卒業論文

PC クラスタの動作テストと OpenMP 並列プログラミング

氏　　名　：大村　浩文
学籍番号　：2210980047-7
指導教員　：山崎　勝弘　教授

提 出 日　：2002年2月18日

立命館大学　理工学部　情報学科

内容梗概

並列処理は大規模な問題でも計算時間が大幅に短縮できるというメリットがあり、欠かせない技術の一つといえる。近年では、共有メモリ計算機の普及に伴い、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モデル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、プログラミングも比較的簡単なので、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。また、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。本論文では、本研究室で構築した SCore 型 PC クラスタの動作テストと、OpenMP での並列プログラミングについて述べる。

PC クラスタの動作テストは、共有メモリ環境の SMP マシン上で動作している、ラグランジェ補間、マンデルブロー集合、ビジネル暗号、ランレングス圧縮、KMP 法、BM 法の6つのプログラムについて実行した。その結果、ビジネル暗号とマンデルブロー集合に関しては、ソフトウェア分散メモリ(SCASH)に関するエラーが出て、スレッド数を増やして実行することはできなかった。その他の4つのプログラムに関しては、スレッド数16台まで動作することができた。また、マンデルブロー集合については、どの範囲までなら実行可能かを検証し、ごくわずかな要素数の配列しか共有できないということがわかった。

OpenMP のプログラミングでは、積分、モンテカルロ法によるπの計算と、SCASH の制約にかからないように、データ構造を変更してマンデルブロー集合のプログラミングを行った。どれも、スレッド数を増やすにしたがって、理想的な速度向上が得られた。モンテカルロ法によるπの計算には、乱数発生に rand 関数を用いたのだが、分散共有メモリ環境の PC クラスタ上では正しく動作したのだが、共有メモリ環境の SMP マシン上では、速度向上が得られなかった。これは、共有メモリ環境では、関数呼び出しの際に各スレッドが同じメモリ空間にアクセスし、値を参照するとき競合がおこってしまったためだと考えられる。

目次

1.はじめに.. 1

2.並列処理と OpenMP 2

2.1 並列プログラミング 2
2.2 PC クラスタ 5
2.3 OpenMP... 7
2.3.1 OpenMP とは.. 7
2.3.2 OpenMP の実行モデル 7
2.3.3 OpenMP の API.. 8

3. PC クラスタのテスト 11

3.1 PCクラスタの構成... 11
3.2 既存の OpenMP テストプログラム 12
3.3 テスト結果 12
3.4 考察.. 15

4. OpenMP によるπの計算... 16

4.1 積分による計算... 16
4.1.1 理論 16
4.1.2 実行結果.. 16
4.1.3 考察 16
4.2 モンテカルロ法による計算.. 18
4.2.1 モンテカルロ法 18
4.2.2 実行結果.. 18
4.2.3 考察 19

5. OpenMP によるマンデルブローの計算.. 20

5.1 問題定義... 20
5.2 並列化手法 20
5.3 実行結果... 20
5.4 考察.. 22

6. おわりに... 23

謝辞 24

参考文献 25

付録 26

図目次

図 1：共有メモリモデル 3
図 2：分散共有メモリ 4
図 3：分散メモリモデル 4
図 4：SCore クラスタシステムソフトウェア 6
図 5：fork- join モデル 7
図 6：クラスタの構成 11
図 7：積分による計算の実行時間 17
図 8：積分計算の速度向上 17
図 9：モンテカルロ法 18
図 10：モンテカルロ法の速度向上比 19
図 11：速度向上比（解像度２８０×３５０） 21
図 12：速度向上比（解像度２８００×３５００） 21

表目次

表 1：PC クラスタ上でのテスト結果 12
表 2：ラグランジェ補間の実行結果 12
表 3：ランレングス圧縮の実行結果 13
表 4：ＫＭＰ法の実行結果 13
表 5：ＢＭ法の実行結果 13
表 6：マンデルブロー集合の実行結果 14
表 7：実行結果（データ数：１２０万） 18
表 8：実行結果（データ数：１２００万） 18
表 9：実行結果（データ数１億２千万） 19
表 10：実行結果（ブロック分割） 20
表 11：実行結果（サイクリック分割） 20

1.はじめに

並列処理の研究の応用分野として気象予測、環境問題、流体計算、デジタル画像処理、遺伝子の解明、データベース処理などがあげられる。特に今後、画像処理などのマルチメディアなどとともに、最も並列処理の活用が広がると考えられているのが、データベース処理である。データベース処理は処理並列性があるばかりではなく、並列 I/O の点で計算機クラスタに向いた性質がある[7]。

並列の計算機には３つのメモリモデルがある。複数のプロセッサがメモリバス/スイッチ経由で、主記憶に接続された共有メモリモデル（SMP）。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、プロセッサはほかのプロセッサの主記憶の読み書きができる、共有分散メモリ。プロセッサと主記憶からな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ができない、分散メモリがある。分散メモリに即した並列プログラミングライブラリとしては、PVM（Parallel Virtual Machine）、および、MPI(Message Passing Interface)が有名である。また、共有メモリに即した並列プログラミングモデルとしては、OpenMP が主流になってきている[1]。

最近の汎用マイクロプロセッサの高性能化は凄まじく、一昔前までは手の出なかった高性能スーパコンピュータに匹敵する、マイクロプロセッサが今や WS や PC に使用され、さらにネットワークにおいても近年目覚しい発展がある。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。結果として，PC、WS 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[7][8]。

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の中の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、クラスタコンピューティングのための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、ワークステーションや PC 等で稼動しているオペレーティングシステムである 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もつ。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を可能にする。これによって、Score クラスタシステム上での OpenMP プログラミングが可能になっている。

本研究では、OpenMP によるプログラミングを、SUN サーバ（SUN Enterprise 4CPU）、2 CPU-PC の共有メモリマシンと PC16台の SCore 型クラスタ上で行った。2章では並列処理と OpenMP の説明を示す。3章ではPCクラスタの動作テスト、4章では積分とモンテカルロ法による π の計算、5章ではマンデルブロー集合の計算の実行結果と考察を示す。

2.並列処理とOpenMP

2.1 並列プログラミング

2.1.1 並列アルゴリズム

並列アルゴリズムは一般的に大きく4つに分類される[6]。

(1) 分割統治法

まず問題に対して何らかの計算を行う。そしてその結果をある条件をもとに分割し、またその計算を行う。ある条件が満たされるまで計算と分割を繰り返していき、その部分解を全て統合することによって結果を得る。

(2) プロセッサファーム

まずマスターによって処理を開始する。マスターは与えられた問題に対し複数の独立した計算に分割し、マスターと各スレーブがそれぞれ計算を行い、その結果を再びマスターが回収し結果を得る。このアルゴリズムはマスターとスレッドで独立して計算が行われるので、並列効果が出やすいといわれる。

(3) プロセスネットワーク

ある問題に対して、計算ステージを複数に分割する。データはあるステージで計算され、それが終わったら次のステージに移り計算される。というふうに各ステージを複数のデータが流れていく。パイプライン処理とも呼ばれる。

(4) 繰り返し変換

ある問題を複数のオブジェクトに分割し、各オブジェクトは複数の計算の繰り返しによって値が変更される。オブジェクトの値がある条件を満たすまで繰り返し実行することで解を求める。

2.1.2 並列プログラミング言語とアーキテクチュア[1]

並列プログラミングを行う上でプログラマが意識すべきアーキテクチュア上の特徴で、プログラミングに影響を与える因子として、メモリモデルが挙げられる。主な、メモリモデルとしては、共有メモリ、分散共有メモリ、分散メモリがある。

共有メモリマシンは、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。このアーキテクチュアを有するシステムのことを SMP（Symmetrical Multi Processor）と呼び、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。このアーキテクチュアに即した並列プログラミングライブラリとして pthread、や OpenMP がある。

分散共有メモリマシンと分散メモリマシンのアーキテクチュアの差はほとんど減ってきている。どちらも、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あり、どちらも大規模なシステムの構築が可能であるという特徴がある。分散共有メモリマシンでは、プロセッサは、他のプロセッサの主記憶を読み書きすることができる。最近の PC クラスタは分散共有メモリの形態を有しており、PC クラスタ上でも共有メモリ用の並列プログラミングライブラリが利用できるようになっている。分散メモリマシンは、プロセッサは他のプロセッサの主記憶の読み書きを行うことはできず、必ず相手のプロセッサに介在してもらう必要がある。PC や WS を LAN で接続したものもこの形態に属し、代表的な並列プログラミングライブラリとして PVM（Paralel Virtual Machine）や MPI（Massage Passing Interface）などのメッセージ通信ライブラリがある。

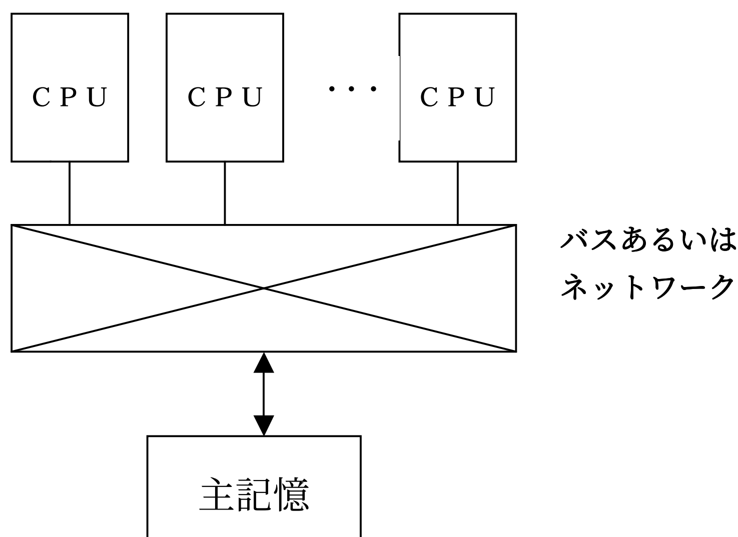


図 1：共有メモリモデル

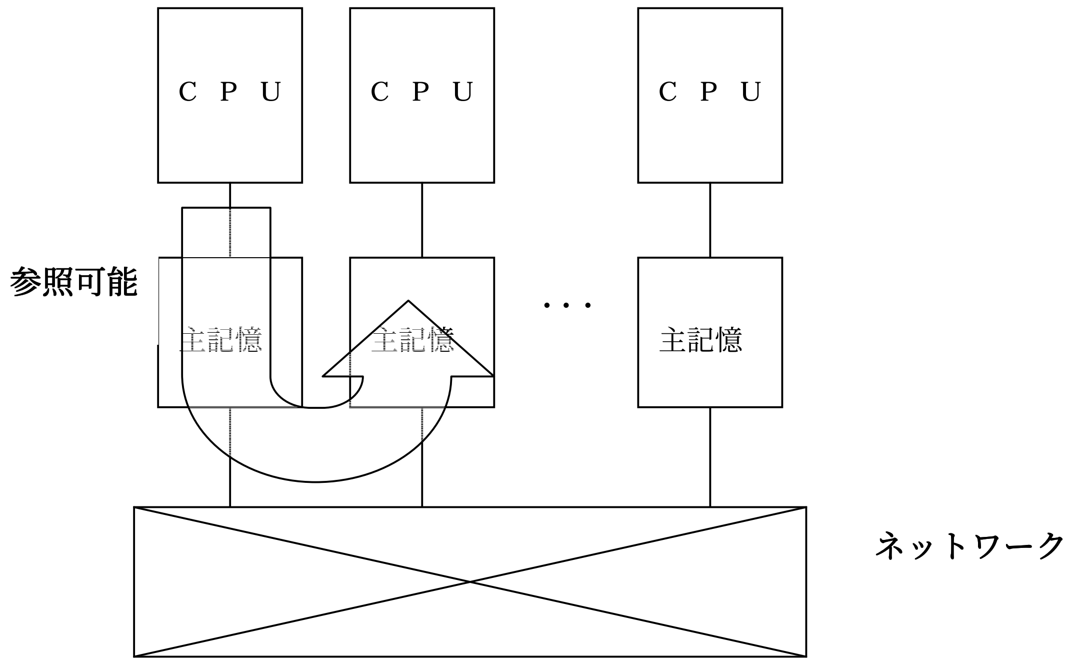


図 2：分散共有メモリ

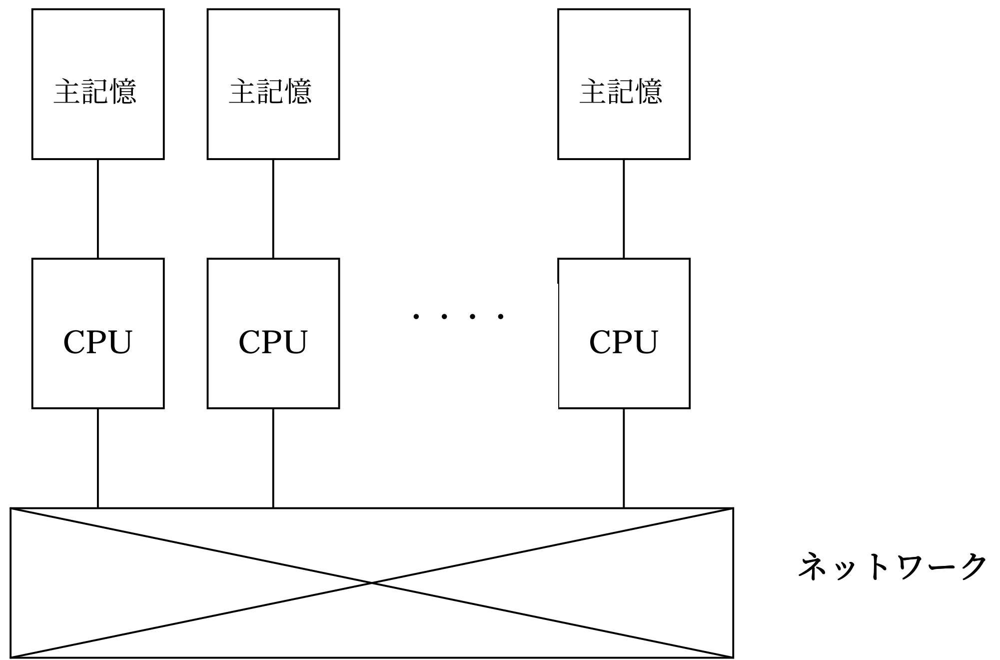


図 3：分散メモリモデル

2.2 PC クラスタ

2.2.1 PC クラスタとは

PC クラスタとは、安価な PC にフリーの OS を載せ、それを高速のネットワークで複数接続し、分散して処理を行うシステムである。PC クラスタは、1990年前後に、数千から数万台の CPU を搭載する超並列計算機の開発が進む一方で、TCP／IP ベースのネットワークで接続された複数台の計算機を仮想的に一台のマシンとしてとらえて並列プログラミングを走らせるような PVM（Parallel Virtul Machine）が開発された。PVM はそれぞれの計算機でメッセージ交換を行うメッセージ通信ライブラリであり、公開されたソフトウェアをインストールするだけで仮想的な並列処理環境が構築できた。これが PC クラスタの幕開けといえる[7]。 1995 年前後になると、イーサネットスイッチや100 Mbit Ethernet などの技術も普及し、比較的安価に高性能なネットワークの構築が可能となった。さらに、Myricom 社の Myrinet などクラスタを指向した専用の高速ネットワークが登場した。これにより、専用の並列計算機並みのスループットを持つネットワークの構築が可能となった。一方で、PC 向けのプロセッサの価格低下と急速な性能向上で、コストパフォーマンスに優れた PC クラスタの実現が可能となった。この時期に MPI フォーラムがメッセージ通信のプログラムを記述するために広く使われる「標準」を目指して作られた、メッセージ通信の API である MPI（Massafe Passing Interface）が広く普及した[7]。

現在では複数のプロセッサを搭載した SMP 型の WS や PC が比較的容易に入手可能になっており、これらをベースにした SMP クラスタが登場している。

2.2.2 SCore 型クラスタ

今までのクラスタは、インターネットで利用されている TCP／IP のプロトコル上にフリーソフトウェアの組み合わせで構築された Beowulf 型クラスタであったが、100Mbit 秒程度の低い通信性能のネットワークを使い、TCP／IP のプロトコルを使っているために専用並列計算機と比べて通信性能で比べて10分の1しか性能が出ていなかった。RWC は高性能通信ライブラリを持つ SCore と呼ばれるクラスタシステムソフトウェアを開発し、Beoulf 型クラスタの持つ欠点を取り除き、専用並列マシンと同等の性能を有するクラスタシステムの構築が可能になった[16]。SCore クラスタシステムソフトウェアの階層図を図4に示す。

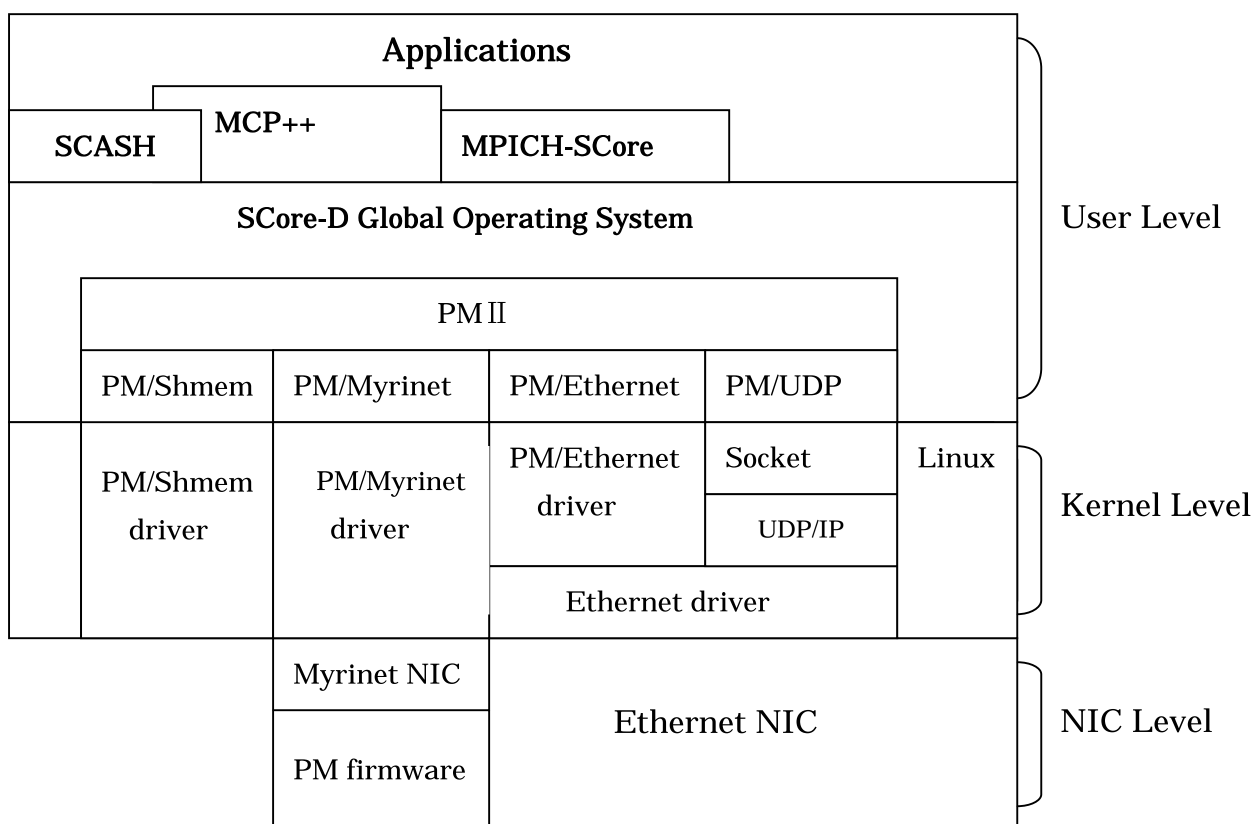


図 4：SCore クラスタシステムソフトウェア

以下に図４の用語を簡単に説明する。

- SCASH ： SCore クラスタシステムソフトウェア分散共有メモリシステムで、分散メモリのクラスタ上で共有メモリプログラミングを可能にする。

- SCore－D ： 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全てのコンピュータを制御し、ギャングスケジューリング（並列プロセス中の全てのプロセス切り替えを同期する）という手法を用いて複数の並列アプリケーションを効果的にスケジューリングする。SCore－D はクラスタをあたかも単一並列コンピュータのように見せる。

- PMⅡ ： 多種のネットワークデバイスをサポートする低レベル高性能通信ライブラリ用の API の名前である。

2.3 OpenMP

2.3.1 OpenMPとは

OpenMPは共有メモリマルチプロセッサの並列プログラミングのためのプログラミングモデルであり、ベース言語(Fortran, C, C++)をコンパイラ指示文(directives/pragma)、ライブラリ、環境変数によって拡張したものである。並列実行や同期をプログラマが明示することにより並列化を行う。また、指示文を無視することで逐次実行が可能なので逐次版と並列版を同じソースで管理でき、段階的な並列化が可能である[4]。

2.3.2 OpenMPの実行モデル

OpenMPはfork-joinによる並列実行モデルを用いている。OpenMPで記述されたプログラムは、マスタスレッドと呼ばれる単一のスレッドで実行を開始する。マスタスレッドは並列構文が現れるまで逐次リージョンを実行する。マスタスレッドは、並列指示文に遭遇すると複数のスレッドから成るチームを作成し、そのチームのマスタになる。ワークシェアリング構文に対応するブロックは、全スレッドによって実行されなければならない。全てのスレッドにより並列に実行される部分を並列リージョンという。また、ワークシェアリング構文にno wait 指示節が指定されていなければ、ワークシェアリング構文の最後で暗黙のバリア同期をチーム内の全スレッドに対して実行し、その後の処理はマスタスレッドのみが実行を続ける。1つのプログラムにおいて、いくつでも並列構文を使うことができる。従って、プログラムは実行中にforkとjoinを繰り返すことになる[5]。

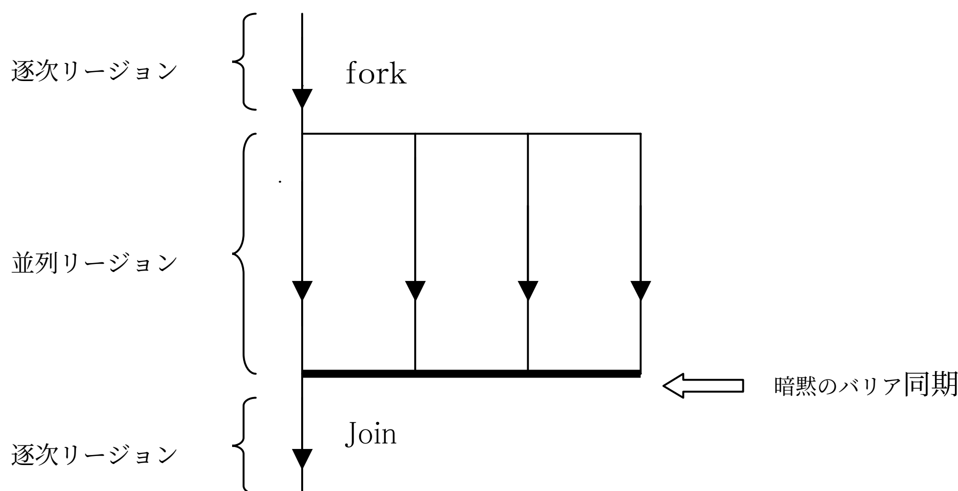


図 5：fork- join モデル

2.3.3 OpenMP の API

OpenMP の指示文は C/C++の場合、#pragma omp から始まる pragma 行である。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。以下にそれぞれの構文を簡単に説明する[5]。

(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（チーム）によって並列実行される部分であり、関数呼び出しも重複実行される。この指示文は並列実行を開始する基本的な構文である。パラレルリージョン内では実行時ライブラリ関数 omp_get_thread_num()でスレッド ID を得ることができる。元のスレッドは、スレッド ID は0になり、マスタースレッドになる。

(2)ワークシェアリング構文

ワークシェアリング構文は、この構文に到達したチームのメンバに、対応するステートメントを分割して実行させる。このとき、ワークシェアリング構文は新たにスレッドを生成しない。また、ワークシェアリング構文の入り口には、暗黙のバリア同期は存在しない。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。

①for 構文

for 指示文は直後の for ループのイタレーションを並列実行（データ並列）する。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簡単にサイクリック分割の指定ができる。For 構文の指示節で schedule の指定を行う。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チャンク（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。chunk_size が指定されない場合はイタレーションはほぼ同じサイズのチャンクに分割され、各スレッドに割り当てられる。

・dynamic

schedule(dynamic,chunk_size)が指定された場合、chunk_size 回のイタレーションのチャンクを各スレッドに割り当てる。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。chunk_size が指定されないときの既定値は1になる。

・guided

shedule(guided,chunk_size)が指定された場合、イタレーションのチャンクを徐々に　小さくしながらスレッドに割り当てる。スレッドに割り当てられたチャンクの処理が終了すると、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 chunk_size が1のときは、残りのイタレーションをスレッド数で割ったおおよその値を、チャンクとして割り当てる。このためチャンクの大きさは1に向って指数的に小さくなる。chunk_size が指定されない場合、既存値は1になる。

・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される。スケジューリングのサイズやチャンクサイズは、環境変数 OMP_SCHEDULE に設定することにより実行時に選択される。この環境変数が設定されていない場合、選択されたスケジューリングは実装依存である。schedule(runtime)を指定した場合、chunk_size は指定できない。

②section 構文

section 指示文は、チーム内のスレッドで分割して実行する構文の集合を指示する、非繰り返しワークシェアリング構文である。各セクションはチーム内のスレッドによって1度だけ実行される。

構文は以下の通りである。

```
#pragma omp sections
  {
 #pragma omp section
  { …section1…}
 #pragma omp section
  {…section2…}
 }
```

③single 構文

single 指示文は、対応する構造ブロックがチーム内の1つのスレッド（必ずしもマスタースレッドでなくてもよい）のみで実行されることを指示する構文である。構文は以下の通りである。

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

⑶**同期のための構文**

①master **構文**

マスタスレッドのみが実行する構造ブロックを指示する指示文である。マスタ以外のスレッドは master が指定されたステートメントを実行しない。マスタセクションの入り口と出口では、暗黙のバリア同期は実行されない。

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。名前のない場合は、ほかの名前がない critical section と排他的に実行。

③barrier 指示文

バリア同期を行う。チーム内のスレッドが同期点に達するまで待つ。並列リージョンのおわり、work sharing 構文で nowait 指示節が指定されない限り、暗黙のバリア同期が行われる。

④atomic 構文

複数の同時書き込みを行う可能性のあるスレッドに対して、指定されたメモリをアトミックに更新することを指示する指示文である。

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。スコープ属性指示節は、その指示節が指定されている指示文の文脈の有効範囲にある変数のみに適応される。

・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がスレッドごとに占有される。

・ firstprivate

-private と同様であるが、直前の値で初期化される。

・ lastprivate

-private と同様であるが、構文終了時に逐次実行された場合の最後の値を反映する。

・ 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。

3. PC クラスタのテスト

3.1 PCクラスタの構成

本研究の PC クラスタはサーバホストが Ethernet（100Base-TX）でコンピュータホスト16台と接続されている。サーバホストはコンピュータホストを制御するために使用される。コンピュータホストには OS に Redhat Linux 7.1 を使用し、その上に RWC が開発したクラスタシステムソフトウェア Score4.2 をインストールしてある。クラスタ間は Myrinet で接続されており、メモリ空間を共有している。

また、コンピュータホストは2つのソケットがあり、現在128MB の SDRAM を2つ実装しており256 MB になっている。256MB の SDRAM を2つ実装することで最大512MB まで増設可能である。

クラスタの構成図を以下に示す。

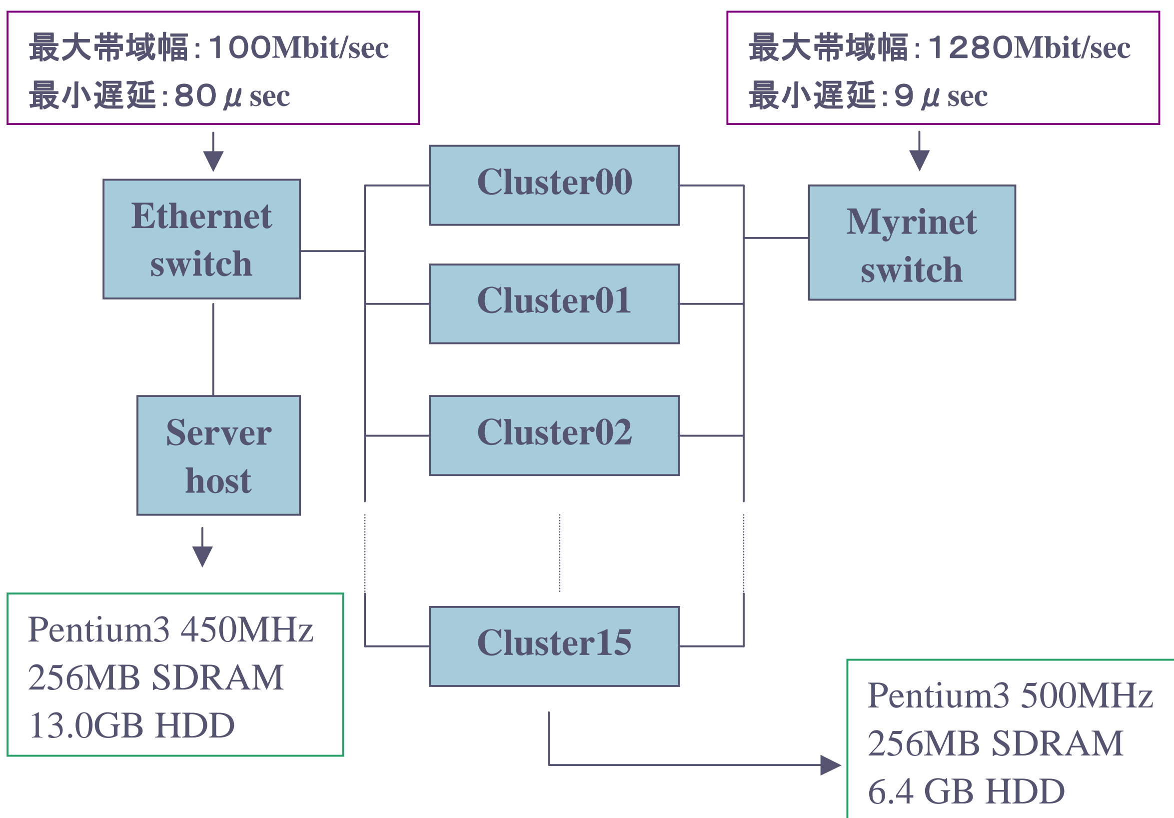


図 6：クラスタの構成

3.2 既存の OpenMP テストプログラム

本研究室では、クラスタシステムソフトウェアである Score をインストールし PC クラスタ上で分散共有メモリ環境の構築を行った。そこで、共有メモリマシン上で既に動いている OpenMP プログラムについて、PC クラスタ上で正しく実行できるかを検証した。

プログラムは、ラグランジェ補間、ビジネル暗号、マンデルブロー集合、ランレングス圧縮、文字列照合（BM 法、KMP 法）の6つである。

3.3 テスト結果

PC クラスタ上でのプログラムのテスト結果を表1に示す。

表 1：PC クラスタ上でのテスト結果

プログラム	テスト結果
ラグランジェ補間	スレッド数16で12．3倍の速度向上
ビジネル暗号	SCASH エラーで実行できず
マンデルブロー	SCASH エラーで実行できず
ランレングス圧縮	スレッド数8で3．2倍の速度向上
KMP 法	スレッド数8で3．0倍の速度向上
BM 法	スレッド数8で1．8倍の速度向上

ラグランジェ補間、ランレングス圧縮、KMP 法、BM 法の実行時間を以下に示す。

表 2：ラグランジェ補間の実行結果

スレッド数	1	2	4	8	12	16
座標数4千	46.1	24.2	13.2	8.4	7.4	7.3
座標数２万	230	116	59	31	22	18

（単位：ミリ秒）

表 3：ランレングス圧縮の実行結果

スレッド数	1	2	4	8	１２	１６
１万文字	0.98	1.4	1.7	2.2	2.8	3.5
１０万文字	9.8	8.2	5.3	4.1	4.6	5.2
１００万文字	103	73	39	32	30	29

（単位：ミリ秒）

表 4：KMP法の実行結果

スレッド数	1	2	4	8	１２	１６
１万文字	0.95	2.2	2.4	3.1	4.0	4.7
１０万文字	9.2	8.1	5.3	4.9	5.5	6.4
１００万文字	99	62	42	33	31	30

（単位：ミリ秒）

表 5：BM法の実行結果

スレッド数	1	2	4	8	１２	１６
１万文字	0.51	1.7	2.1	2.6	3.1	3.8
１０万文字	4.9	5.8	4.1	4.0	4.8	5.5
１００万文字	55	48	36	30	29	28

（単位：ミリ秒）

マンデルブローとビジネル暗号については、スレッド数が1のときは実行できるが、スレッド数を2に増やすと ompsm_scash fatal:shared arg overflow というエラーメッセージが出た。ビジネル暗号については、配列の要素数をかなり小さくしても実行できなかったが、マンデルブローの方では配列の要素数を小さくすると実行可能な場合があった。その結果を表6に示す。D の値とは解像度に関する変数であり、この値を大きくすれば解像度も良くなる。解像度が良くなるとマンデルブロー集合に入る点の個数も増える。このプログラムでは、集合に入る点の個数を数え、その点の X 座標と Y 座標の値を配列に格納している。確保する配列の要素は集合に入る点の数よりも大きい必要がある。

要素合計の求め方は、以下の１から３までの合計である。

１：マンデルブロー集合に入る点を格納する配列

２：その点の X 座標の値を格納する配列

３：その点の Y 座標の値を格納する配列

表 6：マンデルブロー集合の実行結果

D の値	点の数	配列の要素	要素合計	実行結果
0.1	166	[16][20]	656	１６台まで実行可
0.09	205	[16][20]		１６台まで実行可
0.08	264	[16][20]		１６台まで実行可
0.07	345	[16][30]	976	１６台まで実行可
		[16][40]	1296	SCASH エラー
0.06	459	[16][30]		１１台まで実行可（それ以降は時間表示がおかしくなる）
		[16][31]	1008	１１台まで実行可（同上）
		[16][32]	1040	SCASH エラー
		[16][35]	1136	SCASH エラー
		[10][48]	970	９台まで実行可（同上）
		[10][50]	1010	９台まで実行可（同上）
		[10][51]	1030	SCASH エラー
		[8][63]	1016	６台まで実行可（同上）
		[8][64]	1032	SCASH エラー

3.4 考察

共有メモリ環境の京都産業大学のSUN Enterprise 4CPU SMPマシン上で動作するOpenMPプログラムを本研究室の分散共有メモリ環境の SCore 型クラスタ上でテストをしてみた結果、ラグランジェ補間に関しては、スレッド数を増やすに従って理想に近い速度向上を得られた。ランレングス圧縮、KMP法、BM法に関しては、データ数が1万文字と10万文字のときは、データが小さすぎあまり速度向上が出ていないが、データ数が100万文字のとき、スレッド数8までは速度向上が得られた。スレッド数8以降は、ほんの少しの速度向上しか得られていない。データ数をもっと大きくするとスレッド数16のときにもそれなりの速度向上が得られるのではないかと考えられる。

ビジネル暗号とマンデルブローは、うまく動作しなかった。動作しないものはSCore上のソフトウェア分散共有メモリSCASHの制約に引っかかっていると考えられる。あまり大きなデータを共有しようとするとSCASHの制約にかかってしまうので、SCore型クラスタでOpenMPプログラミングを行うときは制約にかからないようなデータ構造を考えなくてはならない。マンデルブローのプログラムでSCASH の限界を検証した結果、配列の要素数の合計が1016のときまでは実行でき、1030のときはSCASHのエラーが出た。配列はそれぞれint型で定義されているので、一つの要素が4バイトとして4064バイトまでは、共有できると考えられる。

4．OpenMP によるπの計算

4.1 積分による計算

4.1.1 理論

$\int_0^1 \frac{dx}{\sqrt{x(1-x)}}$ の積分計算を行う。０から１までの積分範囲をＮ等分し、スレーブにそれぞれ割り当てて計算する。なお、計算には、$\int_a^b f(x)dx = \lim_{n\to\infty}\frac{b-a}{n}\sum_{i=0}^{n-1} f(x_i)\cdot d$、$x_i = a+(i+0.5)\cdot d$、$d=(b-a)/n$ の近似式を用いる。$\int_0^1 \frac{dx}{\sqrt{x(1-x)}}$ の理論上の計算は広義積分により以下のように求められる。

$$\int_0^1 \frac{dx}{\sqrt{x(1-x)}} = \lim_{\substack{\varepsilon_1\to+0\\ \varepsilon_2\to+0}} \int_{\varepsilon_2}^{1-\varepsilon_1} \frac{dx}{\sqrt{x(1-x)}} = \lim_{\substack{\varepsilon_1\to+0\\ \varepsilon_2\to+0}} \int_{-\frac{\pi}{2}+\theta_{\varepsilon_2}}^{\frac{\pi}{2}-\theta_{1\varepsilon_1}} \frac{\frac{1}{2}\cos\theta d\theta}{\sqrt{\frac{1}{4}(1-\sin^2\theta)}}$$

$$= [\theta]_{-\frac{\pi}{2}+\theta_{\varepsilon_2}}^{\frac{\pi}{2}-\theta_{\varepsilon_1}} = \pi-\theta_{\varepsilon_1}-\theta_{\varepsilon_2} = \pi$$

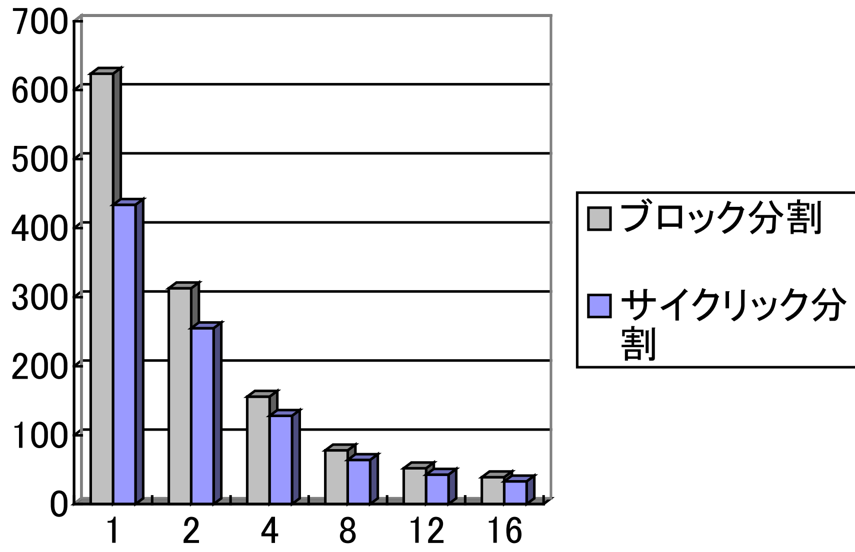
4.1.2 実行結果

計測データとして、データ数を 1200 万、1億 2 千万、12億とし、SCore 型 PC クラスタ上でスレッド数を1、2、4、8、12、16で実行した。また、データの分割をブロック分割とサイクリック分割で行い、それぞれの実行時間を比較した。サイクリック分割には OpenMP の schedule 構文を用いた。

図7に、データ数12億のときの実行結果、図8にデータ数12億のときの速度向上を示す。

4.1.3 考察

サイクリック分割のほうがブロック分割より負荷均衡がとれ、実行時間があきらかに早かった。しかし、速度向上比で見てみるとブロック分割はスレッド数16で15．7倍であったのに対し、サイクリック分割ではスレッド数16で13．1倍に留まった。精度の面から言うと、πの値はデータ数が1200万のとき3．14124、データ数1億2000万のとき3．14148、データ数12億のとき3．14158とデータ数を増やせば増やすほど精度がよくなっていくことがわかった。



1.

図 7：積分による計算の実行時間

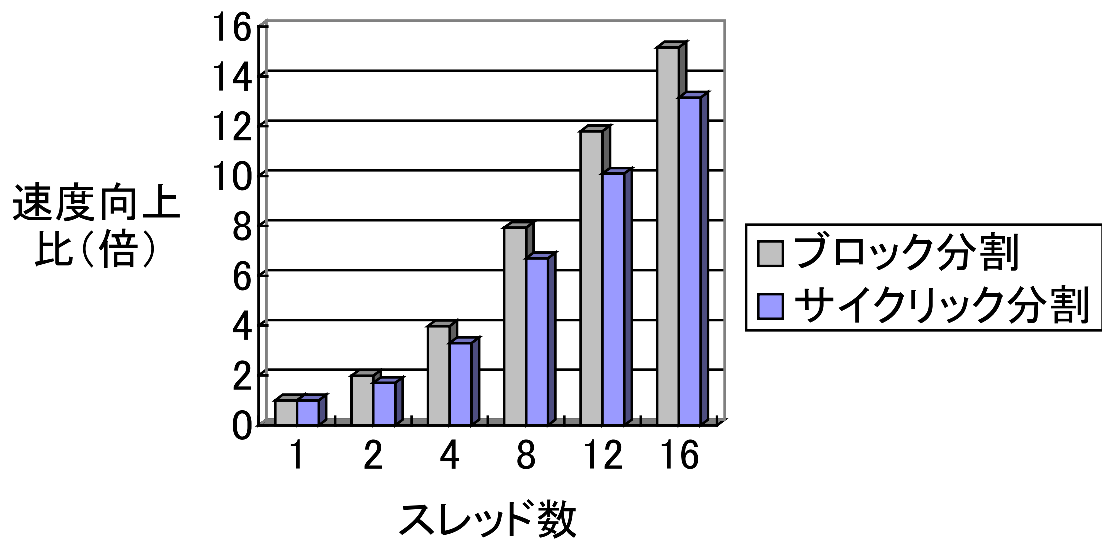


図 8：積分計算の速度向上

4.2 モンテカルロ法による計算

4.2.1 モンテカルロ法

図４に示すような一辺が１の正方形とそれに内接する４分の１円（扇型）の面積比によりπを求める。円の中の点の個数をｒ、全体の点の個数をｎとすると

１：π／４　＝　ｎ／ｒ　より　π＝４*ｒ／ｎとなる。

rand 関数により０から１までの乱数を一辺１の正方形内に発生させ、それが円内に入るかをカウントしていく。

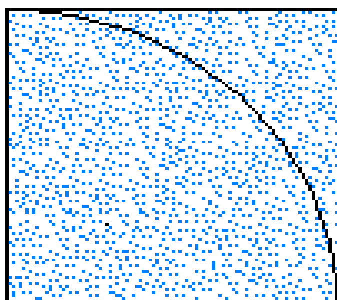


図 9：モンテカルロ法

4.2.2 実行結果

データ数を１２０万、１２００万、１億２千万、スレッド数を１から１６まで実行した結果を以下に示す。

表 7：実行結果（データ数：１２０万）

スレッド数	1	2	4	8	12	16
πの値	3.1431	3.1417	3.1407	3.1417	3.1419	3.1411
実行時間（秒）	0.5192	0.2806	0.1407	0.0727	0.0534	0.0396

表 8：実行結果（データ数：１２００万）

スレッド数	1	2	4	8	12	16
πの値	3.1414	3.1410	3.1416	3.1422	3.1414	3.1414
実行時間（秒）	5.197	2.795	1.399	0.7016	0.4695	0.3542

表 9：実行結果（データ数 1 億 2 千万）

スレッド数	1	2	4	8	12	16
πの値	3.1414	3.1413	3.1414	3.1414	3.1411	3.1413
実行時間（秒）	51.974	28.753	14.7195	6.9876	4.659	3.4968

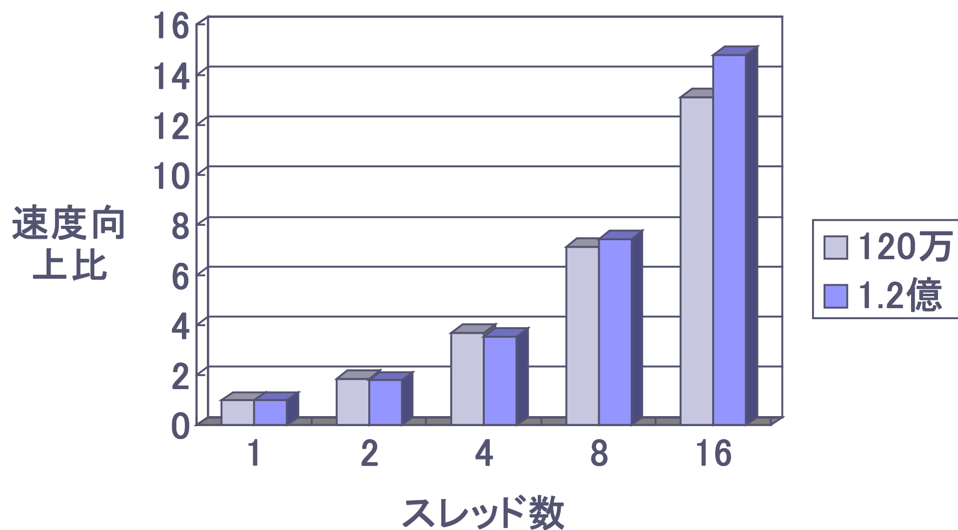


図 10：モンテカルロ法の速度向上比

4.2.3 考察

データ数が1．2億のときに14．9倍と理想に近い速度向上が得られた。精度の面からいうと、データ数を12億まで増やすと、3．1415までの精度は出る。しかし、精度を1桁出すのには100倍のループ回数が必要であるといわれることから、long int 型では約21億までが表現範囲なのでこれ以上の精度を出すのは難しいと考えられる。このモンテカルロ法では、乱数発生に rand 関数を用いたのだが、共有メモリマシン上ではうまく動作しなかった。共有メモリ上では、関数があるメモリ空間に複数のスレッドが同時に関数を呼び出そうとするので競合が起こり、うまく実行できないもではないかと考えられる。

5. OpenMP によるマンデルブローの計算

5.1 問題定義

マンデルブロー集合とは、$Z_0 = 0, Z_{n+1} = Z_n^2 - C$ で表される数列 Zn がnが無限大まで大きくなったときに発散しないような複素数 C の集合である。[11]

5.2 並列化手法

今回は、｜Zn|＞２に発散、ｎ＞３０のときに収束するという条件で計算をし、マンデルブロー集合に入るかを計算した。並列化の手法は、複素平面上の X 座標の範囲をブロック分割とサイクリック分割で実行し、速度向上の比較を行った。このプログラムは、SCASH のエラーにかかってうまく動作しなかったプログラムのデータ構造を変更して実行した。変更した部分は、マンデルブロー集合に入る点をカウントしていく変数を配列を使わずに１つの変数で定義し、その点のX座標とY座標の値を格納する配列を取り除いたところと、一番外側のループのイタレーションを for 構文を使って分割できるように工夫した点である。

5.3 実行結果

マンデルブロー集合を解像度を変更して実行した結果を表１０と表１１に示し、それぞれの速度向上比を図１０と図１１に示す。

表 10：実行結果（ブロック分割）

スレッド数	1	2	4	8	12
解像度 280×350	0.268	0.127	0.074	0.046	0.028
解像度 2800×3500	26.7	13.37	6.98	4.07	2.26

（単位：秒）

表 11：実行結果（サイクリック分割）

スレッド数	1	2	4	8	12
解像度 280×350	0.267	0.135	0.068	0.036	0.020
解像度 2800×3500	26.7	14.05	6.67	3.33	1.67

（単位：秒）

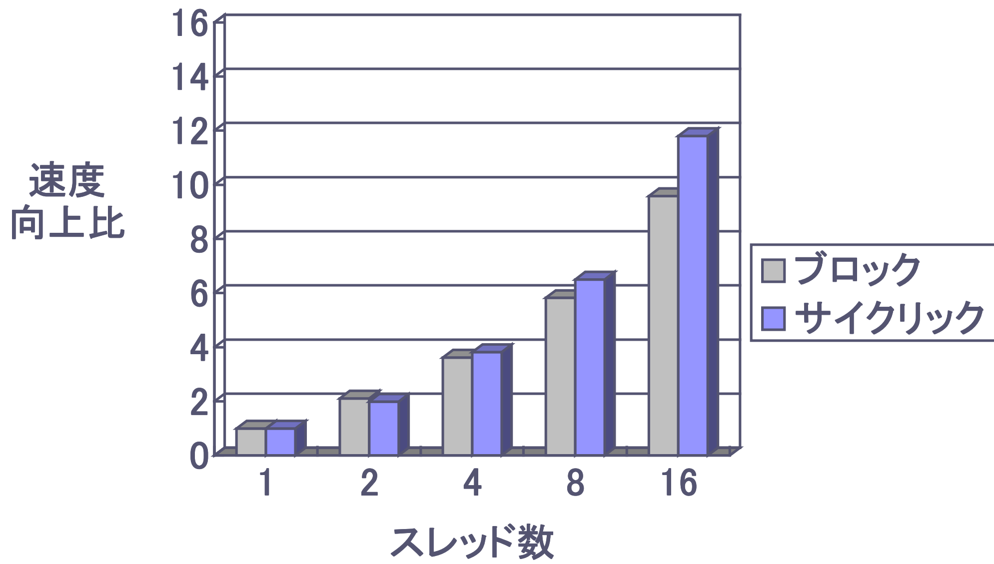


図 11：速度向上比（解像度２８０×３５０）

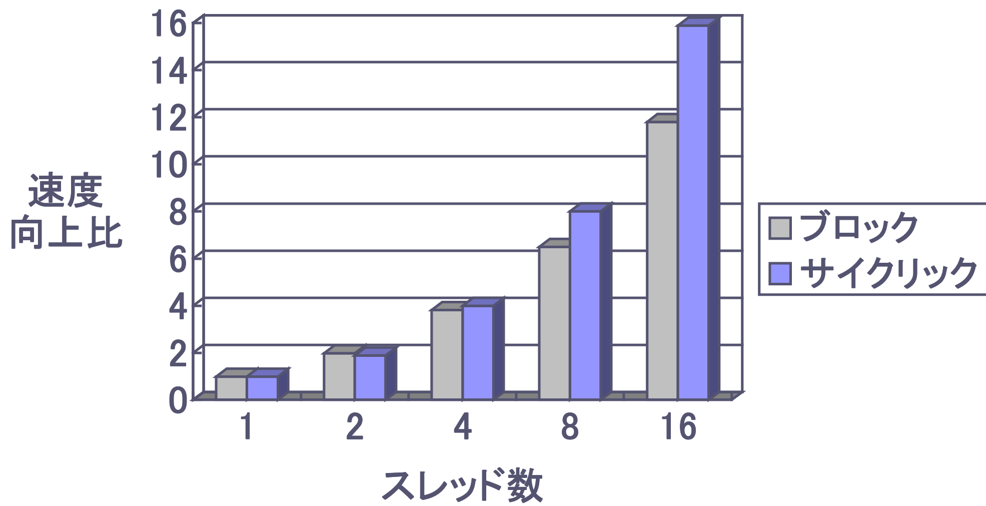


図 12：速度向上比（解像度２８００×３５００）

5.4 考察

ブロック分割での実行結果とサイクリック分割での実行結果を比較してみるとサイクリック分割のほうがスレッド数を増やすほど、実行時間も短くなり、速度向上もよいことがわかる。また、解像度を大きくすればするほど理想に近い速度向上が得られたことがわかる。マンデルブロー集合をX座標で分割するときに、大きな範囲に区切って分割すると発散せずに収束するまで計算されるZnが多く存在する範囲とそうでない範囲が存在するようになり、それを担当したスレッドは他のスレッドよりも多くの計算時間を要してしまい、そのためプログラム全体の実行時間も遅くなってしまう。サイクリック分割で細かい範囲で X 座標を分割することで負荷の均衡が取れて、ブロック分割よりも実行時間が短くなっている。

6. おわりに

本研究では、本研究室で構築した SCore 型 PC クラスタの動作テストと OpenMP による並列プログラミングを行ってきた。今までは、京都産業大学の SUN Enterprise 4CPU SMP マシンと本研究室の 2CPU SMP マシン上で OpenMP プログラミングを行っていたが、SCore 型の PC クラスタの構築を行ったことで、OpenMP が分散共有メモリ環境で正しく実行できるかのテストを行う必要があったので、SMP マシン上で動いている OpenMP プログラムを PC クラスタ上で動作させた。その結果、ソフトウェア分散共有メモリ SCASH の制約があることがわかった。ソフトウェアで共有メモリを実現させるために、どれだけのメモリ領域を必要としているのかを調べてみることも重要であると思われる。OpenMP のプログラミングでは、SMP マシン上で rand 関数を使うとうまく動作しないという問題点があったが、PC クラスタ上では rand 関数が使えるということがわかった。

今後の課題としては、sections 構文などを使ったタスク並列などの複雑な並列化ができる問題や、事例ベース並列プログラミングの中で OpenMP に適したプロセッサファーム型の問題である、hough 変換、レイトレーシングなどの画像処理関連のプログラムを OpenMP に移植し、PC クラスタで高速に処理できればよいと思われる。また、SCore のバージョンがアップされるとともに、OpenMP のコンパイラや、SCASH の性能も良くなっていけば、PC クラスタ上での OpenMP による並列プログラミングは今後ますます実用化されていくことと思われる。

謝辞

本研究の機会を与えてくださり、数々の助言を頂きました山崎勝弘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ご意見を頂きました本研究室の皆様、院生の松井さんに心より感謝いたします。

参考文献

[1]　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.

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

[3]　RWPC Omni OpenMP コンパイラプロジェクト:
http://phase.etl.go.jp/omni/home.ja.html

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

[5]　OpenMP C／C++ API 日本語版、RWPC、2000.

[6]　山崎勝弘：事例ベース並列プログラミングの研究　平成 9 年度～平成10年度科学研究費補助金　基盤研究(C)(2)研究成果報告書、1999.

[7]　森眞一郎、富田眞治：並列計算アーキテクトからみた計算機クラスタ、情報処理学会誌 Vol.39,No.11,pp.1073-1077,1998.

[8]　平木敬、丹羽純平、松本尚：分散共有メモリに基づく計算機クラスタ、情報処理学会誌 Vol.39,No.11,pp.1078-1083,1998.

[9]　長嶋雲兵、関口智嗣：大規模計算機におけるクラスタコンピューティングの可能性、情報処理学会誌、Vol.39、No.11,pp.1084-1088,1998.

[10]　石川裕、堀敦史、手塚宏史：RWCP におけるクラスタ開発記。情報処理学会誌 Vol.39,No.11,pp.1095-1100,1998.

[11]　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.

[12]　土谷悠輝：OpenMP による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、2000.

[13]　樋口道晴：PC クラスタ上での PVM 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、2001.

[14]　林晴比古：改訂新 C 言語入門（ビギナー編、シニア編）、ソフトバンク、1998

[15]　日本情報処理開発協会 ： 調査資料　情報先進国の情報化政策とわが国の情報技術開発における重点分野の選択指針.2000

[16]　三木光範　他：PCクラスタ超入門 2000、PCクラスタ型並列計算機の構築と利用、超並列計算研究会、2000.

付録

自作した OpenMP プログラミングのうち、モンテカルロ法によるπの計算と、マンデルブローの計算を行うプログラムを掲載する。

(1)モンテカルロ法

＜monte.c＞

```
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <stdlib.h>
#include <omp.h>
#include "second.c"

double second();
double   at, bt;

main()
{
   long c;
   long loop = 12000000;

   long cnti = 0;

   double lng = 0.0, sx = 0.0 , sy = 0.0;
   double X = 0.0;
   double pi = 0.0;

   bt = second();
     srand((unsigned int)time(NULL));



#pragma omp parallel for private(sx,sy,X) reduction(+:cnti)
     for(c = 0 ; c < loop ; c++)
```

```
        {
            sx = (double)rand()/RAND_MAX;
            sy = (double)rand()/RAND_MAX;

        X = sx * sx + sy * sy;

            if(X < 1.0)
              cnti ++;
        }



    pi   =(double)4*cnti/loop;

    printf("pi = %lf     ¥n", pi);

    at = second();



    printf(" time: %lf ¥n", at-bt);
```

(2)マンデルブロー

＜mandel.c＞

```
#include <stdio.h>
#include<omp.h>
#include "second.c"
double second();
double s1,s2;

main()
{
    int l,N,n;
    int count;
    float D,T;
    float RS,RE;   /*RS は複素平面の実部の始点、RE は実部の終点 */
    float IS,IE;   /*IS は複素平面の虚部の始点、IE は虚部の終点 */
```

```
s1 = second();

  D = 0.001 ;
  n = 0;
  RS = -2.2   ,RE = 0.6;
  IS = -2.0   ,IE = 1.5;
  T = RE - RS;
  n = T/D;

#pragma omp parallel shared(D,n)reduction(+:count)
 {
   int k,i;
  double A,B,x,X,y,Y;
#pragma omp for   /* サイクリック分割のときはここに schedule 指示節を入れる */
   for(i=0; i<=n;i++)
         {
          for(B=IS ;B<=IE; B+=D)
            {
              x=0.0;
              y=0.0;
             for(k=1;k<=30;k++)
               {
                 A = (i-(-RS/D))*D;
                 X = x*x - y*y + A;
                 Y = 2*x*y + B;
                if((X*X +Y*Y)>4.0) break;
                 x = X;
                 y = Y;
               }
             if((X*X + Y*Y)<=4.0)
               {
                count++;
               }
            }
         }
```

```
 }

/* ======OpenMP 構文終了====== */

  printf("count = %d¥n",count);

  s2 = second();
  printf("TIME = %lf¥n",s2-s1);
}
```