卒 業 論 文

PC クラスタ上での OpenMP 並列プログラミング（Ｉ）

氏　　名　：　平野 茂樹
学籍番号　：　2210000197-0
指導教員　：　山崎 勝弘 教授

提 出 日　：　2004 年 2 月 19 日

立命館大学　理工学部　情報学科

内容梗概

本論文では、本研究室で構築した SCore 型 PC クラスタ上での、OpenMP による並列プログラミングについて述べる。

少し前まで並列処理は敷居が非常に高く、限られた人しか用いることができなかった。しかし現在、高性能な PC が安価で手に入るようになり、Myrinet などの高速なネットワーク環境が普及してきたことから高性能な PC クラスタの構築が可能になった。これにより、今まで並列処理に縁の無かった一般の人もこの技術を使ってプログラムを実行することができるようになった。

また共有メモリ計算機が広範囲に普及し、並列プログラミングも分散メモリ環境から、共有メモリ環境へと移行しつつある。その共有メモリ用のプログラミング言語として現在注目を集めているのが OpenMP である。OpenMP は移植性が高く、段階的に並列化できるなどメリットが多いので、今後並列プログラミングの主流になると期待されている。

OpenMP によって、「N クイーン」、「スプライン曲線」、「スプライン補間」、「ガウス・ジョルダン法」、「ガウス・ザイデル法」の計５つのプログラムで並列化を行なった。その結果、プロセッサファームアルゴリズムで並列化できる「N クイーン」、「スプライン補間」、「ガウス・ジョルダン法」については、ほぼ理想的な速度向上が得られた。「スプライン曲線」と「ガウス・ザイデル法」については、SCASH によるメモリ使用量制限や同期のオーバーヘッドなどにより、並列化すると逆に逐次の時よりも遅くなってしまった。

今後の課題点として、１つのプログラム上で OpenMP と MPI を組み合わせてお互いの長所を生かした並列化を行なう事が挙げられる。

目次

図目次

表目次

１．はじめに

計算速度は、より速くすることが常に求められてきた。高速計算が必要とされる分野（例えば、気象予測、流体計算、画像処理、データベース処理など）では、しばしば有効な結果を得るために大量のデータに対して多数回の繰返し計算が必要なことが多い。計算は「適切な」時間内に完了しなければならない。設計者が効率的に作業するための時間は短いので、解に到達するのに２週間もかかるシミュレーションは、通常受け入れられない。システムが複雑になればなるほど、それをシミュレートする時間はますます増加するし、計算に特定の締切り時間が存在することもある。

計算速度を向上させる１つの方法は、「問題を解くのに複数のプロセッサを使う」というものである。すなわち、プログラム全体をいくつかの部分に分割して、それぞれを別のプロセッサで実行する。このような計算プログラムを書くことを並列プログラミングと呼び、そのための計算プラットフォームである並列コンピュータは、複数のプロセッサあるいは複数の独立したコンピュータを何らかの方法で結合して設計する。この手法はずっと昔から考えられてきた技術であるが、1 台のコンピュータを作るのにも大変な時代に多数のプロセッサをつなぐことはなかなか実現できなかった。

しかし集積回路技術の発達により、今日では多数のプロセッサを接続することは困難ではない。1980 年代後半からは並列計算機の製品化が活発になり、現在はスーパーコンピュータや大規模サーバーなどのほとんどが並列型になっている。[4]

また最近では汎用マイクロプロセッサの高性能化が凄まじく、一昔前までは手の出なかった高性能スーパーコンピュータに匹敵するマイクロプロセッサが、ワークステーションやパソコンに使用されている。さらにネットワークにおいても近年著しい発展があり、ギガビットイーサ、Myrinet などのギガビット級のネットワークが妥当な価格で手に入るようになってきた。ハードウェアばかりでなく、通信におけるソフトウェアオーバーヘッドを低減化した低遅延の通信方式も数々開発されてきている。その結果、パソコン、ワークステーションなどの汎用計算機と汎用ネットワークを用いた計算機クラスタが、コストだけでなく性能の点でも大きな可能性を持つようになってきた。

クラスタシステムの中で最も注目されているのが、新情報処理開発機構によるリアルワールドコンピューティング（RWC）プロジェクトで開発された、クラスタ用システムソフトウェアである SCore を利用したクラスタシステムである。SCore は、Linux 上に構築したトータルソフトウェアであり、高性能通信機能を実現する通信ライブラリを持つ。さらに Linux カーネル上に構築した SCore-D グローバルオペレーティングシステムは、クラスタの構成要素であるコンピュータを全て制御し、クラスタをあたかも単一の並列コンピュータのように見せる。さらにソフトウェア分散共有メモリシステム（SCASH）により、分散メモリのクラスタ上で、共有メモリプログラミングができる。これによって、SCore クラスタシステム上で、共有メモリ型並列言語である OpenMP のプログラミングが可能になっている。[19][28][29]

本研究では、OpenMP によるプログラミングを PC１６台の SCore 型クラスタ上で行なった。並列化した問題は、「N クイーン」、「スプライン曲線」、「スプライン補間」、「ガウス・ジョルダン法」、「ガウス・ザイデル法」の計５つである。

「Nクイーン」は、N 行 N 列のチェス盤に N 個の駒を配置するというパズル問題である。スレッドに分割する対象を増やし、静的サイクリック分割でスレッド間の計算量を均等にすることにより、１６台で約１６倍という理想的な速度向上が得られた。

「スプライン曲線」は、画像処理などで使われる、なめらかな曲線を求めるものである。ループ・アンローリングという手法を用いて依存関係をなくし、独立にブロック分割で計算したが、SCASH のメモリ量制限などにより、逐次の時より遅くなってしまった。

「スプライン補間」は、スプライン曲線を計算後、補間点に対応する関数値を求めるものである。並列計算中に同期が必要なく独立に計算できるため、１６台で約１６倍という理想的な速度向上が得られた。

「ガウス・ジョルダン法」は、連立方程式を解く解法の１つであり、直接法に分類される。ピボットが存在する行全体をピボットで割った後、その他の行をブロック分割で並列化した。すると、逐次の割合が多く残ってしまうが、それでも１６台で約９倍の速度向上が得られた。

「ガウス・ザイデル法」は、連立方程式を解く解法の１つであり、収束法に分類される。並列化は、２つの方法を用いて検討した。１つは、独立に計算できる手法だが、非常に逐次性が高いため、逐次の時よりも遅くなってしまった。もう一つの方法は、パイプライン処理なのだが、「OpenMP の同期は全スレッドが対象」という制限のためオーバーヘッドがかかり、同様に並列効果が全く得られなかった。

２章では、ＰＣクラスタ上での並列プログラミングについての説明を示す。３章は「N クイーン」、４章は「スプライン曲線」、５章は「スプライン補間」、６章は「ガウス・ジョルダン法」、７章は「ガウス・ザイデル法」についての並列化を述べている。３章から７章のそれぞれで、問題定義、並列化アルゴリズム、実行結果、考察などを記述した。全体のまとめは、８章に記した。

２．ＰＣクラスタ上での並列プログラミング

２．１ 並列計算機のメモリモデル

並列処理を行なう計算機のアーキテクチャは、大きく分けて「共有メモリモデル」「分散メモリモデル」「分散共有メモリモデル」の３つに分かれる。[16]

（１）共有メモリモデル

複数のプロセッサがメモリバス／スイッチ経由で主記憶に接続される形態である（図１参照）。この形態は、

・メモリモデルが最も汎用でプログラムが組みやすい

・逐次処理だけでなくスループットを重視するサーバマシン（逐次プログラム／プロセスを多数処理するマシン）としても適している

という長所がある。そのため、近年ますます市場が増えてきている。

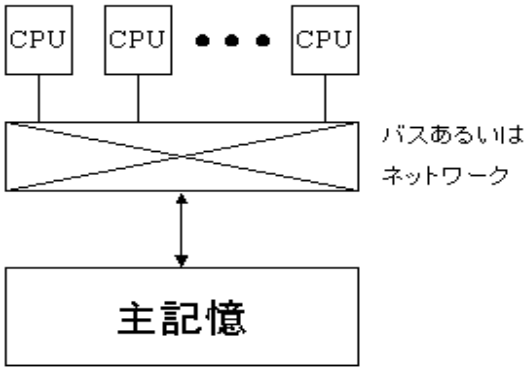

図 1：共有メモリモデル

課題点としては

・1 本のバスで複数の CPU からのメモリアクセスを処理しなければならないために接続できる CPU 数には上限が存在

・キャッシュの一貫性をいかに実現するか

が挙げられる。

（２）分散メモリモデル

プロセッサと主記憶から構成されるシステムが複数個互いに接続された形態で、プロセッサは他のプロセッサの主記憶の読み書きを行なうことができず、必ず相手のプロセッサに介在し

てもらう必要がある（図２参照)。
この形態は、
・大規模なシステム構築が可能
・デットロックさえ気を付ければタイミング依存による嫌なバグが発生することは少ない
という長所がある。通信遅延のオーバヘッドは極めて高くなるが、PC や WS を LAN で接続したものもこの形態に属するといえる。近年、安価なスイッチングハブが入手できるようになり、分散メモリモデルでの並列プログラミングも活発になりつつある。

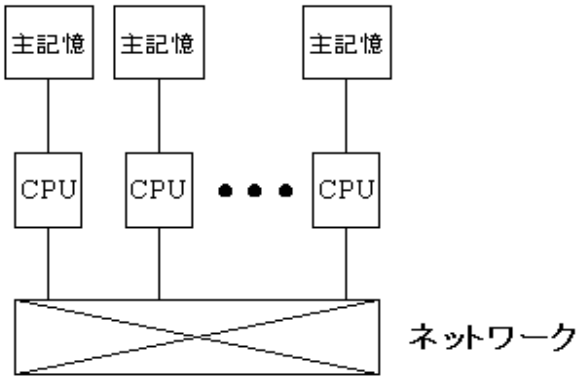

図 2：分散メモリモデル

課題としては
・通信をプログラミング時に全てスケジューリングする事は、プログラマには多大な負担
が挙げられる。

（３）分散共有メモリモデル
　プロセッサと主記憶から構成されるシステムが複数個互いに接続された形態で、プロセッサは他のプロセッサの主記憶の読み書きを行なうことができる（図３参照)。
この形態は
・大規模なシステム構築が可能
・分散メモリに比べて自由度が高い
という長所がある。

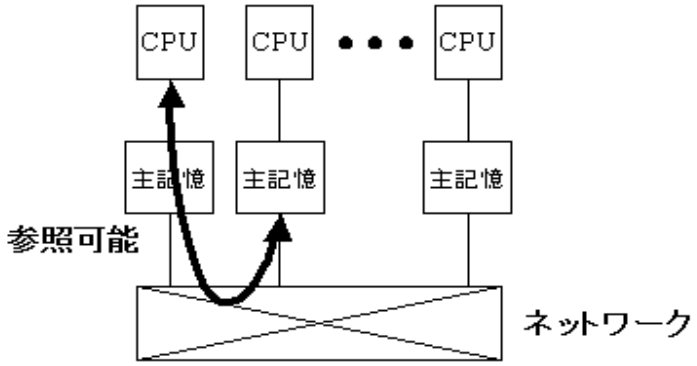

図 3：分散共有メモリモデル

課題としては
・1 個でも同期の場所を間違えると、タイミング依存で非常に解析しにくいバグを作る
が挙げられる。

２．２ ＰＣクラスタ

PC クラスタとは、PC 単体では性能的にスーパーコンピュータには及ばないものの、複数の PC をネットワークで接続し、仮想的に 1 台の並列コンピュータとして利用することで、パフォーマンスを向上させた PC 群を意味する。本研究室では、ＰＣクラスタを用いて並列処理を行なっているが、これは２．１章で記述した、分散メモリモデルに属する。２～８台の小規模な PC クラスタや数千台規模でスーパーコンピュータの性能を発揮する PC クラスタも構成することが可能である。また近年の PC の高性能化、低価格化に伴うコストパフォーマンスの向上により、PC クラスタの構成も費用対効果が良くなった。ネットワークにおいても 100M Ethernet から Gigabit Ethernet や Myrinet などの高速ネットワークの登場で、PC の高性能化に対するバランスが取れるようになり、大規模な PC クラスタで生じるネットワークのオーバーヘッドを最小限に抑えることが可能となった。[26][27]

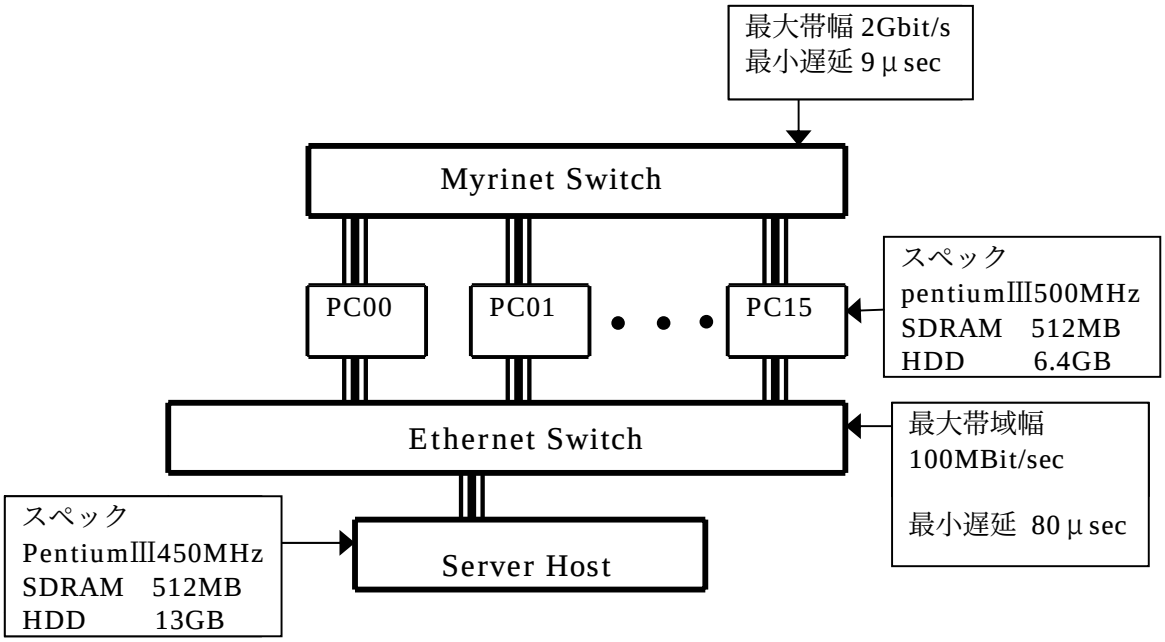

図 4：本研究室のＰＣクラスタの構成

本研究室の PC クラスタは、計算ノードとなる PC クラスタ 16 台とそれらを管理するサーバー1 台で構成されている。

図４に、本研究室の PC クラスタの構成を示す。PC クラスタ１６台を結合するネットワークは、Myrinet2000 である。サーバーと１６台のクラスタの接続は、Ethernet（100Base-TX）である。

各 PC は、OS として Redhat Linux 7.3 がインストールされ、PC クラスタシステムソフトウェアに Score5.6.1 がインストールされている。

２．３ 並列プログラミング

現在、いくつもの並列言語が提案されている。しかし、２．１章で述べた並列計算機の３つのアーキテクチャによって、使用できる並列プログラミング言語が異なる（表１参照)。アーキテクチャに応じて、言語を選択しないといけない。

表 １：並列プログラミング言語の分類

共有メモリモデル：pthread、OpenMP (分散共有メモリモデル)
分散メモリモデル：PVM、MPI、HPF

本研究室の並列計算機はＰＣクラスタであり、これは分散メモリモデルに属する。しかし、Score のソフトウェア分散共有メモリシステム（SCASH）により、分散メモリのクラスタ上で、共有メモリプログラミングが可能となる。

以下に、本実験で用いた共有メモリモデルの並列プログラミング言語である OpenMP について述べる。

OpenMP が開発された背景の根底としては、アーキテクチャによって使用できる並列プログラミング言語が異なるということが挙げられる。これが、一般的に並列処理が広く浸透していく上での大きな問題となっている。しかし、共有メモリ・分散メモリというアーキテクチャ上の差異を完全に無視できるような言語の開発は難しかったため、現在の並列計算用言語では共有メモリ用（分散共有メモリも含む）の言語、分散メモリ用の言語、の２種類に分類され、OpenMP は共有メモリ用の言語に属することになる。

それでは、なぜ新たに OpenMP を開発したのだろうか。理由として、近年のデスクトップからスーパーコンピュータまで、共有メモリ型並列計算機の広範囲な普及に伴い、同計算機における並列計算用言語に共通化の必要性が出てきた事が挙げられる。すなわち、アーキテクチャで言語が異なるのは仕方が無いとして、せめて同一のアーキテクチャ上においては共通の並列プログラミング言語を、ということである。

OpenMP では次のような特徴がある。

・既存の逐次プログラムをベースに、並列プログラムを作ることができる。
・指示文を使って、スレッドを生成・制御することができ、スレッドライブラリなどを使うよりも簡単にスレッドプログラミングができる。
・徐々に指示文を加えることにより、段階的に並列化できる。
・OpenMP の指示文を無視することにより、元の逐次プログラムになる。従って、逐次と並列プログラムを同じソースで管理することができる。

OpenMP の構文は、parallel 構文、ワークシェアリング構文、同期のための構文、の三つに大きく分けられる。以下に、それぞれの基本的な構文を説明する。[12][13][18][20]

（１）parallel 構文

parallel 指示文は並列リージョンを定義する。並列リージョンとは、複数のスレッドによって並列実行される部分である。この指示文は、並列実行を開始する基本的な構文である。終了時には、暗黙のバリア同期が取られる。

またいくつかの指示文では、ユーザが指示節を用いて、そのリージョンの実行中に変数のスコープ属性を制御することができる。以下は主な指示節である。

・shared・・・構文内で指定された変数がスレッド間で共有される。
・private・・・構文内で指定された変数をスレッドごとに持たせる。
・reduction・・・指定した変数に対し、それぞれのスレッドが保有している部分的な値を、指定された演算子によって同じ演算を行い、一つの結果としてまとめる。

（2）ワークシェアリング構文

ワークシェアリング構文は、ループなどを分割してスレッドに割り当て、それぞれのスレッドで計算をする。OpenMP では、以下のようなワークシェアリング構文を定義している。いずれも終了時には、暗黙のバリア同期が取られる。

①for 構文

直後の for ループの実行範囲を並列実行するものである。for ループは canonical（正規形）でなくてはならず、ループ制御変数は整数型で強制的に private 属性になる。for 構文は、何も指定しなかったらブロック分割になるが、スケジュール構文を使うことでサイクリック分割を実行できる。for 構文の指示節で、schedule の指定を行なう。

schedule の種類は以下のとおりである。

・static・・・schedule(static,chunk_size)が指定された場合、ループは chunk_size で指定されたサイズのかたまりに分割される。生成された各チャンクは、チーム内のスレッドにスレッド番号順にラウンドロビン形式で割り当てられる。
・dynamic・・・schedule(dynamic,chunk_size)が指定された場合、ループは chunk_size で指定されたサイズのかたまりに分割される。スレッドに割り当てられたチャンクの演算が終了すると、残りのチャンクがなくなるまで動的に別のチャンクをスレッドに割り当てる。
・guided・・・shedule(guided,chunk_size)が指定された場合、最初大きなかたまりをスレッドに割り当て、処理が進むにつれて、かたまりを小さくしながら割り当てる。スレッドに割り当てられたチャンクの演算が終了すると、残りのチャンクがなくなるまで動的に別のチャンクをスレッドに割り当てる。
・runtime・・・schedule(runtime)が指定された場合、スケジューリングは実行時に決定する。

②section 構文

それぞれの section を並列に実行するというものである。つまり、下のようなプログラムなら section1 と section2 の内容が同時に実行されることになる。

```
#pragma omp sections
{
#pragma omp section
{ …section1…}
```

```
#pragma omp section
{…section2…}
}
```

③single 構文

チーム内の１つのスレッド（必ずしもマスタースレッドとは限らない）のみで実行されることを指示するものである。構文は以下の通りである。

```
#pragma omp single
{
…statement…
}
```

（３）同期構文

①master 構文

マスタースレッドのみで実行されることを指示するものである。マスター以外のスレッドは、master が指定されたステートメントを実行しない。マスターセクションの入り口と出口では、暗黙のバリア同期は取られない。

②critical 構文

排他的に実行される critical section を指定する指示文である。大域的な名前を付けることができ、同じ名前の critical section は排他的に実行される。

③barrier 指示文

バリア同期を取る指示文である。全てのスレッドが同期点に達するまで待つ。parallel 構文と work sharing 構文においては、nowait 指示節が指定されない限り、終了時に暗黙のバリア同期が取られる。

④atomic 構文

複数の同時書き込みを行なう可能性のあるスレッドに対して、指定されたメモリの更新を逐次で行なうものである。

⑤flush 構文

メモリ上にあるオブジェクトに対して、矛盾しないメモリ参照を指示するものである。つまり、指定した変数について、メモリの一貫性を保証する。

２．４ 並列アルゴリズム

一般的に並列アルゴリズムは、プロセッサファーム（Processor Farms）、分割統治法（Divide and Conquer）、プロセスネットワーク（Process Networks）、繰り返し変換（Iterative Transformation）の４つに分類できる（図５参照）。[21]

(a)プロセッサファーム

まずは全体の制御をマスターが行ない、与えられた問題を複数の独立した計算に分割し、各スレーブまたはマスターがそれを担当する。それぞれが与えられた計算を独立に行ない、その結果をマスターに返す。そして最終的にマスターがそれをまとめ、その問題の解を得る。

(b)分割統治法

まずは与えられた問題についてなんらかの計算をする。そして、ある条件をもとに計算を分割し、また、その計算を行なう。そしてまた分割･･･と、再帰的に計算と分割を繰り返しながら解いていく。その問題の解は、分割された部分解を回収し、まとめることにより得られる。

(c)プロセスネットワーク（パイプライン処理）

まずは与えられた問題について、その計算を複数のステージに分ける。入力データは、あるステージ上で計算が行われたら次のステージに移り、また、そこで計算が行われて次のステージに移る･･･と順番に流れていき、各ステージは並列に実行される。

(d)繰り返し変換

まずは与えられた問題をあるオブジェクトに分ける。各オブジェクトは複数の繰り返しの計算により値が変換され、求める値が得られる。その繰り返しは、オブジェクトの値がある与えられた条件を満たすまで続けられ、また、前のステップで計算された値は自分自身または別のプロセッサ上でランダムに利用される。

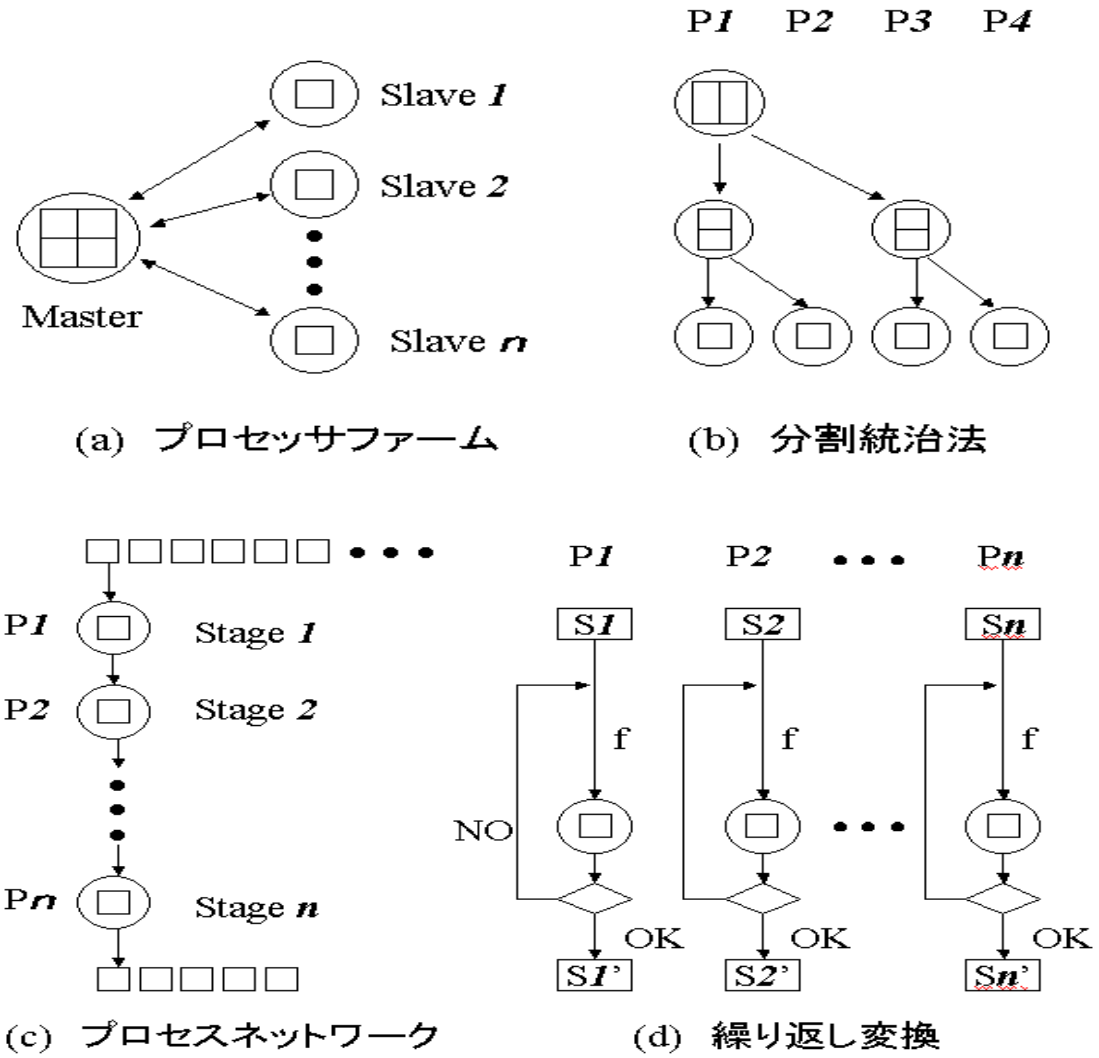

図 5：並列アルゴリズムの一般的分類

本論文では、これら４つの並列アルゴリズムのうち、「プロセッサファーム」、「プロセスネットワーク（パイプライン処理)」、「繰り返し変換」の３つの考え方を用いて実験をしている。

３．N クイーンの並列化

３．１ 問題定義

N クイーンとは、N 行 N 列のチェス盤に N 個の駒を配置するパズル問題である。しかし置き方に条件があり、同じ行・同じ列・同じ斜め方向に駒を 2 つ以上置くことはできない。
以下に例を示す。

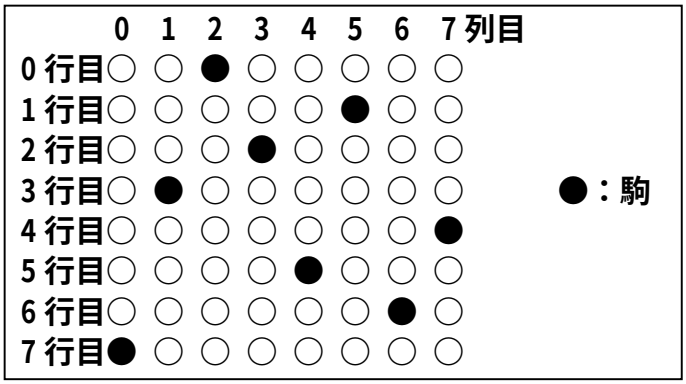

図 6：Nクイーン（N＝8）の解の一例

N=８では解が９２個あり、図６はそのうちの１つである。

解を見つけるのに、まず０行目のどこかに駒を置く。それから、それとぶつからないように１行目に駒を置き、それまでの２つとぶつからないように２行目に駒を置き・・・という具合に進めていく。途中の行で置く場所が無くなった時は、１つ前の行に戻って、その行のクイーンを別の列に置いて再び前に進む。１つ前の行のクイーンをどこに置いても駄目な場合には、さらに前の行に戻ってクイーンの置き場所を変更する。最後の行まで来ると、その置き方は解の１つとなる。つまり、深さ優先探索のアルゴリズムで答えを求めることになる。

ある位置に駒が置けるかどうかは、その位置の同じ行・同じ列・同じ斜め方向にすでに置かれた駒があるかどうかを調べればよい。

３．２ 並列化手法

（１）０行目で置く場所が違うと、その解は必ず違ったものになる。よって、０行目の位置を分割して並列計算ができる。例えばN＝８の時に、２台で並列処理をする場合、０行目の０～３列目に駒を置いた時の計算をスレッド０、４～７列目に駒を置いた時の計算をスレッド１に割り振る。

（２）　（１）のやり方だと、例えばN=１７の時に１６台で並列実行する場合、０行目の１７個の要素を１６台に振り分けることになる。計算が膨大なので、このやり方でも一応並列効果は出るが、１台に約１個だとスレッドの生成・消滅のオーバーヘッドが目立ってしまう。よって０行目だけでなく、０行目と１行目の両方での位置を分割させる。例えば N=８なら、０行目と１行目の両方の置き方は８×８＝６４通りある。ここで、（０行目の位置、１行目の位置）＝（０列目、０列目）なら「０」、（０行目の位置、１行目の位置）＝（０列目、１列目）なら「１」、・・・、（０行目の位置、１行目の位置）＝（７列目、７列目）なら「６３」という風に、０行目と１行目の置き場所に対し、０～６３までの数字を割り振る。そして２台で並列実行するなら、０～３１の数字がつけられた部分をスレッド０が計算、３２～６３の数字がつけられた部分をスレッド１が計算・・・として処理する（図７参照)。

（３）　（１）（２）ともに並列処理はブロック分割である。しかし、０行目と１行目のどこに駒を置くかによって、それぞれの解の個数（計算量）は異なる。スレッド間で計算量が違う場合、一番遅いスレッドが終わるまで他のスレッドは待つ必要があり、これにより処理時間が長くなってしまう。よって（２）のやり方の分割を、サイクリック分割で行なう。サイクリック分割とは、計算範囲を細かく分割してスレッドに割り当てることにより、スレッド間の計算量を均等にする手法である。実際には、「０、１行目 静的サイクリック分割」、「０、１行目 動的サイクリック分割」、「０、１行目 Guided サイクリック分割」の３つで実験した（それぞれ図８～１０を参照)。

以下に、N＝８の時、０行目と１行目の置き方により分割する手法の例を示す。
０行目と１行目の置き方に対して、「０」～「６３」までの数字を割り振る。

「０」～「３１」 (スレッド０が担当)	「３２」～「６３」 （スレッド１が担当）

図 7：ブロック分割

図７は、ブロック分割である。計算範囲をスレッド数で割り、その計算ブロックを各スレッドに割り当てて処理をする。

「0」 (スレッド0)	「1」 (スレッド1)	「2」 (スレッド0)	・・・ ・・・	「62」 (スレッド0)	「63」 (スレッド1)

図 8：静的サイクリック分割

図8は、静的サイクリック分割である。計算範囲を細かく分割して、それをスレッド0から順に割り当てる。

「0」 (スレッド0)	「1」 (スレッド1)	「2」 (スレッド1)	・・・ ・・・	「62」 (スレッド0)	「63」 (スレッド0)

図 9：動的サイクリック分割

図9は、動的サイクリック分割である。計算範囲を細かく分割してスレッドに割り振る点は、静的サイクリック分割と同じである。違いは、静的サイクリック分割がスレッド0とスレッド1にデータを順番に割り当てているのに対し、動的サイクリック分割は、処理が終了したスレッドにデータを割り当てる。よって、静的サイクリック分割と比べて効率が良いが、その分コストもかかる。

「0」～「9」 (スレッド0)	「10」～「19」 （スレッド1）	・・・ ・・・	「62」 （スレッド0）	「63」 (スレッド1)

図 10：Guided サイクリック分割

図10は、Guided サイクリック分割である。最初はおおきな範囲で分割し、計算を進めるにつれて分割数を小さくしていく。

３．３ 実行結果

表 ２：Nクイーン並列実行結果（N＝１７）、単位は秒

	１台	２台	４台	８台	１６台
０行目 ブロック分割	３６０５０	１９２０７	１１７８４	７１２１	４７７３
０、１行目 ブロック分割	３６０５０	１８０４０	１０１４１	５３０４	２８３４
０、１行目 静的 サイクリック分割	３６０５０	１８７９９	９４０６	４７０６	２３５５
０、１行目 動的 サイクリック分割	３６０５０	１８９７３	１２１４３	６４７０	３５８３
０、１行目 Guided サイクリック分割	３６０５０	２２４９４	１１２６５	６１１０	４０２５

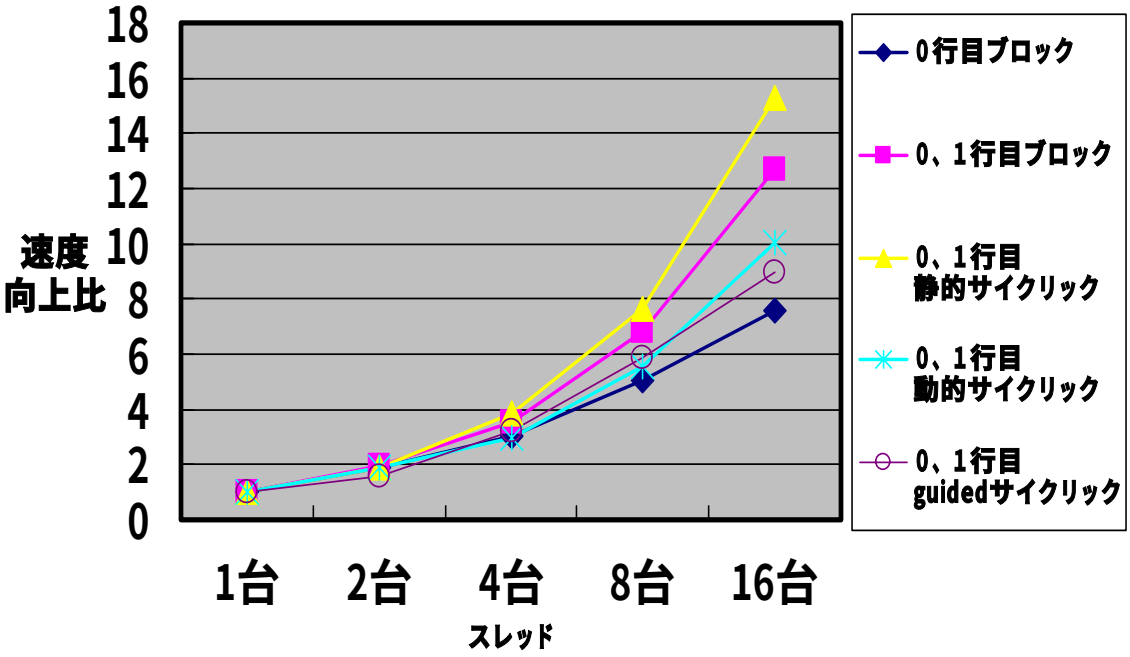

図 11：Nクイーン速度向上比（N＝１７）

（注）n 台で並列処理をしたときの速度向上比 S(n)は、次のように定義される。
S(n)=（逐次処理実行時間）÷（n 台での並列処理実行時間）
ちなみに、特別な理由がない限り、n 台での最大速度向上比は n である。

３．４ 考察

２台・４台の時は、５つの方法で速度向上の差はない。８台・１６台の時に、スレッド生成・消滅のオーバーヘッドやスレッド間の計算量不均衡が原因で、それぞれの方法に差が出ている。

１６台の時に着目すると、「０、１行目 静的サイクリック分割」が約１６倍と一番速く、理想的な速度向上が得られている。これは、スレッドへ分割する対象が多く、またスレッド間で計算量が均等になるようにしたためである。

ところで、「０、１行目 動的サイクリック分割」と「０、１行目 Guided サイクリック分割」は共に、スレッド間で計算量が同じなのにも関わらず、計算量不均衡である「０、１行目 ブロック分割」よりも遅くなった。「０、１行目 ブロック分割」の計算量（解の個数）が、どれくらいスレッドごとに違うのか調べてみると、最大でも４倍程度であった。動的に分割するためにはその分コストが必要で、「０、１行目 動的サイクリック分割」、「０、１行目 Guided サイクリック分割」の速度向上が思ったよりも出ていないのは、そのオーバーヘッドの影響だと考えられる。もし、スレッド間の計算量が１０倍くらい違っていたなら、「０、１行目 動的サイクリック分割」や「０、１行目 Guided サイクリック分割」の方が一番速かったかもしれない。

４．３次スプライン曲線の並列化

４．１ 問題定義

与えられた点（制御点）を全て通る、なるべく滑らかな曲線を求めたいときに使う手法がスプライン曲線である（図１２参照)。2 次元グラフィックスなどで曲線を表現する場合、その曲線を構成する全ての点をいちいち列挙していたのでは、計算や記憶のための手間がかかりすぎる。よって、制御点と制御点の間を通る曲線の多項式を求め、曲線全体はその制御点間をなめらかに接続させる（２次連続性を持たせる)。スプラインとは自在定規の意味である。

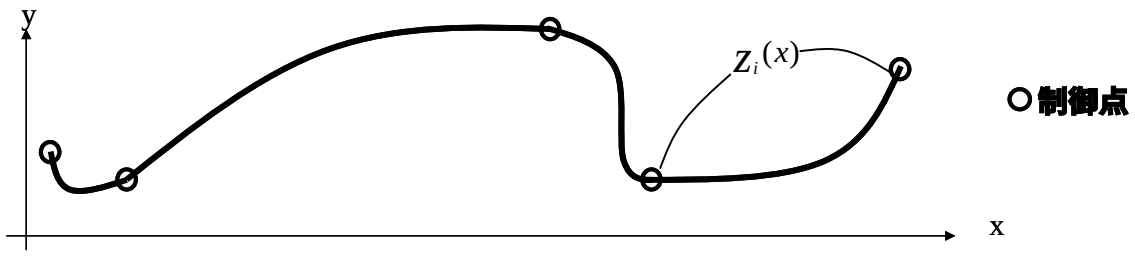

図 12：３次スプライン曲線

スプライン曲線では、与えられた区間[a,b]をいくつかの小区間に分けて、それらの小区間ごとに最大次数が m である多項式を用いる。f(x)がこのような多項式を各点で接続して構成されている関数であるとき、それを m 次のスプラインという。よく利用されているのは 3 次の多項式（m＝３）であるので、以下その場合について考察する。

仮に 3 次スプラインを f(x)とし、与えられた点（制御点）が $a=x_0<x_1<\cdots<x_n=b$ のように配置されているとする。このとき 3 次の多項式を

$$z_i(x)=p_i+q_i(x-x_i)+r_i(x-x_i)^2+s_i(x-x_i)^3 \quad (i=0,1,\ \cdots,\ n)$$

式(１)

と仮定する。$z_0(x),\ z_1(x),\ \cdots,\ z_{n-1}(x)$ は
それぞれ小区間 $[x_0,\ x_1],\ [x_1,\ x_2],\ \cdots,[x_{n-1},\ x_n]$ で定義されている。

また、スプライン曲線を求めるのに $z_n(x)$ を計算することは不要であるが、計算の都合により区間 $[x_n,\ \infty]$ で定義する。スプライン曲線を完成させるためには、それぞれの式（１）の多項式で、係数（p、q、r、s）を求める必要がある。

まず、式(１)を 2 回微分すると、

$$z_i''(x)=2r_i+6s_i(x-x_i)$$ 式（2）

を得る。ゆえに

$$z_i''(x_i)=2r_i$$ 式(3)

である。各点では２階の微係数が一致するから

$$z_i''(x_{i+1})=z_{i+1}''(x_{i+1})$$ 式(4)

である。この式の右辺は式（３）から

$$z_{i+1}''(x_{i+1})=2r_{i+1}$$

であるので、式(4)より

$$z_i''(x_{i+1})=2r_{i+1}$$ 式(5)

となる。 $z_i''(x)$ は１次関数であることと、式(3)と式(5)から、

$$z_i''(x)=2r_i+\frac{2(r_{i+1}-r_i)}{x_{i+1}-x_i}(x-x_i)$$

が得られる。この式から

$$s_i=\frac{r_{i+1}-r_i}{3h_i}$$ 式(6)

を得る。ただし $h_i=x_{i+1}-x_i$ とする。

ところで $z_i(x_i)=f(x_i)$, $z_i(x_i)=p_i$ であるから

$$p_i=f(x_i)$$ 式(7)

である。式(6)と式(7)から、式(1)は

$$z_i(x)=y_i+q_i(x-x_i)+r_i(x-x_i)^2+\frac{r_{i+1}-r_i}{3h_i}(x-x_i)^3$$

となる。ただし $f(x_i)=y_i$ とした。この式に $x=x_{i+1}$ を代入すると、

$$z_i(x_{i+1})=y_i+q_ih_i+r_ih_i^2+\frac{r_{i+1}-r_i}{3}h_i^2$$

となる。

さて、$z_i(x_{i+1})=z_{i+1}(x_{i+1})$ であるので、$z_i(x_{i+1})=y_{i+1}$ である。ゆえに

$$q_i=\frac{1}{h_i}(y_{i+1}-y_i-r_ih_i^2-\frac{r_{i+1}-r_i}{3}h_i^2) \qquad \text{式(8)}$$

である。式（6）と式（8）から、s_i と q_i は、r_i から求められる。

そこで r_i を１階の微係数が一致するという条件 $z_i'(x_{i+1})=z_{i+1}'(x_{i+1})$ から求めることにする。いま

$$z_i'(x)=q_i+2r_i(x-x_i)+\frac{r_{i+1}-r_i}{h_i}(x-x_i)^2$$

であるから、

$$z_i'(x_{i+1})=q_i+2r_ih_i+(r_{i+1}-r_i)h_i$$
$$=q_i+(r_{i+1}+r_i)h_i \qquad \text{式(9)}$$

である。また $z_i'(x_i)=q_i$ であるので

$$z_{i+1}'(x_{i+1})=q_{i+1}$$

である。そして $z_i'(x_{i+1})=z_{i+1}'(x_{i+1})$ から、式（9）は

$$q_{i+1}=q_i+(r_{i+1}+r_i)h_i$$

となる。この式に、式（8）を代入すると、

$$\frac{1}{h_{i+1}}(y_{i+2}-y_{i+1}-r_{i+1}h_{i+1}^2-\frac{r_{i+2}-r_{i+1}}{3}h_{i+1}^2)$$

$$=\frac{1}{h_i}(y_{i+1}-y_i-r_ih_i^2-\frac{r_{i+1}-r_i}{3}h_i^2)+(r_{i+1}+r_i)h_i$$

となる。これをまとめると

$$h_{i+1}r_{i+2}+2(h_i+h_{i+1})r_{i+1}+h_ir_i=\frac{3(y_{i+2}-y_{i+1})}{h_{i+1}}-\frac{3(y_{i+1}-y_i)}{h_i}$$

$$(i=0,1,\cdots,n-2) \qquad \text{式(１０)}$$

となる。

ところで、式（１）の 3 次多項式の未知数は n+1 個である。それに対して、式（１０）の方程式の個数は n-1 個なので、条件式が２つ不足する。よって不足する条件式を補い、３次スプライン関数を一意的に決めるために、人為的な端末条件を付与する必要がある。そこで、一般的に用いられるものとして「自然条件」というのを適用させる。これは、両端点における 2 次導関数を０とするもので、曲線の端点に荷重や曲げが加わらない場合に相当する。

以上の条件を使い、

$$z_i''(x_0)=r_0=0$$

$$z_i''(x_n)=r_n=0$$

となる。

さて、式（１０）の係数を

$$h_{i-1}=a_i \quad (i=2,\ \cdots,\ n-1)$$

$$2(h_i+h_{i-1})=b_i \quad (i=1,\ \cdots,\ n-1)$$

$$h_i=c_i \quad (i=1,\ \cdots,\ n-2)$$

$$\frac{3(y_{i+1}-y_i)}{h_i}-\frac{3(y_i-y_{i-1})}{h_{i-1}}=d_i \quad (i=1,\ \cdots,\ n-1)$$

とおけば、式（１０）は

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & \cdot & \cdot & \cdot & 0 & 0 & 0 \\ a_2 & b_2 & c_2 & 0 & \cdot & \cdot & \cdot & 0 & 0 & 0 \\ 0 & a_3 & b_3 & c_3 & \cdot & \cdot & \cdot & 0 & 0 & 0 \\ \cdot & \cdot & \cdot & \cdot & & & & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & & & & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & & & & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & 0 & \cdot & \cdot & \cdot & a_{n-2} & b_{n-2} & c_{n-2} \\ 0 & 0 & 0 & 0 & \cdot & \cdot & \cdot & 0 & a_{n-1} & b_{n-1} \end{bmatrix} \begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ \cdot \\ \cdot \\ \cdot \\ r_{n-2} \\ r_{n-1} \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \cdot \\ \cdot \\ \cdot \\ d_{n-2} \\ d_{n-1} \end{bmatrix}$$

となる。このような方程式を解くために、次のような新たに二つの変数 g_i と u_i を導入する。

$$\frac{c_1}{b_1} = g_1 \qquad \text{式（１１）}$$

$$\frac{d_1}{b_1} = u_1 \qquad \text{式（１２）}$$

$$\frac{c_i}{b_j - a_j g_{j-1}} = g_j \quad (j = 2, \cdots, n-2) \qquad \text{式（１３）}$$

$$\frac{d_j - a_j u_{j-1}}{b_j - a_j g_{j-1}} = u_j \quad (j = 2, \cdots, n-1) \qquad \text{式（１４）}$$

このとき、方程式は

$$\begin{bmatrix} 1 & g_1 & 0 & 0 & \cdot & \cdot & \cdot & 0 & 0 & 0 \\ 0 & 1 & g_2 & 0 & \cdot & \cdot & \cdot & 0 & 0 & 0 \\ 0 & 0 & 1 & g_3 & \cdot & \cdot & \cdot & 0 & 0 & 0 \\ \cdot & \cdot & \cdot & \cdot & & & & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & & & & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & & & & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & 0 & \cdot & \cdot & \cdot & 0 & 1 & g_{n-2} \\ 0 & 0 & 0 & 0 & \cdot & \cdot & \cdot & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ \cdot \\ \cdot \\ \cdot \\ r_{n-2} \\ r_{n-1} \end{bmatrix} = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \cdot \\ \cdot \\ \cdot \\ u_{n-2} \\ u_{n-1} \end{bmatrix}$$

となる。したがって、 r_i は、後退消去によって

$$r_{n-1} = u_{n-1} \qquad \text{式（１５）}$$

$$r_j = u_j - g_j r_{j+1} \quad (j = n-2, \cdots, 1) \qquad \text{式（１６）}$$

で与えられる。

４．２ 並列化手法

式（１３）を見ると、g[j]を求めるなら g[j-1]が必要になっている。つまり依存関係がある。他にも同じことが、式（１４）、式（１６）で該当する。依存関係があるため、並列化手法としてパイプライン処理を用いて実行することが考えられる。しかし、１つ前の値が知りたいだけなので、パイプライン処理で並列化しても、その処理内容は逐次処理と変わらない。それどころか同期が必要になるため、逐次処理よりも遅くなってしまうことが予想される。

そこで「ループ・アンローリング」という考え方を用いて、依存関係をなくす。仮に、依存関係のある式を a[i]=a[i-1]+b[i]として、これを式①とする。a[i]を求めるなら、a[i-1]が必要という事である（依存関係がある）。ここで、a[i-1]=a[i-2]+b[i-1]と変形し、式①に代入すると a[i]=a[i-2]+b[i-1]+b[i]となり、一つ飛びの漸化式に書き直すことができる。これにより、a[i]を求めるなら、a[i-2]が必要という事になり、偶数番目と奇数番目でそれぞれ独立に２台並列実行可能となる。さらにこれを繰り返すと、４つおきの漸化式、８つおきの漸化式、・・・、のようにどんどん並列性を上げることができる。

但し、並列性を上げれば上げるほど式が複雑になり、全体の計算量が増えてしまうという副作用がある。

４．３ 実行結果

表 ３：３次スプライン曲線の並列実行結果、単位は秒

	１台	２台	４台	８台	１６台
制御点６００万個	１０．３	１２．９	１６．２	１９．７	２２．２
制御点１００万個	１．７	２．４	４．１	６．８	１１．４

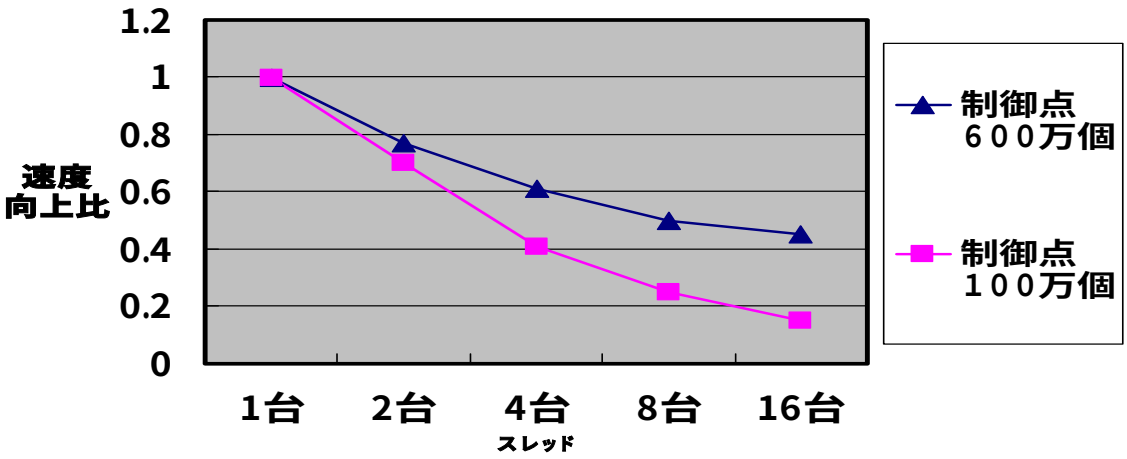

図 13：３次スプライン曲線の速度向上比

（注）SCore の SCASH によって、仮想的な共有メモリを実現しているため、ユーザーが使用可能なメモリ量には制限がある。現在の SCore5.6.1 においては、１６０MB が限界であることを確認している（１６０MB に近づけるほど処理は不安定になる）。つまり、各クラスタのメモリを５１２MB 積んで、それが１６台あったとしても、現状では１６０MB までのプログラムしか動かせない。

上記のスプライン曲線の結果において、制御点６００万個のプログラムで約１４０ＭＢである。これ以上データ数を増やすなら、より多くのメモリが必要となるため、今の研究室の環境では補間点６００万個が限度である。

４．４ 考察

逐次処理と比べて、並列化させた方が遅くなってしまった。理由を以下に述べる。

（１）制御点６００万個の時に、１台で約１０秒ほどしか出ていない。理想的な速度向上が出た３章の「Ｎクイーン」や５章で述べる「ガウス・ジョルダン法」にしても、１台の処理時間がこのように短いと、同様に並列化効果が出ない。データ数を増やせば処理時間が増えるだろうが、４．３の（注）の所にも書いたとおり、SCASH によりメモリ量が制限され、これ以上は実行できない。

この制約を打開するためには、以下の事柄が挙げられる。

①無駄な配列などを削除して、メモリ量を節約する。
②分散メモリ環境で、プログラムを実行する。

①については本プログラムにおいて、一時的にしか使わない配列を削除するなどを行なった。
これ以上メモリ量を減らすようにするなら、アルゴリズム自体を一から考え直す必要がある。
②については分散メモリモデルだと、Score はいらないので SCASH の制約もなく、各スレッド分のメモリを全て使える。本研究室は、５１２ＭＢのクラスタが１６台あるので、５１２ＭＢ×１６台＝８１９２ＭＢを使用できる（但し、それぞれのクラスタに OS などを入れているので、実質ユーザーが使えるメモリ量はこれよりも少ない）。

（２）ループ・アンローリングによって、台数を増やして実行すればするほど、式が複雑になり、全体の計算量が増えたことも影響している。
・制御点６００万個で、１台しか実行できないプログラム上で逐次実行すると、約１０．３秒
・制御点６００万個で、１６台並列実行可能なプログラム上で逐次実行すると、約３５．３秒

（３）いくつかの計算領域間でバリア同期を取っているが、それによるオーバーヘッドも多少関係している。

速度低下最大の原因は、上記（１）だと思われる。というのも、（２）と（３）については、「ガウス・ジョルダン法」などの他のプログラムと同様に、データ数が増えて処理時間が長くなればなるほど、影響は少なくなると考えているからである。
ちなみにプログラム全体から見ると、スプライン曲線並列化の部分が占める割合は非常に大きいため、仮にデータ数を増やすことが可能なら、速度向上は期待できるはずだと予想している。

５．３次スプライン補間の並列化

５．１ 問題定義

与えられた点（制御点）からスプライン曲線を求め、制御点以外の点ｘ（補間点）に対する関数値ｙを求めるものである。４章で述べたスプライン曲線は､曲線までしか求めていないが､今回は曲線を計算後、補間点に対応した関数値ｙまでを計算する。

この手法は、三角関数や対数関数などの関数の粗い数表しかなかった時代に、数表の間の値を近似する目的で考えられてきた。しかし現在では、コンピュータで任意の点における関数値が簡単に計算できるので、その本来の意味での補間法は必要ない。だが数式で表現不可能な、医学や社会モデルのような諸問題をコンピュータによってシミュレーション（模擬実験）する場合に、補間法は解析手法として有用である。

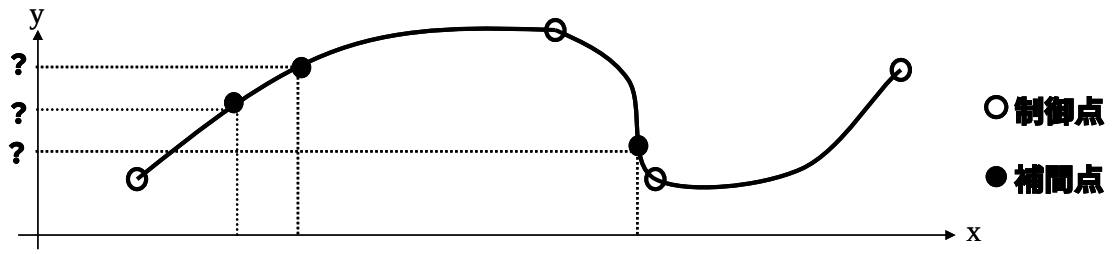

図 14：３次スプライン補間

５．２ 並列化手法

補間点をプロセッサファーム（ブロック分割）でスレッドに割り当て、並列実行することが考えられる。依存関係もなく独立に計算でき、バリア同期も必要ない。

５．３ 実行結果

表 ４：３次スプライン補間の並列実行結果（制御点は１０００個)、単位は秒

	１台	２台	４台	８台	１６台
補間点１００万個	８８０．０	４４３．０	２２１．５	１１０．７	５５．３
補間点１０万個	８７．１	４３．５	２１．８	１０．９	５．４７

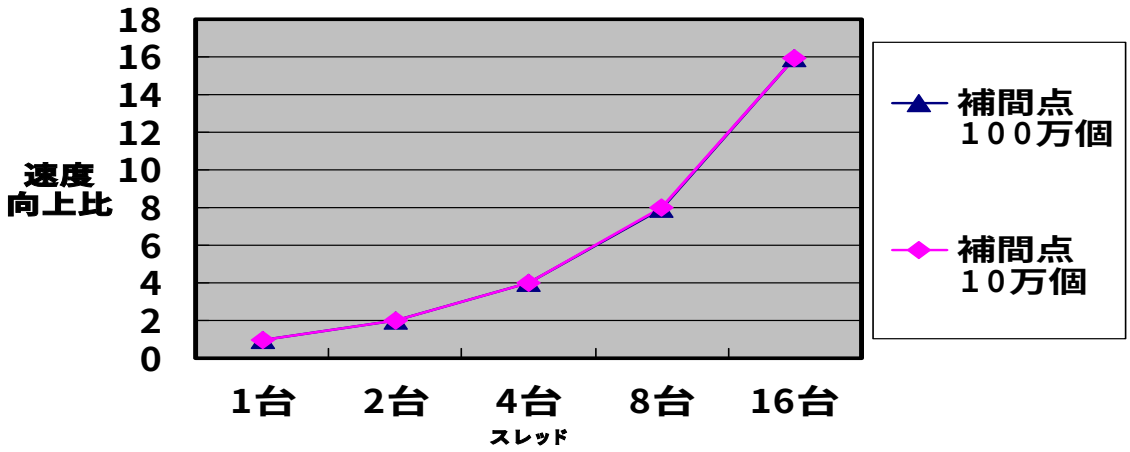

図 15：３次スプライン補間の速度向上比（制御点は１０００個）

５．４ 考察

データ数が少なくても多くても、１６台で約１６倍という極めて理想的な速度向上が得られた。

SCASH によって擬似的な共有メモリを実現しているわけだが、スプライン補間のように、あるスレッドでアクセスしたデータを、ずっと同じスレッドにてノンストップでアクセスできるような並列化なら、共有データへのアクセスのオーバヘッドもかからない。よって速度向上も非常に期待できる。

６．ガウス・ジョルダン法の並列化

６．１ 問題定義

連立１次方程式の解法は、直接法・反復法・共役勾配法に分類される。この章で述べる「ガウス・ジョルダン法」は、直接法の１つである。

ガウス・ジョルダン法というのは、中学校で習う「加減法」に相当する。つまり、連立方程式のある行を数倍し、他の式に足したり引いたりする方法である。人間が行なう場合、計算のしやすさを考えてどの式を何倍しても、どの変数から計算しても良かったが、掃き出し法では順序を厳密に定める。コンピュータはそちらの方が分かりやすいからである。

例として、図１６の５つの連立方程式を解くとする。

```
{ 2a + 3b + 4c + 5d +  e = 15・・（1）
{ 5a +  b + 2c + 3d + 4e = 15・・（2）
{ 4a + 5b +  c + 2d + 3e = 15・・（3）
{ 3a + 4b + 5c +  d + 2e = 15・・（4）
{  a + 2b + 3c + 4d + 5e = 15・・（5）
```

図 16：５つの連立方程式の例

一番最初に、a の項を消去することから考える。その際、対角項（丸のついた係数）を「ピボット数（軸数)」と呼ぶ。まず（１）式のピボット数である「２」で（１）式全体を割る（図１７参照)。

```
{  a + 3b/2 + 4c/2 + 5d/2 +  e/2 = 15/2・・(1)
{ 5a +  b   + 2c   + 3d   + 4e   =  15 ・・(2)
{ 4a + 5b   +  c   + 2d   + 3e   =  15 ・・(3)
{ 3a + 4b   + 5c   +  d   + 2e   =  15 ・・(4)
{  a + 2b   + 3c   + 4d   + 5e   =  15 ・・(5)
```

図 17：計算の流れ（ピボットで式全体を割った状態）

そして、図１７の（２）式の四角で囲んだ「５a」を消すために、（１）式全体を５倍にして引く。（３）式の四角で囲んだ「4a」を消すために、（１）式全体を 4 倍にして引く・・・、という感じでこれらの処理を繰り返し、それぞれの a の項を消去していく。

次に、（２）式の b の項をピボットとして以上の処理を繰り返す。最終的に

```
{ a       = 1 ・・ (1)
{   b     = 1 ・・ (2)
{     c   = 1 ・・ (3)
{       d = 1 ・・ (4)
{         e = 1 ・・ (5)
```

となり、答えが求められる。

しかし、上記の方法には問題点がある。それは、対角項の係数が「０」（ピボットが０）の場合に、「０」除算を行なってしまう事である。またピボットが０でなくても、０に近い数字だと、除算などの計算をするたびに式全体の値が大きくなり、丸め誤差が生じる可能性も出てくる。

よってピボットの値で除算をしようとするたびに、ピボットの値の絶対値が最大になるような行を探して、行を入れ換える。これをピボット選択という。

ガウス・ジョルダン法に、ピボット選択のアルゴリズムを加えることで、信頼性が増す。

６．２ 並列化手法

図１６の問題例から、行を分割して、それぞれのスレッドで１つのピボットを用いて並列化しようとしたものが図１８である。例えば１行目と２行目をスレッド０に、３行目と４行目と５行目をスレッド１に、・・・、という感じで割り当てる。しかし、ピボット自身が存在する行に対して除算を行なっても、その後に、他のスレッドでピボットを持つ行からの計算をされて、式の値が変わってしまう可能性も十分ある。これを防ぐには、ピボット自身が存在する行に対して除算をする処理のタイミングを制御する（同期を取る）。だがこのやり方は、逐次処理と変わらなくなってしまう。

$$
\begin{cases}
(2a) + 3b + 4c + 5d + e = 15 \\
5a + b + 2c + 3d + 4e = 15 \\
\hline
4a + 5b + (c) + 2d + 3e = 15 \\
3a + 4b + 5c + d + 2e = 15 \\
a + 2b + 3c + 4d + 5e = 15
\end{cases}
$$

図 18：ガウス・ジョルダン法の並列化その１

もう一つの並列化の方法を考えてみる。図１７の状態から、ピボットを持っていない行を分割して並列化を行なう（図１９参照）。これだと依存関係がない。例えば、２行目と３行目をスレッド０に、４行目と５行目をスレッド１に、・・・、という感じで割り当て、１行目を数倍して足したり引いたりしてピボットの項（a の項）を削除する。次のピボットを選択しようとするたびに（次の解 x を計算しようとするたびに）、バリア同期を取る。

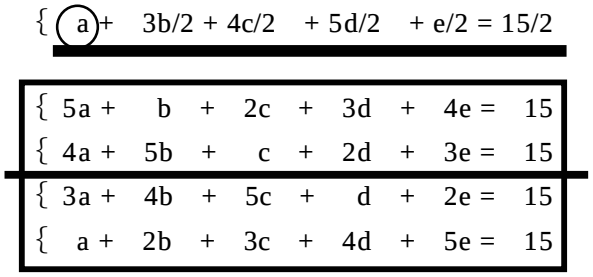

図 19：ガウス・ジョルダン法の並列化その２

またピボット選択については、ピボットの存在する列において、複数のピボット候補の係数をブロック分割する。そして各スレッド内で、絶対値が最大のものを探す。その後、スレッドごとに求めたピボットの候補をそれぞれ比較し、全てのピボット候補の中から絶対値が最大のものを見つける（図２０参照)。但し、スレッド間で最大値を比較するために、同期を取る必要がある（オーバーヘッドがかかる)。

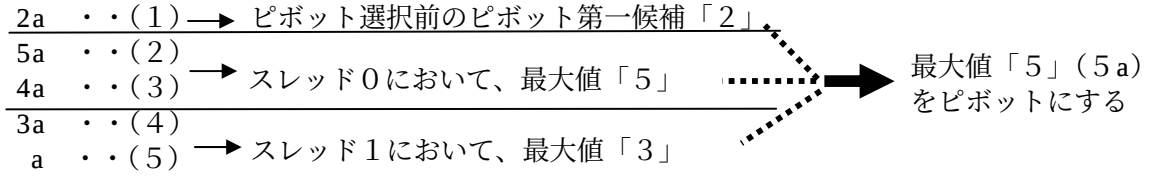

図 20：ピボット選択の並列化

図２０よりピボットは「５」となり、（１）式と（２）式を入れ替えると、以下のようになる。

{ 5a + b + 2c + 3d + 4e = 15・・（１）
{ 2a + 3b + 4c + 5d + e = 15・・（２）
{ 4a + 5b + c + 2d + 3e = 15・・（３）
{ 3a + 4b + 5c + d + 2e = 15・・（４）
{ a + 2b + 3c + 4d + 5e = 15・・（５）

ピボット「５」を中心にして、処理を進める。

６．３ 実行結果

表 ５：ガウス・ジョルダン法の並列実行結果、単位は秒

	１台	２台	４台	８台	１６台
ピボット選択並列 方程式６０００個	１５６２４	９８８３	５２７９	２８４６	１７７５
ピボット選択並列 方程式３０００個	２３１９	１５２４	８４４	４７２	３３１
ピボット選択逐次 方程式６０００個	１５６２４	１１０９６	６７０５	４７７９	３４７２
ピボット選択逐次 方程式３０００個	２３１９	１８４４	１２９５	９６８	８０６

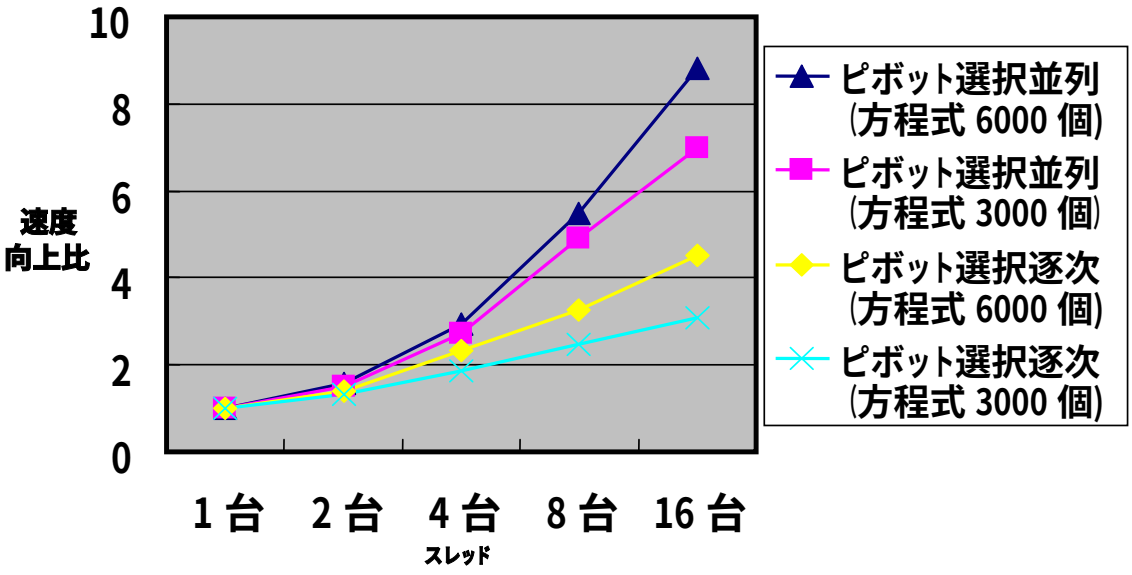

図 21：ガウス・ジョルダン法の速度向上比

（注）方程式６０００個で、約１４０ＭＢほどメモリが必要になっている。４章のスプライン曲線の実行結果と同じで、SCASH の制約により、これ以上データ数（メモリ量）を増やして実行することはできない。

６．４ 考察

ピボット選択並列化の同期によるオーバーヘッド以上に、処理時間が短縮した。これにより、ピボット選択を並列化させた方が処理時間が速くなる事が分かった。またデータ数を増やせば増やすほど、速度が向上した。

しかし方程式６０００個で逐次処理時間約１万５０００秒と、計算の負荷が十分かかっているのに、理想的な速度向上を得られていない。これは、ピボットが存在する行全体を除算後に並列化したことにより、逐次の割合が多く残ってしまったためだと考えられる。

７．ガウス・ザイデル法の並列化

７．１ 問題定義

連立 1 次方程式 Ax=b を反復法で解く解法の１つである。係数行列 A を対角成分が 0 の行列 A”と対角行列(対角以外は０の行列)D との和に分解すれば、方程式は、$(A'' + D)x = b$
すなわち　$x = D^{-1}(b - A''x)$　と書ける。したがって、$x \leftarrow D^{-1}(b - A''x)$ を何回も繰り返し、収束すればその x が解である（x の初期値は０とする）。いつも収束するわけではないが、「i 行目の対角成分の絶対値 ＞ i 行目の非対角成分の絶対値の総和」の条件を満たせば確実に収束する。この方法は係数の書き換えがないので、収束条件を満たす問題ならガウス・ジョルダン法より高速化できる。実際、工学的問題では対角成分が大きくなることが多く、ガウス・ザイデル法の適用範囲は多い。

収束は x^{k} と x^{k+1} の差が十分小さくなったことで判断する（今回、差は０．００１とする）。
また、適当な回数（今回は１００回）を決めてそれを越えると反復を止める。
以上の操作を式に表すと以下のようになる。

①(x は今回求めた解)　②(x は前回求めた解)

$$x_i^{(k)} = \frac{b_i - \boxed{\sum_{j=0}^{i-1} a_{ij} x_j^{(k)}} - \boxed{\sum_{j=i+1}^{n-1} a_{ij} x_j^{(k-1)}}}{a_{ii}}$$

図 22：ガウス・ザイデル法の計算式

ｋは計算回数である。

	x[0]	x[1]	x[2]	x[3]	x[4]
1 回目	0.672	1.443	0.854	1.586	0.704
2 回目	0.789	1.312	0.976	1.439	0.844
3 回目	0.816	1.240	・	・	・
・	・	・	・	・	・
・	・	・	・	・	・

図 23：ガウス・ザイデル法の計算処理の流れ

図２３は、５つの連立方程式の解が全て「１」である時の処理の流れを示している。例えば３回目の x[2]を求める場合、図２２の計算式において、四角で囲んだ①の部分にて 3 回目で求めた x[0]＝0.816 と x[1]= 1.240 が必要になり、四角で囲んだ②の部分にて２回目で求めた

x[3]=1.439、x[4]=0.844 が要求される。つまり、k 回目の x[i]を計算するなら、同じ k 回目の x[0]、x[1]、・・・、x[i-1]を先に求める必要がある。

また、対角要素 a_{ii} (i=1,2,・・・,n)は「０」でないとする。

７．２ 並列化手法１

図２２の計算式を見ると、時間のかかる処理は、和を計算する四角で囲んだ①と②の部分であることが予測できる。そのため、①と②の処理を並列化させる。

まず、計算回数 k を分割する方法が考えられる。例えば、０回目の解 x と１回目の解 x の計算をスレッド０に、２回目の解 x と３回目の解 x の計算をスレッド１に、・・・、という感じである。しかし、何回目で解 x が収束するかは分からないため、k を分割する並列化は不可能である。

次に、図２２の計算式の j を分割して並列化させる方法がある。例えば、j=0 と j=1 の時の計算をスレッド０に、j=2 と j=3 の時の計算をスレッド１に、・・・、という流れである。これだと依存関係がないので、プロセッサファーム（ブロック分割）で並列化可能である。

但し k 回目の解 x[i]を計算するなら、同じｋ回目の解 x[0]～x[i-1]が必要になってくるため、次の解 x を求めようとするたびにバリア同期を取る必要がある。

７．３ 手法１の実行結果と考察

表 ６：ガウス・ザイデル法の手法１による並列実行結果、単位は秒

	１台	２台	４台	８台	１６台
方程式６０００個	３７８	４４９７	６９４７	１２９３６	２３６５４
方程式３０００個	８７	９５３	１５１６	１８８３	２９６１

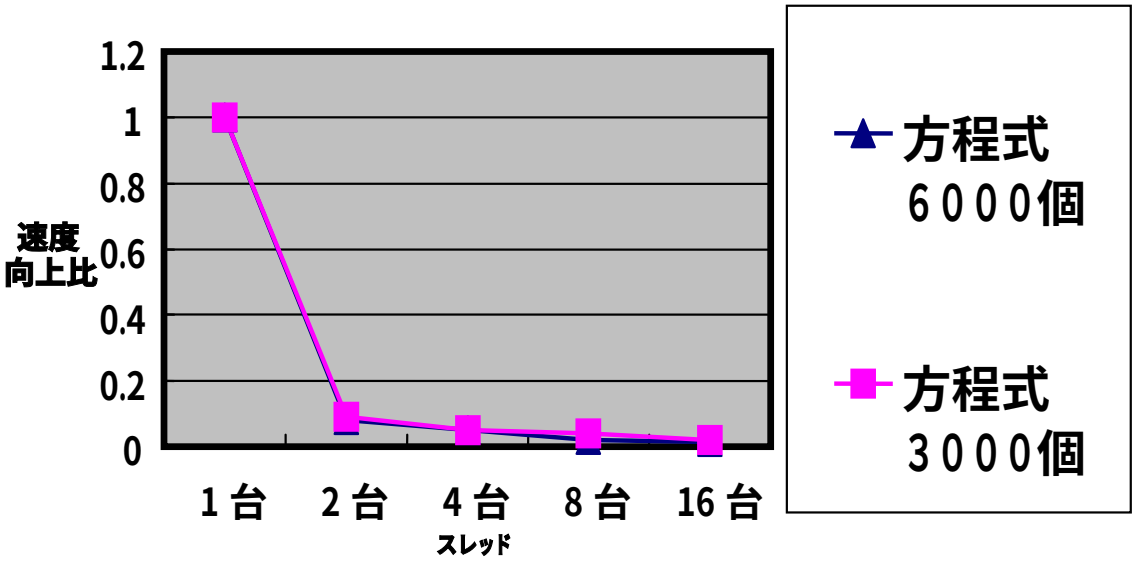

図 24：ガウス・ザイデル法の手法１による速度向上比

（注）方程式６０００個より大きなデータでの実行は、SCASH の制約によりできない。

並列化させると、データ数を大きくしても逐次処理より遅くなってしまった。

原因は、j を分割して並列処理を行なっても、逐次処理部分の割合が全プログラム中に大きく占められているからである。逐次性が高いプログラムは、バリア同期やスレッド生成・消滅のオーバーヘッドがどうしても目立ってしまい、並列効果が得られない。

７．４ 並列化手法２

図２２の計算式の i を分割して並列化させる方法を考える。i を分割するということは、計算式より、解 x[i]の計算を分割してスレッドに割り当てる意味にもなる。例えば、i=0 つまり k 回目の x[0]を求める計算をスレッド０に、i=1 つまり k 回目の x[1]を求める計算をスレッド１に、・・・、という感じである。解 x[i]は収束条件を満たすまで、k 回繰り返し計算されるので、並列化アルゴリズムは繰り返し変換になる。

ここで図２２の計算式の四角で囲んだ②に関して注目してみる。k 回目の x[i]を求めるなら、この②の部分の処理にてｋ－１回目の解 x[i+1]～解 x[N-1]までの値が必要になるが、これは既に計算されている値である。よって、この部分ではプロセッサファームで独立に並列化できる。

次に図２２の計算式の四角で囲んだ①に注目する。この部分では、k 回目の解 x[i]を求める

には、同じ k 回目の解 x[0]～解 x[i-1]までの値が必要になる。つまり、依存関係がある。そのため、一対多通信のパイプライン処理で並列化を行なう。OpenMP は共有メモリ型の並列プログラミング言語なので、メッセージパッシングは使わず、共有変数へのアクセスによりパイプライン（データ伝達）を実現する。

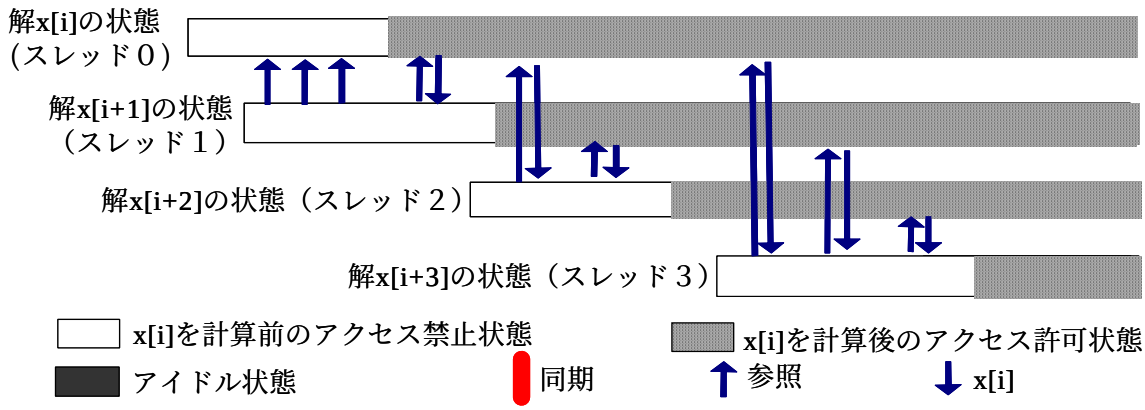

図 25：フラグ判定によるパイプライン処理の流れ

図２５は、パイプライン処理の流れを示したものである。例として、スレッド２で x[i+2]を求めたいとする。x[i+2]を計算するためには、x[i]と x[i+1]が必要である。x[i+1]は、x[i]が分からないと求められないので、まずスレッド２は x[i]の値を参照しようとする。しかし、スレッド０は x[i]を求めるまでは、他のスレッドのアクセスを許可しない。x[0]を計算後、共有変数のフラグを立て、全スレッドに対してアクセスを許可する。この処理を繰り返す。

しかしこの方法では問題が生じる。どういうわけか、あるスレッドでアクセス許可を与えても、他のスレッドからはアクセス不可状態に見えてしまうのである（共有変数が更新されない）。よって、永久に処理が終わらない状態になる。全スレッドで共有変数は同じアドレスの場所に存在する事を確認したので、おそらく同期の問題だと思われる。

そこで、バリア同期を取るやり方でパイプライン処理を実現する。

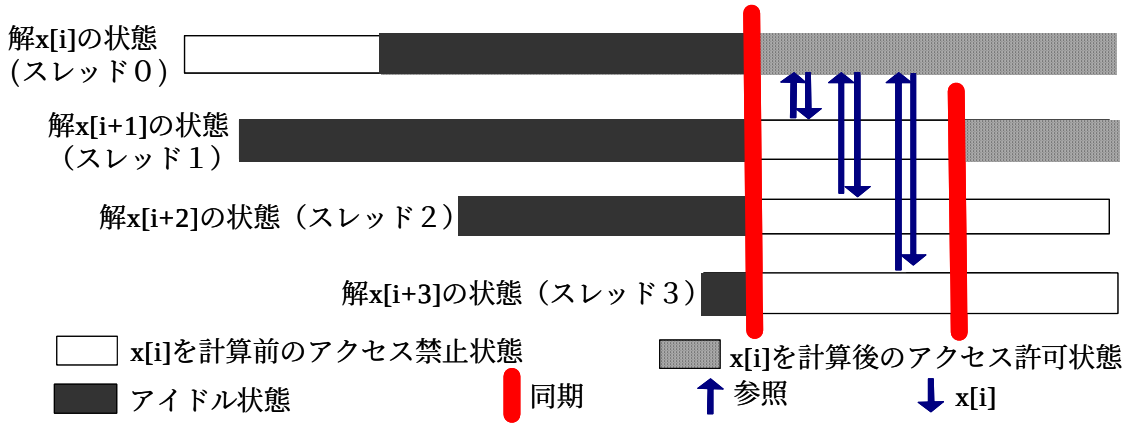

図 26：バリア同期によるパイプライン処理の流れ

図２６は、バリア同期を用いたパイプライン処理の流れを示したものである。しかし、この方法にも問題点がある。それは、OpenMP の仕様上、同期を取る場所に全スレッドが到着しないと先に進めないという点である。フラグ判定によるパイプライン処理の場合（図２５参照)、解 x[i]を計算したスレッド０は、次の解 x を求めるために次のステップに進むことができた。一方、バリア同期を使うパイプライン処理（図２６参照）だと、スレッド０は解 x[i]を計算した後も、スレッド３が到着するまで何もせずにずっと待たないといけない（次のステップに行けない)。非常に効率の悪い処理となってしまう。

（注）他の並列プログラミング言語には、signal-wait 構文のような概念がある。

```
[スレッド０]
x=a;   /*共有変数 x に a を代入*/
signal(&req);    /*他のスレッドに知らせる*/

[スレッド１]
wait(&req);   /*知らせを受け取るまで待つ*/
p=x;      /*共有変数 x を参照*/
```

signal-wait 構文を使うと、「スレッド１は、スレッド０からの知らせを受け取るまでは wait 状態であり、知らせが来れば処理を開始する」という処理が作れる。まさにパイプライン処理のための構文だが、OpenMP にはこれに似た構文はない。

７．５ 手法２の実行結果と考察

表 ７：ガウス・ザイデル法の手法２による並列実行結果、単位は秒

	１台	２台	４台	８台	１６台
方程式１０００個	１５．２	３１．８	３８．０	４９．６	５６．３
方程式５００個	３．１	１４．４	１７．５	２６．３	３５．７

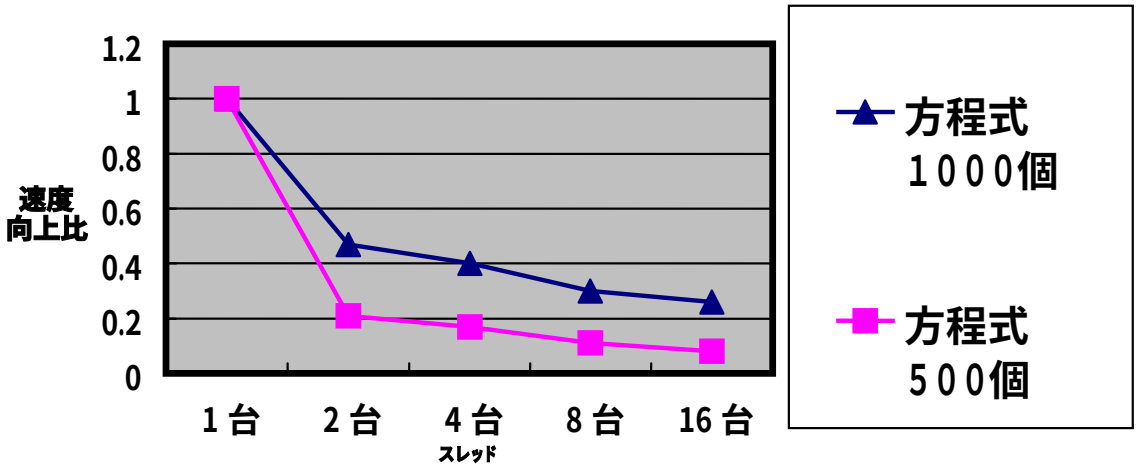

図 27：ガウス・ザイデル法の手法２による速度向上比

（注）SCASH の制約により、方程式１０００個より大きいデータを用いて並列実行することはできなかった。

図２６の処理の流れの通り、解 x を計算しても先に進むことができないため処理時間が大幅にかかり、並列化させると逐次の時より遅くなってしまった。台数を増やすほど、全スレッドが到着するまでの待ち時間が増え、全体の処理時間も長くなった。

これを改善するためには、MPI などの分散メモリ型並列プログラミング言語により実現することが考えられる。MPI はメッセージパッシングにより並列化を行ない、signal-wait 構文を

使って特定のスレッドに対してのみ同期を取ることができる。これにより、図２５の効率の良いパイプライン処理が実現できる可能性が高い。

８．おわりに

本研究では、「Ｎクイーン」、「スプライン曲線」、「スプライン補間」、「ガウス・ジョルダン法」、「ガウス・ザイデル法」の計５つを Score 型 PC クラスタ上で並列実行し、処理時間の測定を行なった。その結果、以下の事が分かった。

（１）OpenMP は、同期が必要なく各スレッドで独立に計算できる問題に対しては、他の並列プログラミング言語に比べると簡単に理想的な速度向上を得ることができる。

（２）パイプライン処理など同期が必須の問題に対しては、オーバーヘッドで速度が上がらなかったり、各スレッドの動きを把握しにくくデバックが難しいなどの欠点がある。

（３）Score の SCASH によって仮想的な共有メモリを作っているため、ユーザーが使うことのできるメモリ量に制限がある。

上記の（１）については、元々OpenMP は fork-join の考え方を用いているため、プロセッサファームで並列化できる問題に対しては非常に強いというのが理由である。

上記の（２）と（３）については、７．５章の考察にも書いたが、分散メモリ型並列プログラミング言語で、現在の主流である「MPI」にて作成することにより解決するのではないかと考えている。それは、MPI はメッセージパッシングにより並列化を行ない、signal-wait 構文を使って特定のスレッドに対してのみ同期を取ることができるからである。また、分散メモリ型なので Score を必要としない。よって使用するメモリ量に制限がなく、データをもっと大きくして実行する事ができる。しかしながら、MPI はプログラミング時に全ての通信をスケジューリングするため、プログラマには多大な負担となる。また、本研究室のＰＣクラスタ上での MPI による並列化は、まだほんの一部の問題にしか行なわれていない。

これらの事をふまえて、今後の課題として次の点が挙げられる。

（１）ＰＣクラスタ上での MPI による並列化の実績を作る。

（２）独立に計算できる箇所は OpenMP で、パイプラインなどの同期が必要になる箇所は MPI という風に、１つのプログラム上で２つの並列プログラミング言語を組み合わせてお互いの長所を生かした並列化を行なう。

謝辞

本研究の機会を与えてくださり、様々なアドバイスを頂きました山崎勝弘教授、小柳滋教授に心より感謝いたします。また本研究にあたり、励ましの言葉や貴重な意見を頂きました本研究室の皆様、先輩方に心より感謝いたします。特に、クラスタが故障した際には迅速に対応していただいた大学院生の林雅樹氏には深くお礼を申し上げます。

参考文献

[1]南里豪志,天野浩文：OpenMP 入門(1)、九州大学情報基盤センター研究部、2001.
http://www.cc.kyushu-u.ac.jp/RD/watanabe/RESERCH/MANUSCRIPT/KOHO/OpenMP/openmp0109.pdf

[2]南里豪志,渡部善隆：OpenMP 入門(2)、九州大学情報基盤センター研究部、 2002.
http://www.cc.kyushu-u.ac.jp/RD/watanabe/RESERCH/MANUSCRIPT/KOHO/OpenMP/openmp0201.pdf

[3]南里豪志：OpenMP 入門(3)、九州大学情報基盤センター研究部、2002.
http://www.cc.kyushu-u.ac.jp/RD/watanabe/RESERCH/MANUSCRIPT/KOHO/OpenMP/openmp0209.pdf

[4]バリーウイルキンソン,マイケルアレン（著）, 飯塚肇,緑川博子（訳）：
並列プログラミング入門―ネットワーク結合 UNIX マシンによる並列処理、
丸善、2000.

[5]P.パチェコ（著）,秋葉博（訳）：MPI 並列プログラミング、培風館、2001.

[6]河西朝雄：C 言語によるはじめてのアルゴリズム入門、技術評論社、1992.

[7]奥村晴彦：C 言語による最新アルゴリズム事典、技術評論社、1991.

[8]峯村吉泰：C と Java で学ぶ数値シミュレーション入門、森北出版、1999.

[9]服部雄一：C 言語と PAD による数値計算、培風館、1995.

[10]中島康彦:C 言語で「やりたい」ことを「できる」にかえる基本の 12 章､メディアテック出版、2001.

[11]WilliamH.Press,WilliamT.Vetterling,SaulATeukolsky,BrianP.Flannery（著）,
丹慶勝市,佐藤俊郎,奥村晴彦,小林誠（訳）：ニューメリカルレシピ・イン・シー 日本語版―C 言語による数値計算のレシピ、技術評論社、1993.

[12]High Performance Programming
http://www.na.cse.nagoya-u.ac.jp/~reiji/lect/hpc02/

[13]誰にでもわかる OpenMP
http://www.ulis.ac.jp/~hasegawa/DELL/OpenMP.html

[14]PC Cluster Consortium
http://www.pccluster.org/

[15]OpenMP
http://www.openmp.org

[16]湯浅太一,安村通晃,中田登志之:はじめての並列プログラミング(bit 別冊)、共立出版、1998.

[17]R.Chandra,L.Dagum,D.Kohr,D.Maydan,J.McDonald,R.Menon：
Parallel Programming in OpenMP、Morgan Kaufman Publishers、2000.

[18]佐藤三久：JSPP'99 OpenMP チュートリアル資料、RWPC、2000.

[19]石川裕,堀敦史,原田浩,佐藤三久,住元真司,高橋俊行：
Linux で並列処理をしよう、共立出版、2002.

[20]OpenMP C/C++ アプリケーション プログラム インタフェース 日本語版、RWPC、2000.

[21]山崎勝弘：コンピュータは進化する、1999.

[22]古川智之,松田浩一,安藤彰一：並列プログラム事例集、高性能計算研究室、1996.
/common/jirei/all/caseset

[23]米田健治,徳山美香,青地剛宙：並列プログラム事例集 2、高性能計算研究室、1999.
/common/jirei/caseset2

[24]大村浩文：PC クラスタの動作テストと OpenMP 並列プログラミング、立命館大学理工学部情報学科卒業論文、2002.

[25]内田大介：OpenMP による並列プログラミング（Ⅰ）、立命館大学理工学部情報学科卒業論文、2000.

[26]三浦誉大：PC クラスタ上での並列プログラミング環境の構築、立命館大学理工学部情報学科卒業論文、2002.

[27]林雅樹：PC クラスタ上での有限要素法によるカルマン渦の並列化、立命館大学理工学部情報学科卒業論文、2003.

[28] 柿下裕彰：PC クラスタ上での OpenMP 並列プログラミング（I)、立命館大学理工学部情報学科卒業論文、2003.

[29]黒川 耕平：PC クラスタ上での OpenMP 並列プログラミング（Ⅱ)、立命館大学理工学部情報学科卒業論文、2003.