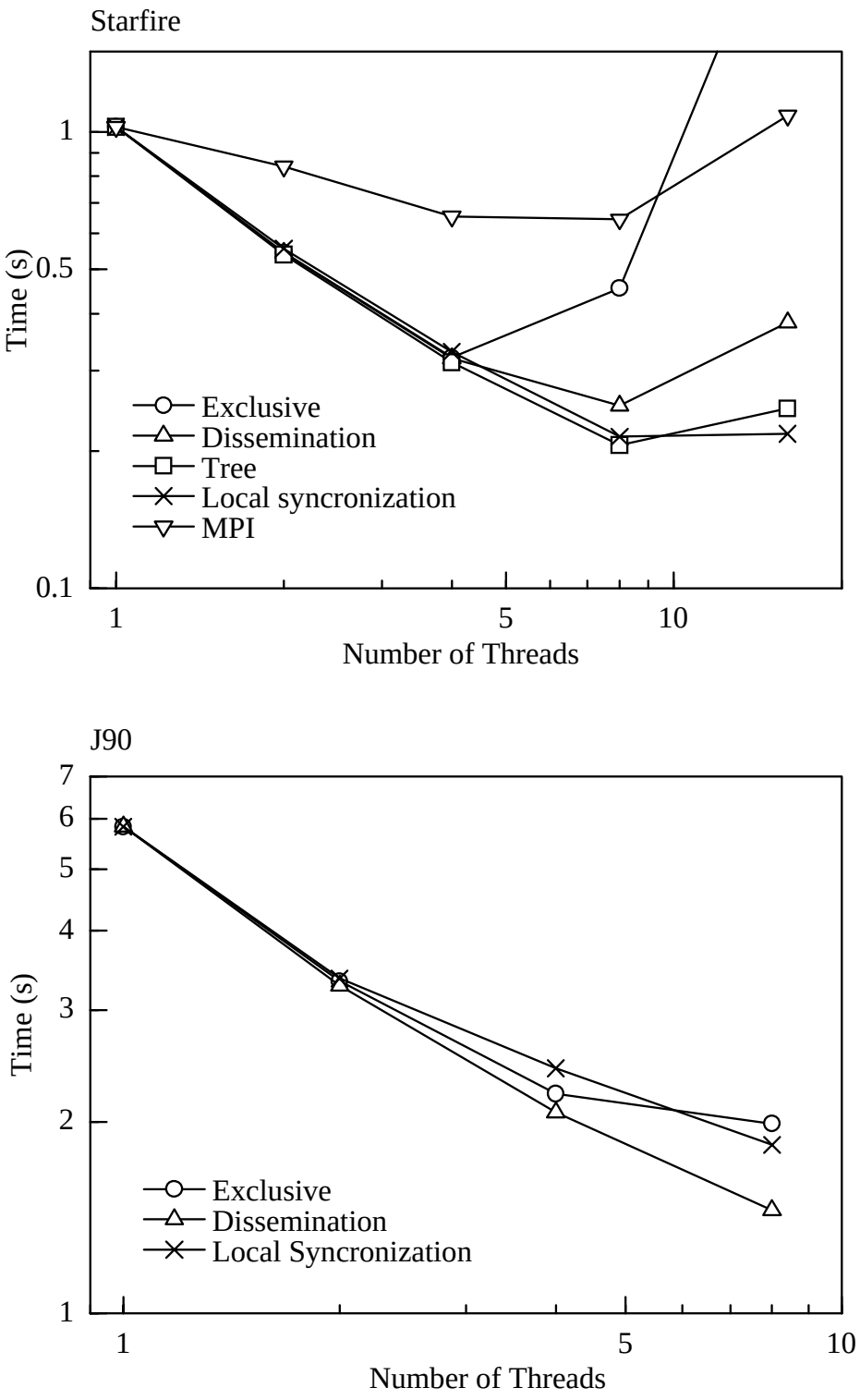

図 6.4: Computation Time 130 × 130 (upper:Starfire bottom:J90)

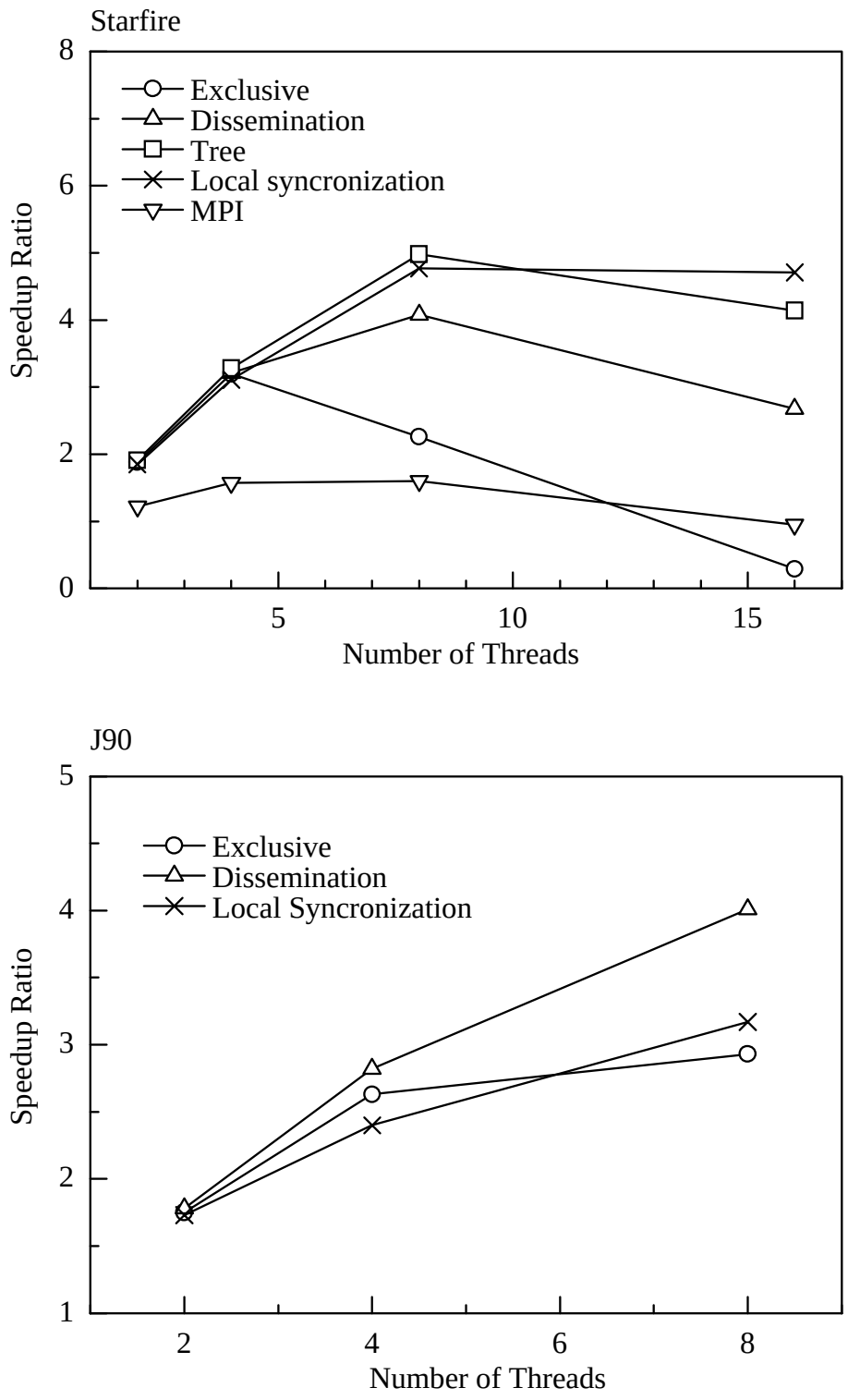

図 6.5: Speedup Ratio 130 × 130 (upper:Starfire bottom:J90)

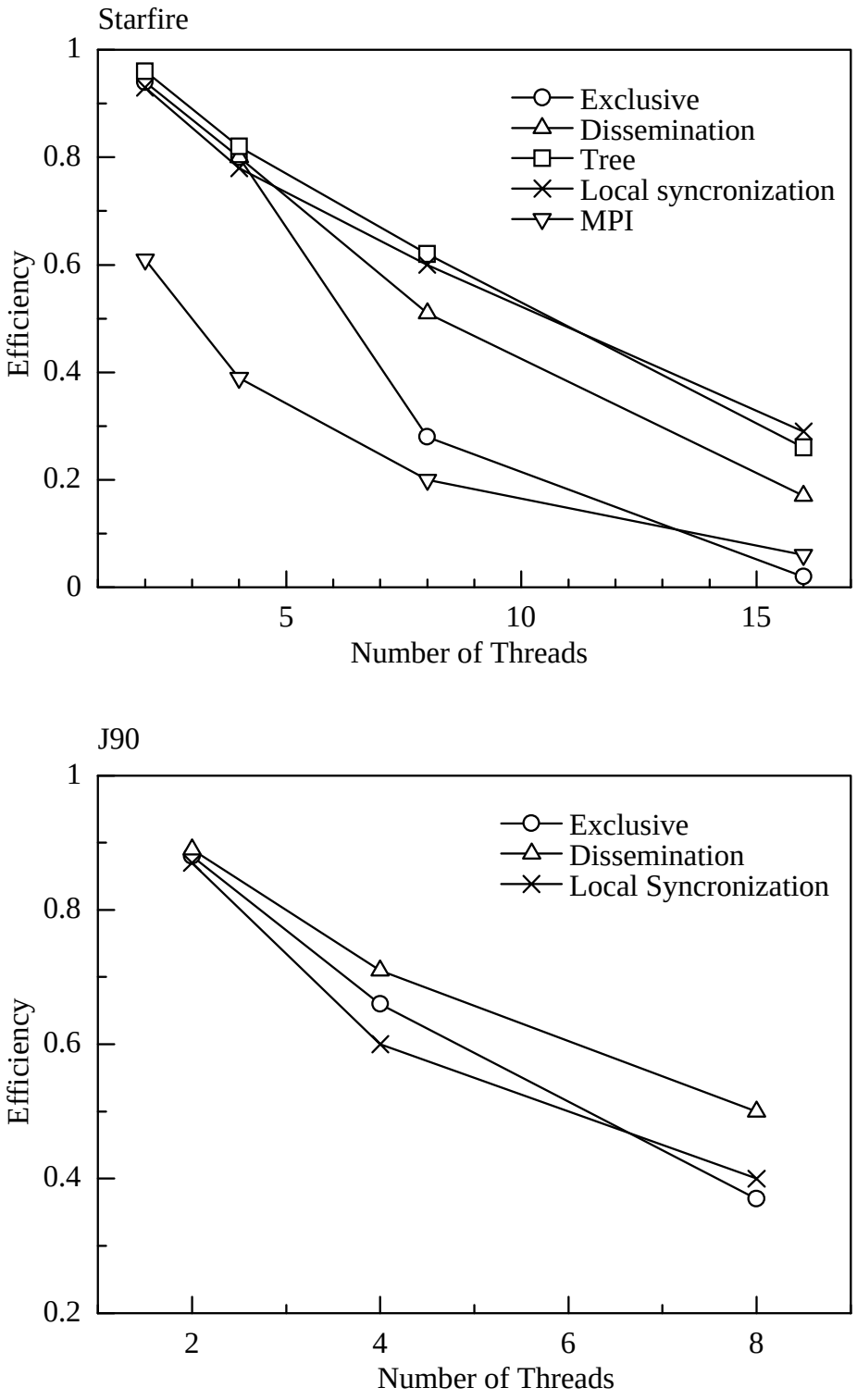

図 6.6: Efficiency 130 × 130 (upper:Starfire bottom:J90)

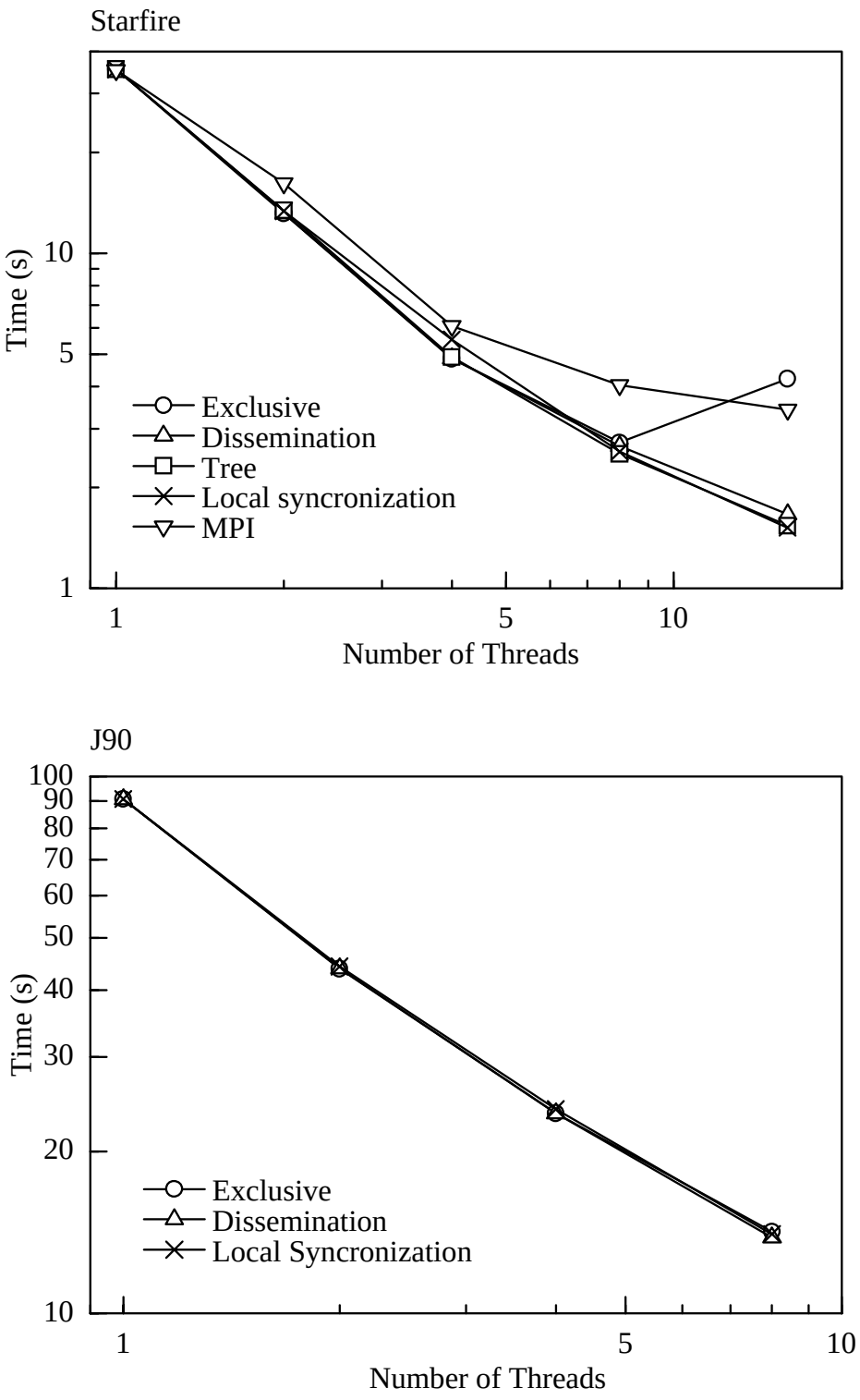

図 6.7: Computation Time 514 × 514 (upper:Starfire bottom:J90)

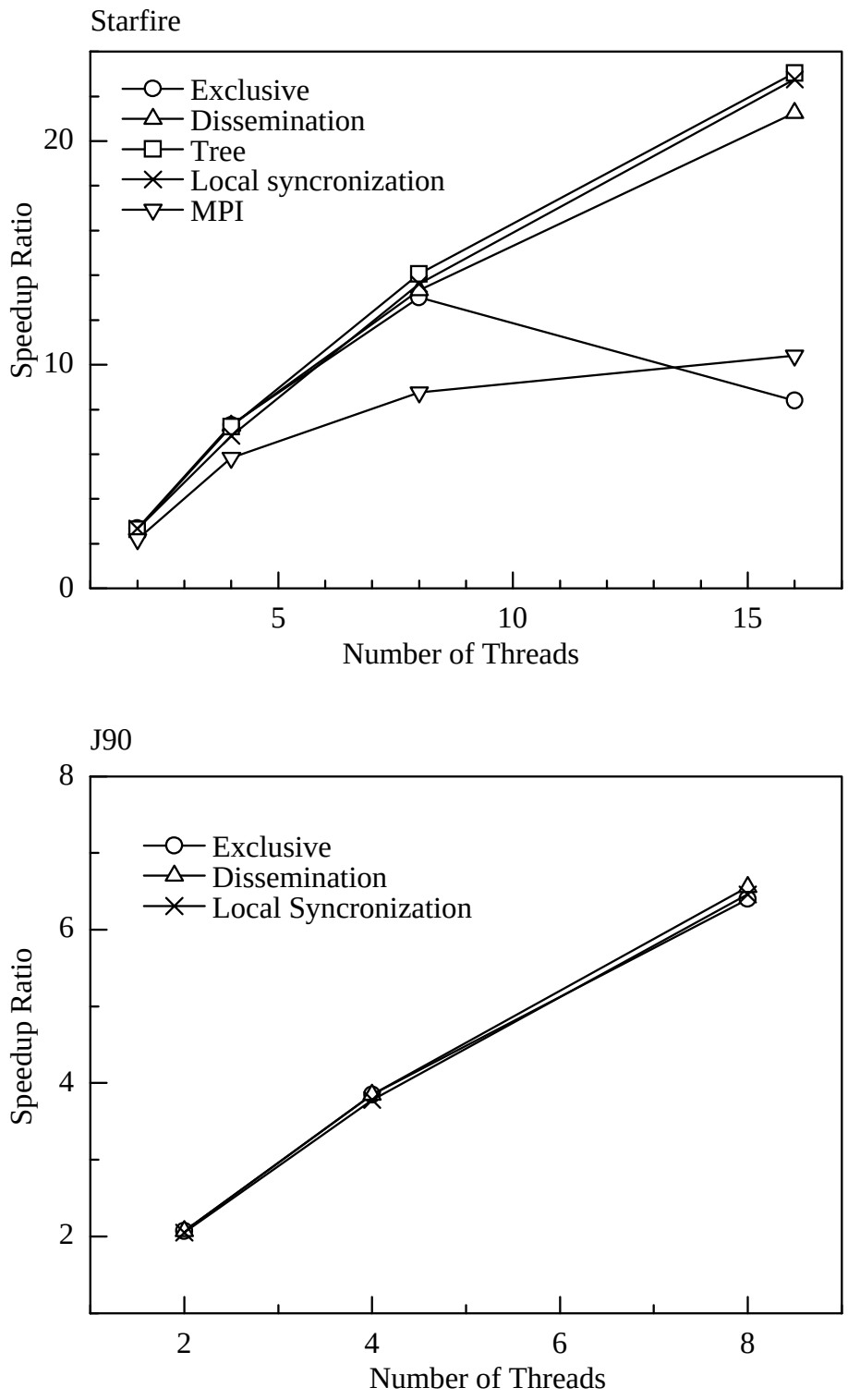

図 6.8: Speedup Ratio 514 × 514 (upper:Starfire bottom:J90)

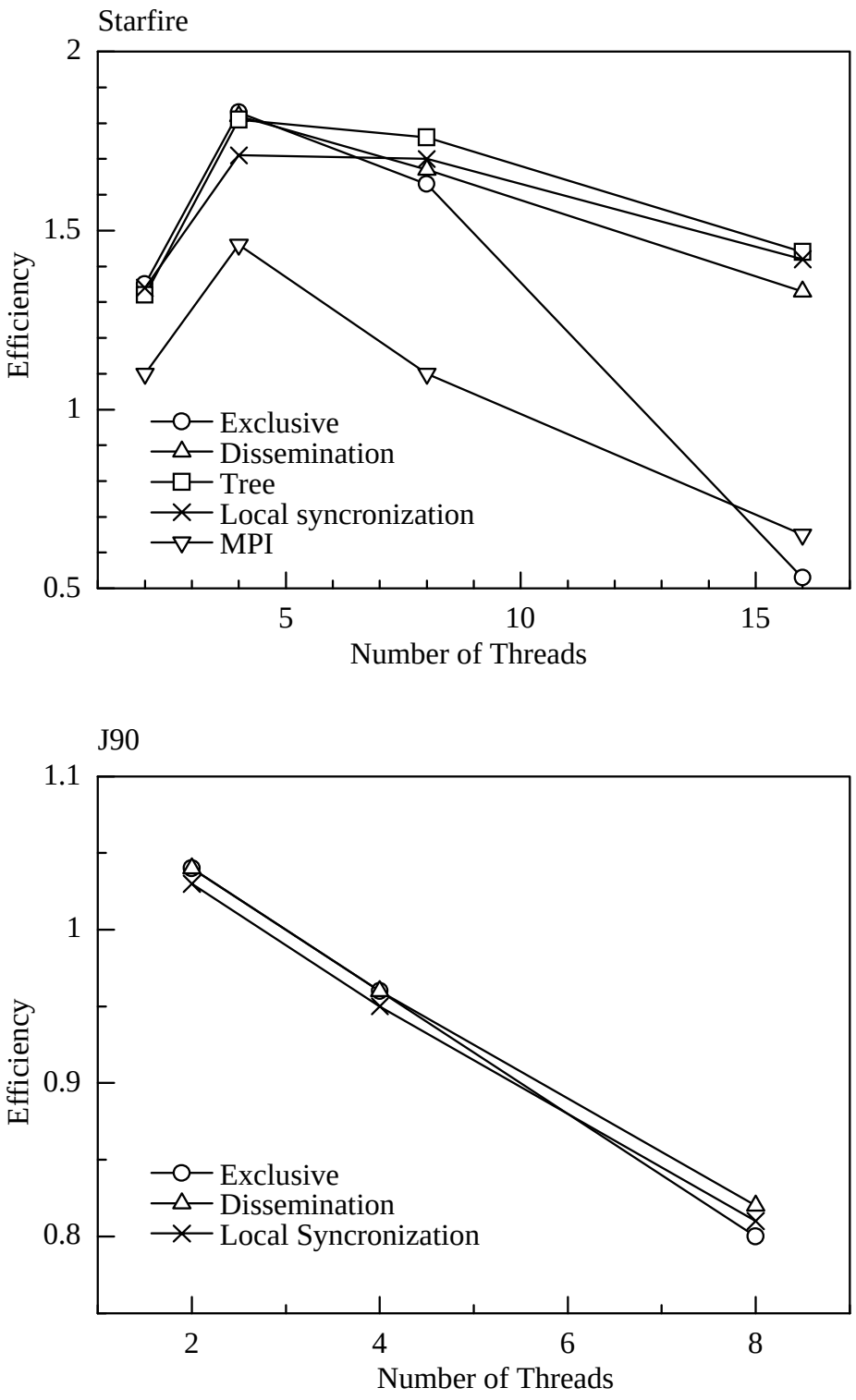

図 6.9: Efficiency 514×514 (upper:Starfire bottom:J90)

第 7 章

流 体 解 析

7.1 解 析 問 題

流体問題への適用を行なう。計算対象は、二次元非圧縮性粘性流体の Cavity 流れの解析を行なう。この流れ現象は、図 7.1 に示すように、上壁面を移動させることによって生じる正方形容器内の渦を計算するものである。この計算対象では、現在シミュレーション結果として最も正しいとされている、Chia ら [9] によるシミュレーション結果との比較検討が行なえ、比較的簡単な問題であるためコードの正確さの確認が容易に行なえる。

二次元非圧縮粘性流体のプログラムの基礎式は、連続の式と非圧縮粘性流れの Navier-Stokes(NS) 方程式である。

$$\begin{cases} \nabla \cdot V = 0 \\ \frac{\partial V}{\partial t} + (V \cdot \nabla)V = -\nabla p \frac{1}{Re} \Delta V \end{cases}$$

以下のような計算条件を用いた。

- 計算スキームには、MAC 法を用いた。MAC 法は Marker and Cell method の省略形で仮想的なマーカ粒子の動きを計算することで流体問題を扱う。連続の式を圧力の式の中に取り込み、安定的に計算が勧められるようになっている。
- 離散化は差分法を用い、差分格子はスタッガード格子を用いた。スタッガード格子を用いる利点は、一つのセルで連続の式が自然に表現でき、NS 方程式の性質が自然に表現できる。
- 対流項の計算には、三次の風上差分、それ以外の空間微分は二次の中心差分、時間微分は前進オイラーで離散化を行った。

- 圧力のポアソン方程式の解法には、SOR 法を用い、加速係数 1.6、収束判定 10^{-8} とした。
- 解析領域は、1.0 × 1.0 の正方領域、上辺に u=1(m/s) を与えた。
- Reynolds 数は 1000、分割数は 130 × 130。
- 時間進行は 1 Time Step あたり無次元時間で 5×10^{-3}、全 Time Step は 6000 step までとした。
- 各圧力反復の収束判定条件は無次元残差が 1.0×10^{-8} とした。

境界条件は、左右下壁面で、速度 $u, v = 0.0$、圧力 $\frac{\partial p}{\partial x} = 0.0$ 、上面で、速度 $u = 1.0, v = 0.0$、圧力 $\frac{\partial p}{\partial x} = 0.0$ とした。境界条件を図 7.1 に示す。

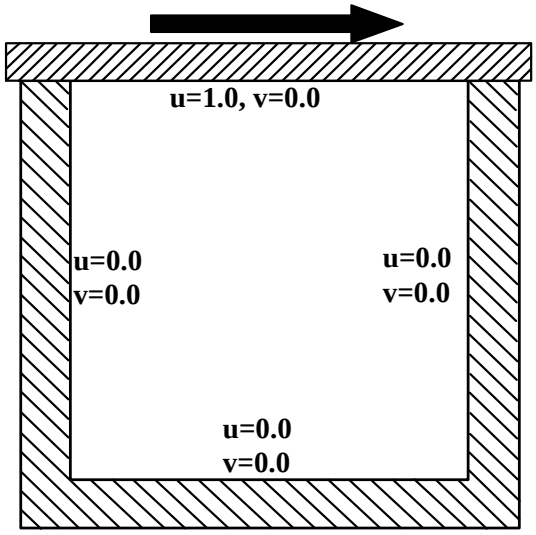

図 7.1: Flow Model Boundary condition

流体現象が定常に達した計算結果を流線 (図 7.2) とベクトル図 (図 7.3) に示す。並列化を行なった場合でもこの結果に変化はない。(ベクトル図のベクトルの大きさは Normalize された大きさである。)

並列化は、圧力の Poisson 方程式の解法に 上記のパイプライン処理による SOR 解法を用いた。その他の計算部分については、単純なループの分割で並列化が行なえる。使用するスレッド数は、Starfire では 2, 4, 8, 16 とし、J90 では 2, 4, 8 とする。並列化の評価

については、前章で用いた Time、Speedup、Efficiency を用いた。評価地点は、時間反復の 500、6000 step の経過時間を用いて、それぞれの値を評価している。実験の結果から、問題が大きい場合、本方法でもその他の方法でもかなり高効率な結果が得られることが実験的にも経験的にも明らかであるが、小さな問題でスレッド数を増やした場合、どの程度効率が低下するかが判断できるものか評価することができる。また、実験では、バリア同期と局所同期では、ほとんど違いがなく局所同期の有効性が疑問であったが、実際問題に局所同期を用いた場合にその効果が明らかにできる。

図 7.4、7.7 に計算時間 (Time)、図 7.5、7.8 に並列加速率 (Speedup)、図 7.6、7.9 に並列化効率 (Efficiency) をそれぞれ示す。

7.2 結　果

計測を行なった結果の全てに現れていることは、4 スレッドを越えた場合に Exclusive バリアを用いた場合の並列化の効果があまり現れないことである。そして、バリア同期の全てにおいて、並列度が増しても並列の効果が上がっていない。しかし、局所同期を用いた場合、並列度が増してある程度の効率の低下は見られるが、バリア同期を用いた場合よりも並列の効果が高いことが分かる。

また、500step の場合と 6000step の場合、グラフの概形に大きな変化は見られない。しかし、明らかに 500step の場合より 6000step の場合の方が全体的に並列の効果は下がっている。

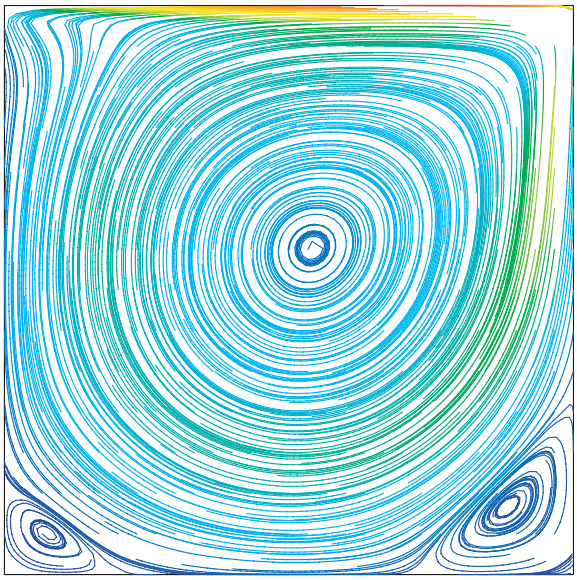

図 7.2: Streamline

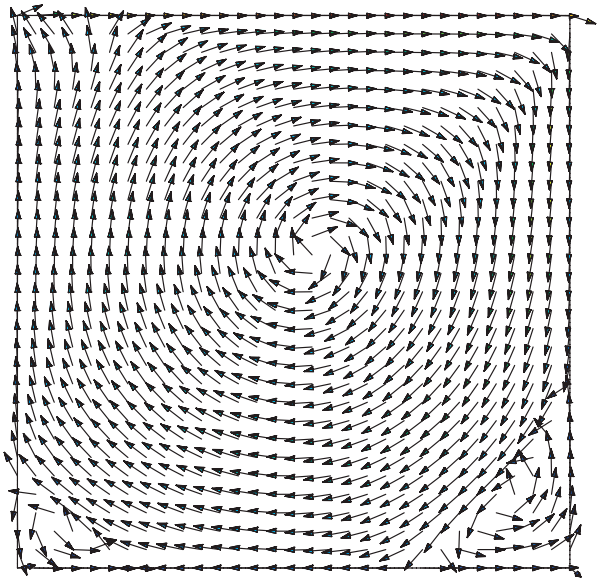

図 7.3: Flow Pattern

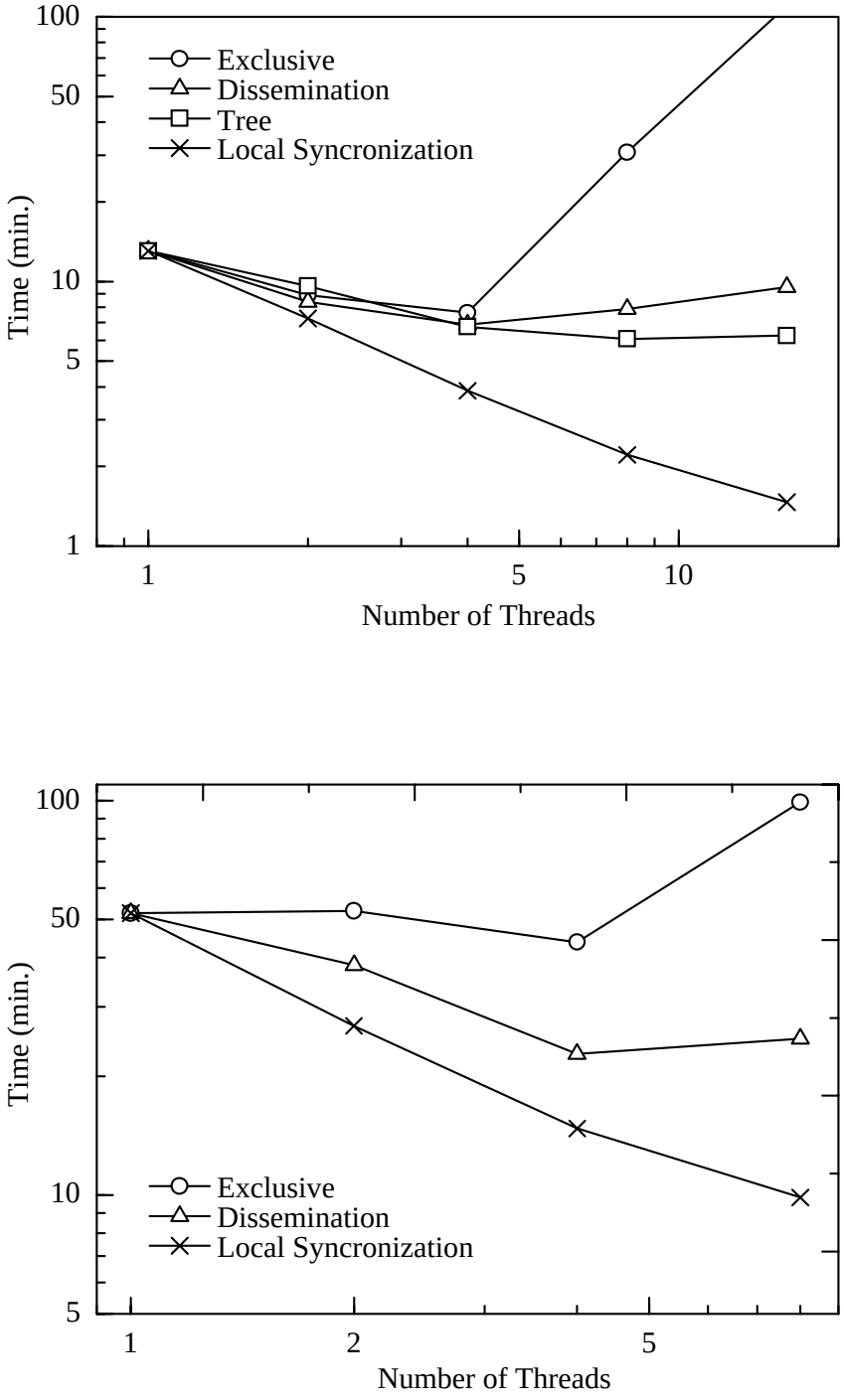

図 7.4: Computation Time (500step) upper:Starfire bottom:J90

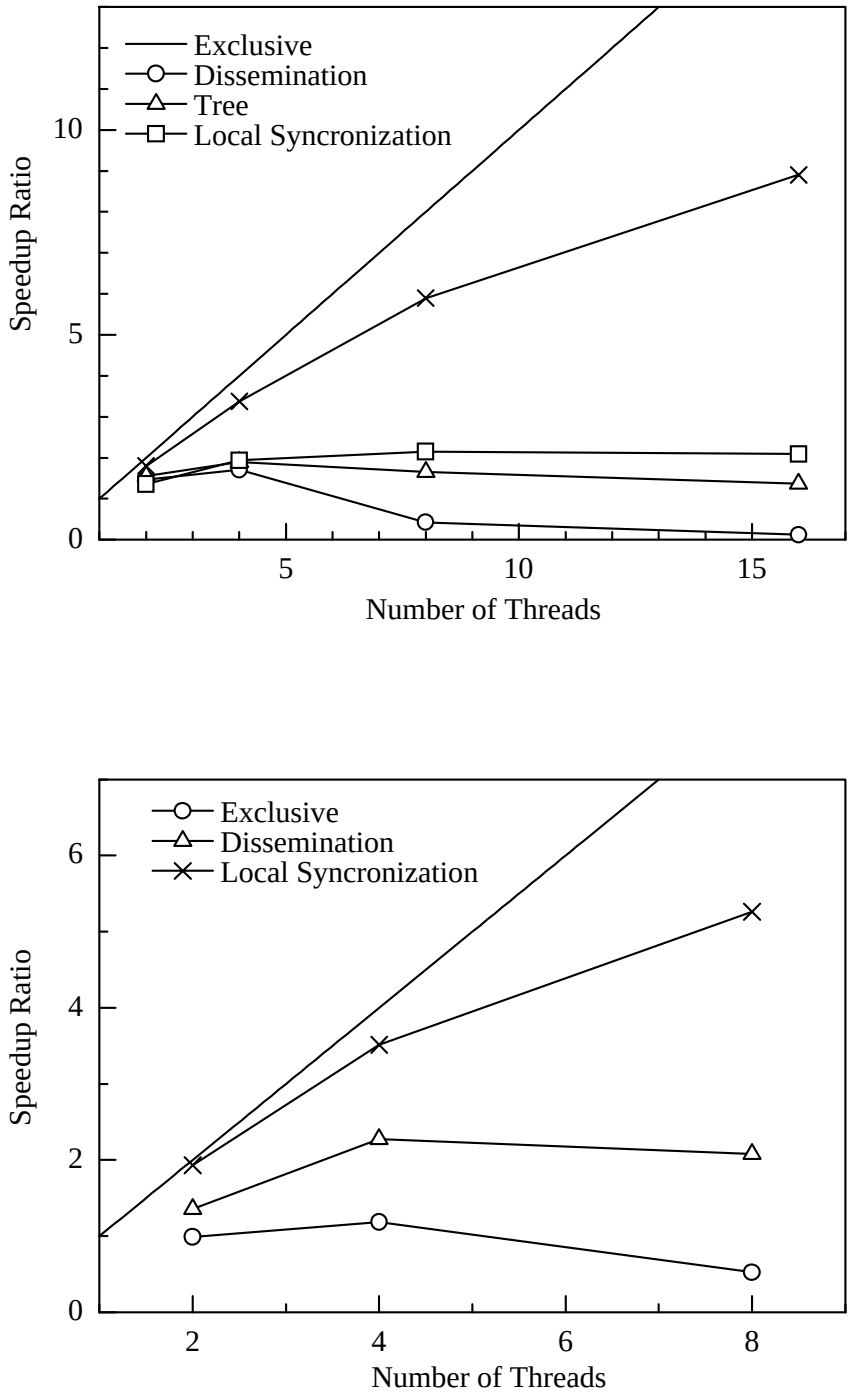

図 7.5: Speedup Ratio (500step) upper:Starfire bottom:J90

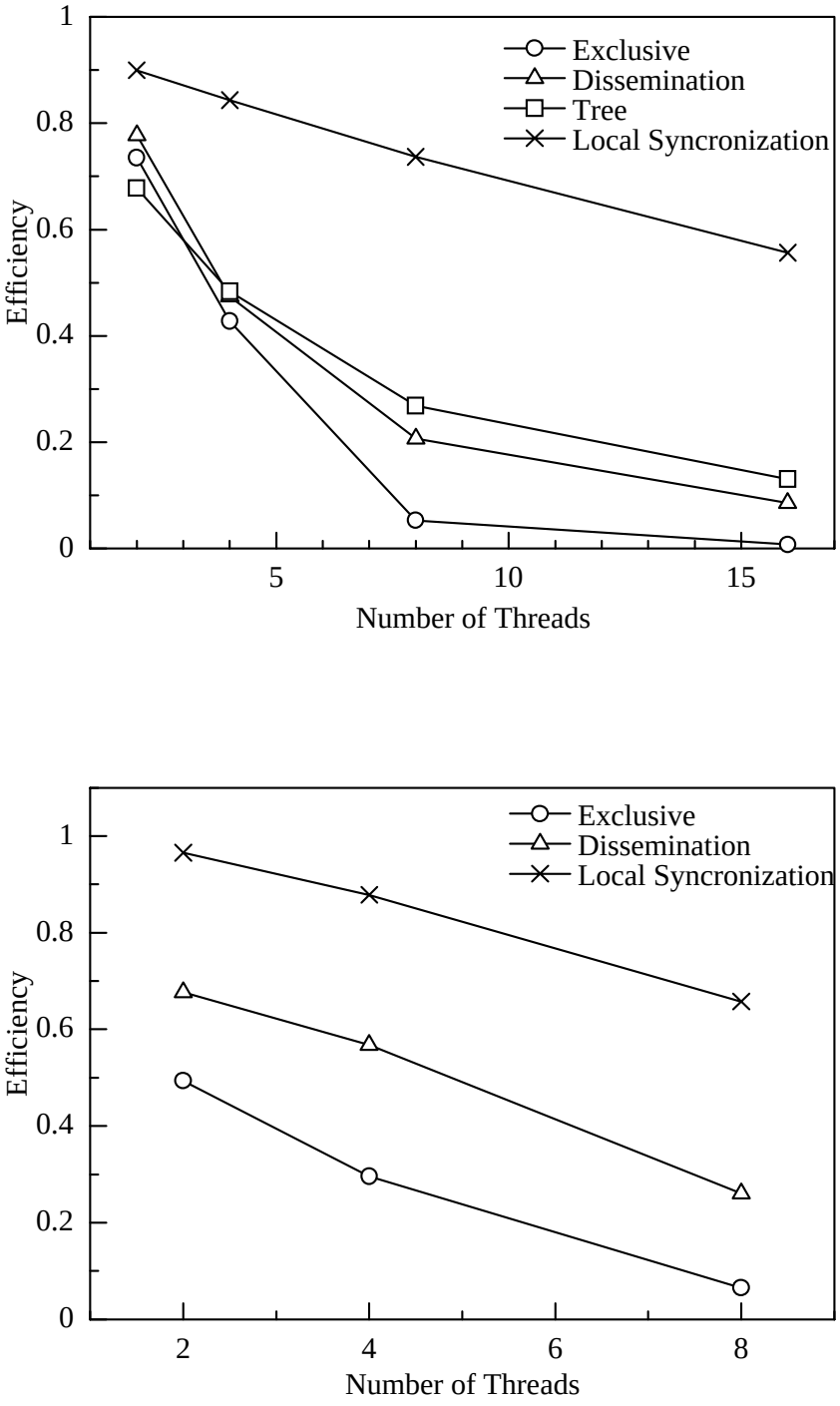

図 7.6: Efficiency (500step) upper:Starfire bottom:J90

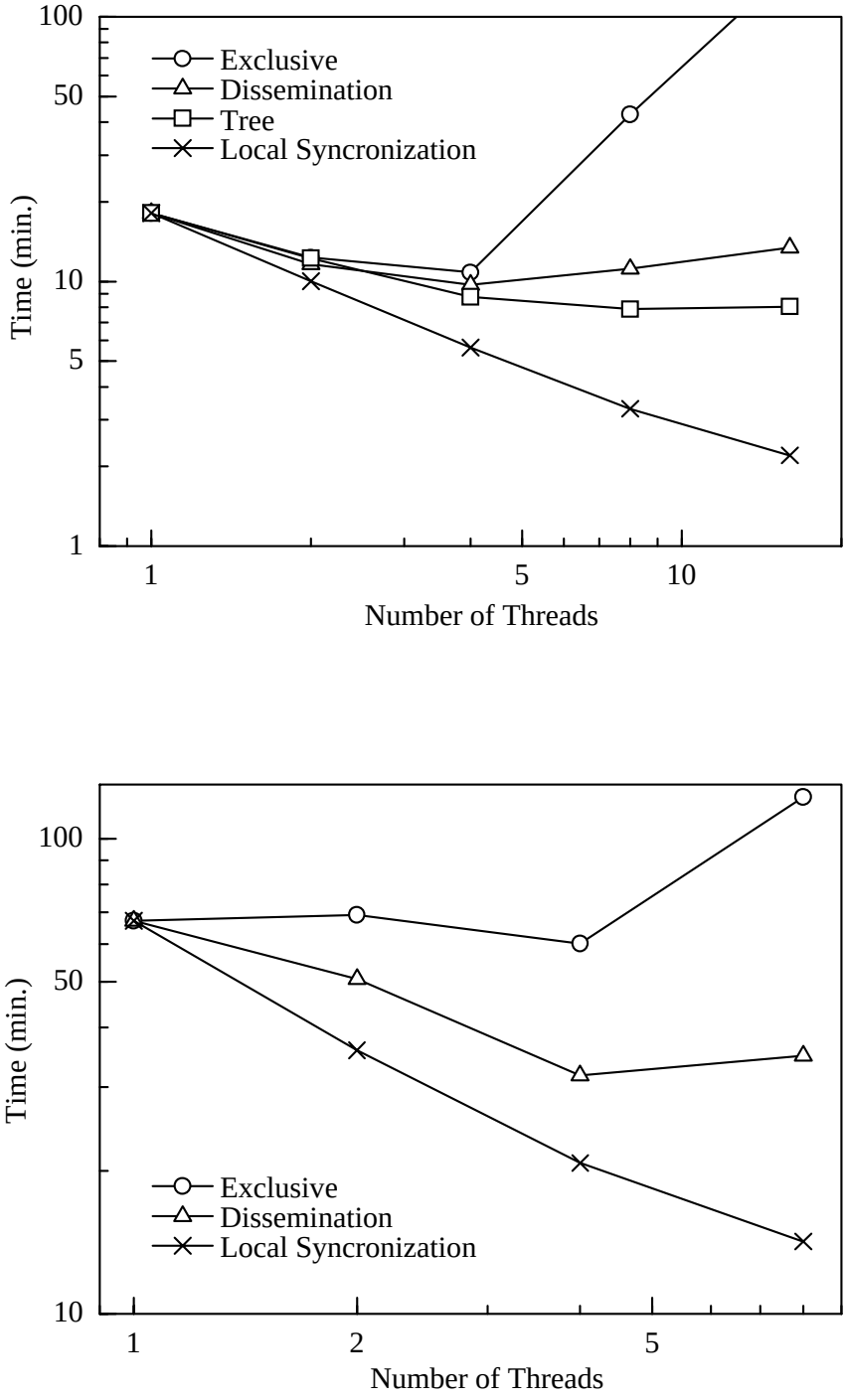

図 7.7: Computation Time (6000step) upper:Starfire bottom:J90

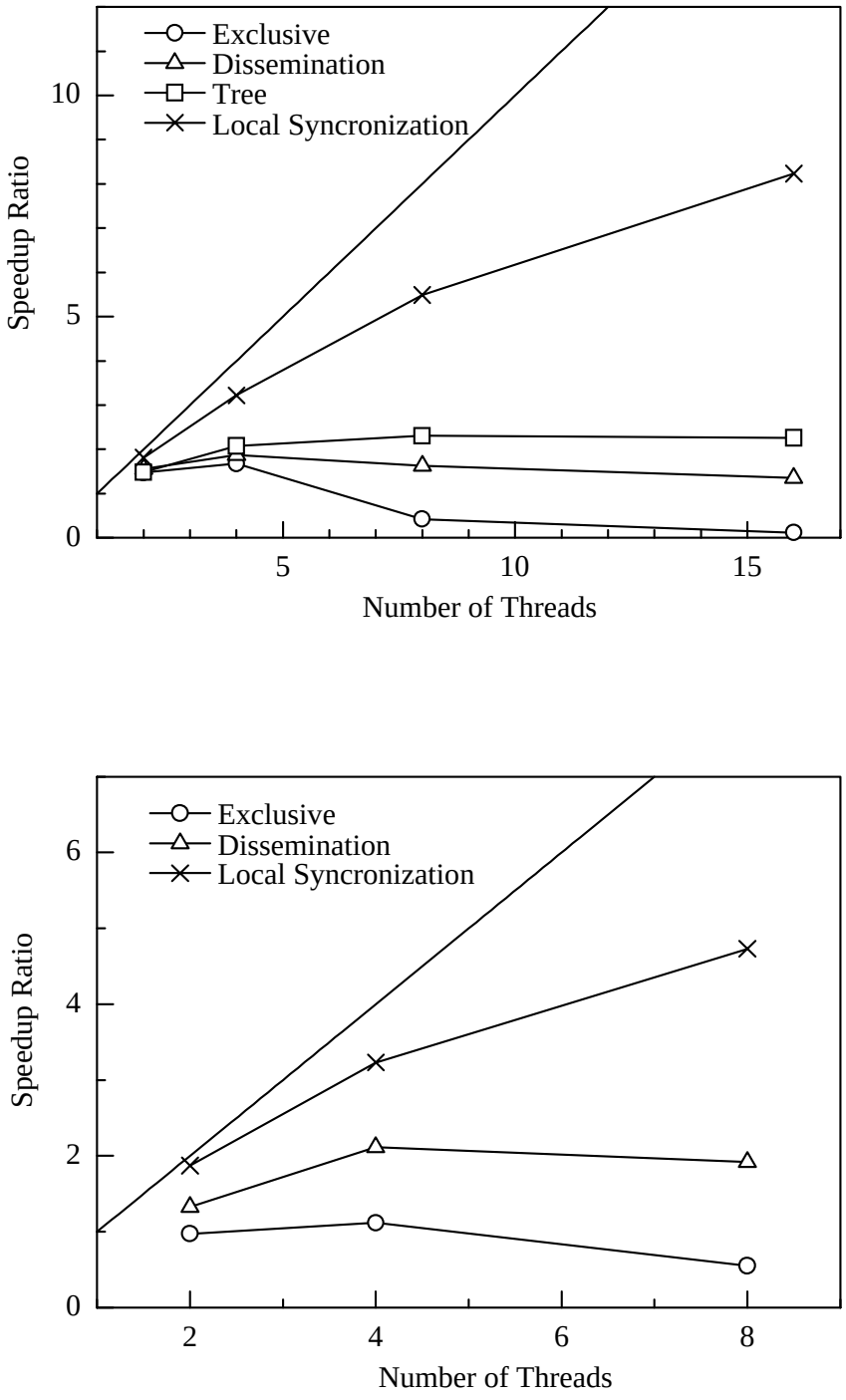

図 7.8: Speedup Ratio (6000step) upper:Starfire bottom:J90

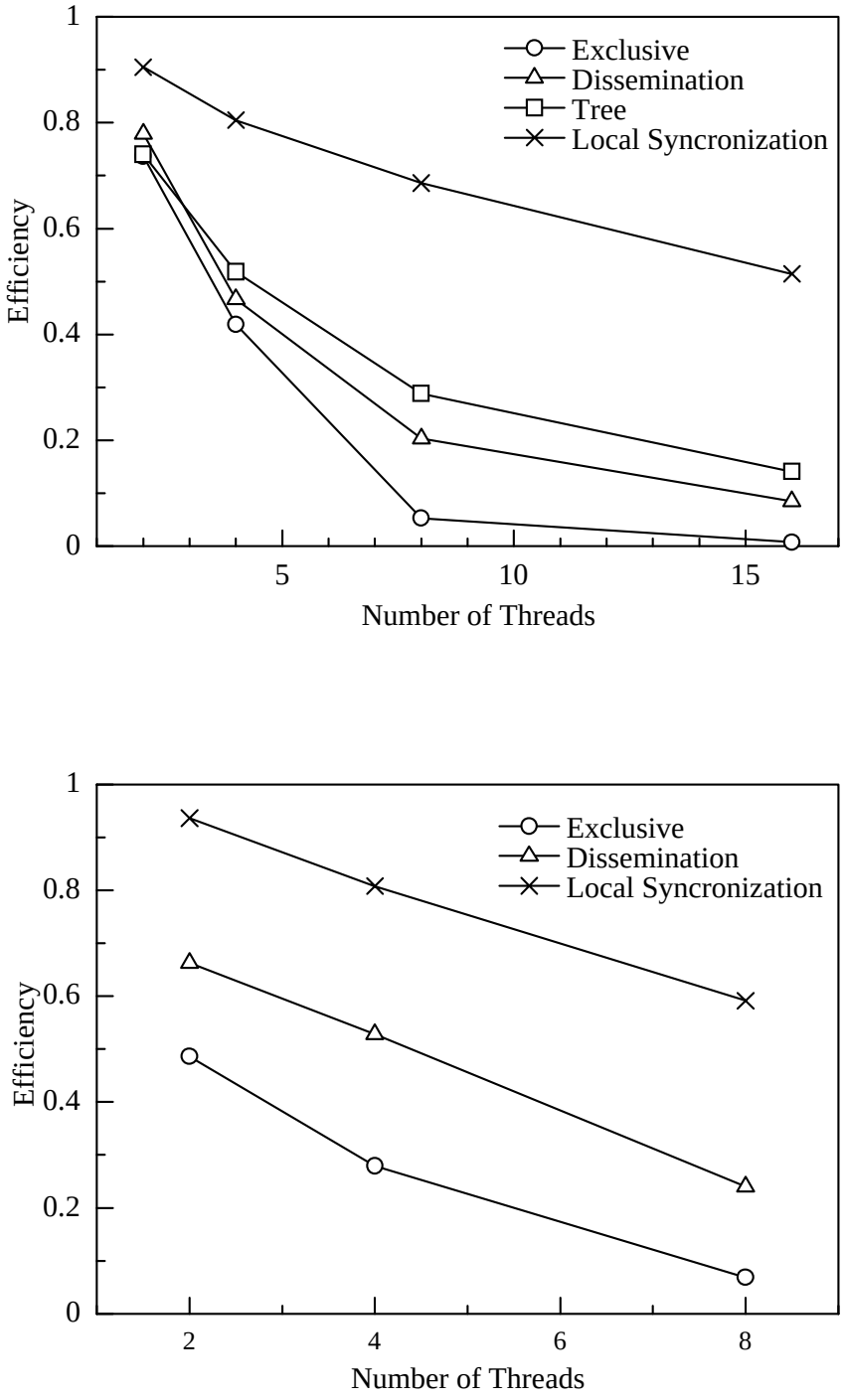

図 7.9: Efficiency (6000step) upper:Starfire bottom:J90

第 8 章

考　察

各同期方法を用いた SOR 法のパイプライン処理を実際の流体解析問題に適用し、その効果を調べた。SOR 法を従来のような領域分割法で解かず、共有メモリシステムでパイプライン処理で解くことによって、並列化効率のみならず、従来の領域分割法による方法よりも収束性が向上している。

8.1　Starfire

実験の結果から高い並列化効率が得られると思われた。しかし、各バリア同期を用いた場合では、実験の結果に反して、あまり良くない結果が得られた。局所同期に関しては、並列度が低い場合には実験よりも少し悪いが、並列度が高くなると実験の結果よりも良い値になっている。

実験では、反復に収束条件などは与えず、回数で打ち切っていた。しかし、実際問題では反復回数は収束値によって、変動するそのため収束を監視するコストが掛かるが、バリア同期では監視している間、全てのスレッドの実行が停止してしまう。このため、パフォーマンスに大きく影響しているものと考えられる。局所同期では、監視しているスレッドが止まってもその他のスレッドは動作可能な部分まで実行を進めている。そのため、局所同期では、同期の影響が少いスレッド、つまり両端のスレッドに監視させることで、監視の遅れをある程度取り返すことが可能となる。

そして、並列度が低い場合に実験結果より効率が低く、並列度が高くなると、実験結果よりも効率が良くなる原因については、実験では反復が 1000 までであるが、実際には 500step までの場合 10000 ～ 数百回の反復が行なわれている。また、6000step までの場

合 10000 ～ 数回の反復が行なわれているため、反復が多い場合スレッド数が多い方が計算時間が早くなると考えられる。

そして、局所同期を用いた場合高並列時の計算領域の減少による効率の落ちは、2 スレッド使用時に並列化効率は、500step の場合 90% 、16 スレッド使用時では、58% 程度になる。6000step の場合では、それぞれ 90% 、55 % になる。500step と較べた場合、6000step 時の 16 スレッドを使用した場合に効率の減少が大きいのは流体現象が定常に近付くに連れて、並列の効果に支配的な圧力解法部分の反復が非常に少くなるために 500step と較べた場合高並列時の効率が減少している。

8.2 J90

傾向としては Starfire と同様である。実験では、バリア同期の効率が高かったが、実際問題に適用した場合、局所同期の方が高いパフォーマンスが得られている。この原因も前述のように圧力反復の収束判定部分である。

局所同期を用いた場合の高並列時の計算領域の減少による効率の落ちを見てみると、2 スレッド使用時に並列化効率は、500step の場合 97%、8 スレッド使用時では、67%程度になる。6000step の場合では、それぞれ 93%、57%になる。Starfire の場合と較べて、ほぼ同程度の並列化効率の減少で済んでいることが分かる。

8.3 同期機構の効果

高速なバリア同期を用いた場合、並列度が少い時にはバリア同期を用いても十分効果があることが分かる。そのため、PC や WS ベースの小規模な SMP システムでは、バリア同期を用いても十分なパフォーマンスが得られると考えられる。また、Starfire と J90 の結果を較べる限り、どちらのシステムで、どの同期方法を用いても、大きな傾向の変化はない。そして、局所同期はこの 2 つのシステムで安定した高効率な同期手法であり、多分に汎用性があると考えられる。そして、その他の解法を並列化する場合にも、高速なバリア同期機構と局所同期を適所に使用することで十分なハイパフォーマンスな数値計算が可能である。

また、計算粒度が小さくなった場合 Starfire および J90 においても、大きく並列化効率は、下がっている。しかし、MPI を用いた熱流体解析ベンチマークでの 130 × 130 の場合の並列度が同じものと比較した場合、熱流体ベンチマークでは、並列化効率は 2%程度

まで落ち込むが、局所同期を用いた場合、6000step 時でも 40%程度で済んでいる。よって、局所同期を用いた場合、領域分割法による MPI を用いた方法に対して非常に効率が良いといえる。

以上のように圧力解を得るための反復が多ければ多いほど、並列化効率は落ちてこないことがわかる。そのため、本稿では扱った流体問題は定常流れの問題であるため、流体が定常状態に近付くに連れて、圧力の反復は少なくなる。しかし、流体問題の大半を占める流れ場は、非定常問題である。そのため、時間ステップを追うごとに、圧力の反復がすくなくなるということはない。そのため、非定常問題に適用した場合にも効果的であると考えられる。

実際問題にこれらの並列手法を用いた場合に最も重要なのが、収束判定部分であることが分かった。そこで、今後の課題として収束判定部分の改良案を示す。

バリア同期や局所同期の効率低下を防ぐための手段として、1 スレッドを収束判定用に用意しておくことである。収束判定のためだけに、1 スレッド用いると計算機資源の利用効率という意味では、効率の悪い方法であるが、計算効率をとった場合、非常に有効であると考えられる。そして、実装が比較的用意に行なえるが、8 PE のシステムに 9 スレッド用いることによるパフォーマンスの低下がどのようになるかが、この方法が成功するかの鍵となる。

第 9 章

ま と め

9.1 SMPシステムの実効性能

本稿では、大規模 SMP システムである Starfire の実効性能を各種ベンチマークを行なうことにより明らかにした。そのなかでも、自動並列化コンパイラの性能および SMP システム上での MPI の性能について詳細に調べた。自動並列化コンパイラは、ループ内部の変数に依存関係がない場合には、良い結果が得られるが、並列化できない場合の方が、実用問題には多分に含まれる。また、SMP システムがキャッシュの再参照性に影響されることが明らかになった。MPI で並列化した大規模計算を見た場合、そのパフォーマンスはかなり良いものである。

実問題のプログラミングのコストを考えた場合、ユーザー自身が並列化を行なう部分と自動並列化コンパイラに任せる部分を明確にし、その使いわけを行なえば、プログラミングコストはかなり削減できるものと考えられる。

9.2 各種同期機構の実装と実効性能

スレッドライブラリやアルゴリズムを用いて、各種同期機構を実装した。バリア同期は、排他処理を用いたプログラミングを行なった場合、同期遅延時間が掛かり、使用するスレッドが増えるに連れて、スレッド数の 2 乗で影響する。それを回避する色々なアルゴリズムが考案されている。高速なバリア同期を考えた場合、それは使用するシステムによって、バリア同期の方法が異なる。しかし、実際問題に適用した場合、大域的な同期手法が影響して並列のパフォーマンスが減少する場合が見られる。

そして、バリア同期という大局的な同期では、プログラミングの自由度も少くなり、実効パフォーマンスの低下が見られるため、データアクセスに対して同期を取る局所同期を実装した。この同期方法では、実際の流体問題に適用した場合でもパフォーマンスの減少はほとんど見られない。そして、計算粒度が小さくなる高並列部分においても、効率のよい並列化が行なえることが分かった。

9.3 今後の課題

現時点では、SMP システムの効率的な利用方法を理解し、その理解に基づき基本的な流体問題の高速解法の方法を、二種類のシステムで実装し、 システムによる違いを調べた。そして、両システムにおいて良い結果が得られた。その上で今後、更に複雑な流体解析問題に対してこの方法を適用し、実際問題を高速に解析する手段として活用できるようにする必要がある。

そのために、必要なことは、

- 圧力の Poisson 方程式を解く段階で得られる、大規模 n 元連立方程式の係数行列に前処理を施し、最も計算時間を要する解析を高速化する。
- 上述の前処理の並列化も同時に行ない、更に高速計算を行なう。
- 一般問題への適用とそれに伴う三次元の大規模問題への適用。

以上のことが上げられる。

謝辞

まず最初に、本研究を進めるに当たり、御指導、御鞭撻を賜わりました、松澤照男 教授に深く感謝致します。研究中で生じた問題に対して、適切な助言、御指摘を頂きました松澤研究室の諸兄、特にお世話になりました博士後期過程の古山 彰一氏には感謝の意を表します。また、研究会において様々な御助言を賜わりました電総研の建部修見先生に感謝の意を表したいと思います。

最後に、研究生活を支えて下さいました両親、家族に感謝の意を表したいと思います。

参考文献

[1] 黒川原佳, 松澤照男: 自動並列化コンパイラを用いた SMP 型計算機の評価, 第 9 回計算流体シンポジウム講演会論文, pp.415-426, 1998

[2] 黒川原佳, 松澤照男: SMP 型計算機による非圧縮粘性流れの並列計算 第 12 回数値流体力学シンポジウム講演会論文, pp.533-534,1998

[3] 建部修見, 関口智嗣: 共有メモリ計算機における局所同期機構, 情報処理学会研究報告 98-HPC-72, pp.97-102(1998)

[4] Charlesworth, A., Aneshansley, N., Haakmeester, M., Drogichen, D., Gilbert, G., Williams, R., Phelps, A., : The Starfire SMP Interconnect, SC97:HIGH PERFORMANCE NETWORKING & COMPUTING(1997)

[5] 河村洋, 高橋達矢, 松澤照男, 黒川原佳, 荒川忠一, 功刀彰, 木村俊哉: 熱流体解析における並列計算のベンチマーク, 第 11 回計算工学講演会論文集, pp.583-584 ,1998

[6] Mellor-Crummey, J.M. and Scott, M.L. : Algorithms for Scalable Synchronization on Shared-Memory Multiprosessors, *ACM Trans. on Computer Systems*, Vol.9, No.1, pp.21-65(1991)

[7] Yew,P.C., Tzeng, N.F., Lawrie, D.H. : Distributing hot-spot addressing in large-scale multiprocessors, *IEEE Trans. Comput. C-36*, 4, pp.388-395(1987)

[8] Hensgen, D., Finkel, R. and Manber, U. : Two algorithms for barrier synchronization, *Int. J. Parallel Program*, Vol.17, No.1, pp.1-17(1988)

[9] Ghia,U., Ghia, K.N., Shin, C.T., : High-Re Solutions for Incompressible Flow Using thr Navier-Stokes Equations and a Multigrid Method, *J. Comp. Phys.* 48, 387-411, (1982)

[10] Han, Y., Finke, R., : An optimal schem for dissemination information, Proceedings of th 1988 Int. conf. on Parallel Processing II pp.198-203, 1988

[11] Kleiman, S., Shah, D., Smaalders, B., : Programming with THREADS, Prentice Hall press, 1995

[12] Message Passing Interface Forum, : A Message-Passing Interface Standard, URL http://www.mpi-forum.org/, Jun12, 1995

[13] 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 -WS, スーパーコン, 並列計算機, 丸善, 1991