卒 業 論 文

PC クラスタ上での
並列プログラミング環境の構築

氏　　名　：　三浦　誉大
学 籍 番 号　：　2210980222-4
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

汎用のワークステーション・パーソナルコンピュータをネットワーク結合した PC クラスタが急速に広まっている。クラスタは、葡萄などの果実のようにコンピュータネットワーク上にぶら下がり、コンピュータが集団化することによって一連の統一した作業を処理するイメージから、クラスタリングもしくはクラスタシステムと呼ばれている。PC クラスタは、スーパーコンピュータ並の高速計算処理能力を有し、最先端情報処理技術に欠かせない重要な基盤技術となっており、今までスーパコンピュータを必要とした応用分野以外の新しい応用分野が開拓されている。

本論文では、RWC の開発したクラスタリングソフトウェアである SCore を使用して PC クラスタ上での並列プログラミング環境(分散メモリ環境・分散共有メモリ環境)の構築を行った。管理用 PC である Server Host 1 台と計算用 PC である Compute Host 16 台を 100Base-TX の Ethernet で、Compute Host 16 台を Myrinet で接続し、RedHat 7.1 上に SCore 4.2 をインストールした。

PVM と OpenMP の比較をサイクロイドで実行した。今回の結果では PVM の方が実行時間は短くすんだ。その後、Ethernet と Myrinet の比較を行った。Myrient の方が実行時間は短く、時間も一定で安定していた。OpenMP はプログラミングが容易である、Myrinet などの高速ネットワーク環境を構築できる点で有用である。

目次

図目次

表目次

1. はじめに

18 ヶ月に 2 倍という半導体集積度の上昇に従って性能向上し続けるマイクロ・エレクトロニクス技術は、あらゆる技術・システムに入り込み、技術体系全体を底上げすると同時に、システムの高度化・知能化をもたらしている。それは、PC の価格性能比の向上やネットワーク環境の充実をもたらし、Myrinet に代表される高速ネットワーク環境の構築を可能とした[1][16]。

大量の計算を必要とする分野（流体・構造解析、計算科学・分子設計、有限要素法解析、航空機設計・エンジニアリング、金融モデリング、自動車設計・エンジニアリング、長期気象予報、電子設計・分析など）での計算は逐次では膨大な時間がかかることが多く、これらの計算の処理を速く終えるためにも並列処理は欠かすことができない。計算機の性能向上よりも計算需要が上回っているため、今後もこの傾向は強まっていくことが予想される[9][16]。

並列処理とはプログラムの実行速度を上げる考えの 1 つを指し、プログラムを同時に実行可能な複数の部分に分け、それぞれをプロセッサ上で動かす。プログラムを N 個のプロセッサに分けて実行すれば、1 つのプロセッサで動かすのに比べて N 倍速く実行できると考えられる[2]。

アーキテクチャがプログラミングに影響を与える因子としては、メモリモデルが挙げられる。主なメモリモデルとして、次の 3 種類がある[5]。

- 分散メモリ：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できない。
- 共有メモリ：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。
- 分散共有メモリ：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できる。

汎用のワークステーション・パーソナルコンピュータをネットワーク結合したクラスタが急速に広まっている。米国 NASA のゴダード研究所で開発された Beowulf 型クラスタ、日本の RWC プロジェクト（現在は PC クラスタコンソーシアムに移行している）の中で開発された SCore システム、Windows NT ベースの PC クラスタなどがある[1][14]。

PC クラスタには以下のようなメリットがある。

- 高速な演算が可能: CPU の数に比例して実行速度が向上する。
- 非常に大規模な計算が可能:　1 台ではメモリ不足で解けないような計算が、並列化によって可能になる。
- 信頼性の向上: 複数の計算機で結果を照合できる。
- コストの削減:　レンタルでも、1 カ月で数千万～1 億円以上かかるスーパーコンピュータに比べると、非常に安価で構築できる。

分散メモリ型並列計算機の通信ライブラリは、従来、各メーカが、それぞれのハードウェアの性能を引き出すために提供したり、並列基本ソフトウェアベンダの独自仕様のものが用いられ、並列プログラムの移植性を損なっていた。このため、米国アルゴンヌ国立研究所を中心に、標準化の検討がされ、Parallel Virtual Machine（PVM)、Message Passing

Interface（MPI）の仕様が決められた[15]。

また、共有メモリ型並列計算機向けプログラムの標準化のために、並列計算機メーカー、並列化コンパイラベンダーが集まって、OpenMP という仕様を策定した。[15]

これまでの並列計算機では、ハードウェアごと並列化の方法が異なり、プログラミングが難しくなっていた。クラスタシステムでは、ノード間のデータのやり取りは通信プログラムとなるため、プログラミングが複雑になるという問題があった。前者に対しては、並列化の記述を容易にする OpenMP が注目されている。後者に対しては、メモリをソフト的に扱うことでユーザがノード間通信をプログラムする必要がないソフトウェア分散共有メモリシステムが有効となっている。RWC の SCASH を例にあげることができる[1][15]。

本研究においては、16 台の PC を Myrinet と Ethernet(100Base-TX)で接続し、分散メモリ環境と分散共有メモリ環境のクラスタシステムを構築することを目的とする。PVM で分散メモリ環境の構築と、RWCP の提供するクラスタソフトウェア SCore を利用して分散共有メモリ環境の構築を行う。その後、PVM と OpenMP、Myrinet と Ethernet で性能テストを行い考察する。

本論文では、2 章で PC クラスタと並列プログラミングについての概念について説明し、3 章で SCore を使用したクラスタ環境の構築方法、4 章で並列効果の考察と PC クラスタの評価を行う。

2. PC クラスタと並列プログラミング

2.1 PC クラスタ

葡萄などの果実のようにコンピュータがネットワーク上（葡萄の実を繋げている茎に相当する）にぶら下がり､コンピュータ（葡萄の実に相当する）が集団化することによって、一連の統一した作業を処理するイメージから、クラスタリングもしくはクラスタシステムと呼ばれている。クラスタシステムでは、同じ種類もしくは異なる種類の機能を持つコンピュータを 8 台から 1000 台動作させ、同一作業目的の処理を実行させることで、単体コンピュータでは限界だった信頼性や処理能力の大幅な向上を実現させるシステムを構築することが可能となる。

これは、単体のコンピュータに複数のプロセッサを用意すれば解決するシステムとは異なり、単体のコンピュータがそれぞれネットワークを介して、相互のコンピュータと協調するクラスタリング構成をとることを意味する。これにより、単体のマルチプロセッサシステムよりも高い処理能力や高い可用性を実現することが可能となる。

PC クラスタは、目的別に高可用性つまり HA（High Availability）を実現するものか・高性能つまり HPC（High Perfoemance Computing）を実現するものかで 2 種類に分かれる。

一般的にクラスタと呼ぶ場合にはこの HA クラスタを指し、Web サーバ等で利用されている。一例としては、ロードバランサ機を 2 台置き、その背後に Web サーバやデータベースサーバなどを設定した任意の台数のサービスノードを置く構成になる。これによりサービスを連続して提供できる高い可用性(High Avairability)を実現している。

これに対して、並列処理でのクラスタは HPC クラスタと呼ばれ、大規模分散処理を主目的としたクラスタとなっている。

2.1.1 Beowulf 型クラスタ

1994 年に NASA の Earth and space science プロジェクトのために CESDIS（Center of Excellence in Space Data and Information Sciences）で開発された。Linux をベースとした 16 ノードクラスタであり、10Mbps の Ethernet （TCP/IP）を相互結合網として使用した。並列化は PVM（Parallel Virtual Machine）と MPI(Message Passing Interface）を用いている。Beowulf クラスクラスターの特徴は、市販の PC を使用して、公開されたソフトウェア（Linux）を使うことで、簡単に高性能のクラスタが構築できることである。

2.1.2 SCore 型クラスタ

SCore は RWC の開発したクラスタリング環境を提供するソフトウェアである。

Beowulf のように TCP/IP を用いると、何階層ものプロトコル変換が行われるために、応用プログラムの多様な資源割り当て要求に対応することが難しい。そこでオー

バーヘッドを減らすために、ユーザレベルから直接ネットワークハードウェアが制御できる zero-copy 通信ソフトウェアの開発が課題となっていた。SCore は、そういった Beowulf 型クラスタの問題点を改善している。

SCore の特徴は以下の通りである[1]。

・高性能通信環境

PM II 高性能通信ライブラリ。

・シングルシステムイメージ

効率よいコンピュータ管理が可能。

(シングルユーザモード・マルチユーザモード・バッチモード)

・シームレスクラスタ群環境

ネットワークの種類を気にすることなくアプリケーションを実行することが可能。

・ハイアベイラビリティ

チェックポイント・リスタート機能：実行中のアプリケーションイメージをディスクに格納し、システムの停止再起動後、格納したイメージをメモリにロードして再開することが可能。

マイグレーション機能：実行中のアプリケーションを別のクラスタ上に移動させることが可能

・インストレーションツール

インストールを簡単にするための Easy Installation Tool が備わっている。

SCore のソフトウェア階層を図 1 に示す。

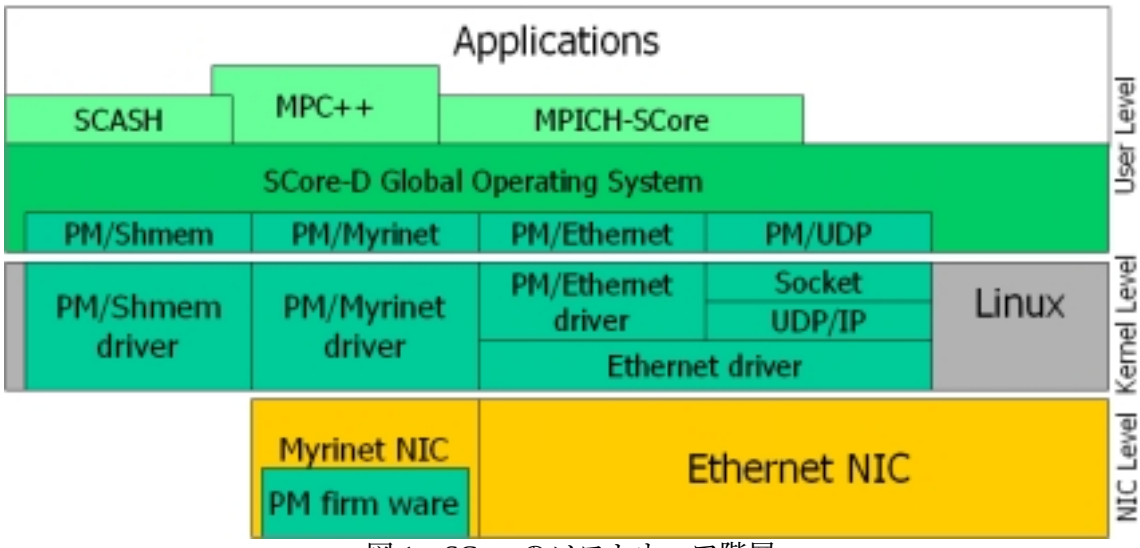


図 1　SCore のソフトウェア階層

Beowulf 型クラスタと SCore 型クラスタの違いを表 1 に示す。

表 1　Beowulf 型と SCore 型の違い

	Beowulf 型	SCore 型
特徴	代表的な既存ソフトの組み合わせでシステム構築	PC クラスタシステムソフトウェアを提供
通信層	TCP プロトコル	UDP プロトコル
運用管理	PBS（Portable Batch System） DQS（Distributed Queueing System）	PBS（Portable Batch System） SCore-D グローバルオペレーティングシステム

Beowulf 型と SCore 型の違いは通信層の違いといえる。

TCP（Transmission Control Protocol）は送受信ホストのプロセス間に信頼性のあるコネクションを確立する。個々のブロック送信は 1 つのメッセージと見なされず、送受信間のコネクションを通して伝送される、一連のストリームの中のデータブロックとして扱われる。個々のブロックは単にストリームの一部に過ぎず、ユーザ自身が書いたコードによってこのストリームからブロックを取り出す必要がある。しかし、メッセージを分解するための印がないために、コネクションはネットワークの障害に弱く、それぞれのプロセス当たりで同時にコネクションを張る数にも制限がある。信頼性があるために、TCP は UDP と比べてオーバーヘッドが大きくなる傾向がある。

これに対して UDP（User Datagram Protocol）はデータをアプリケーション間に送受信コネクションレス通信で、信頼性はアプリケーションに委ねている。UDP は送信するブロックを独立したメッセージとしていて、そのメッセージは伝送中になくなる可能性がある。しかし、UDP はメッセージが到着していればその中身は壊れていないことを保証しており、1 つの UDP メッセージの断片だけを受け取ることはない。また、UDP はソケットプロトコルの中では最速である場合が多い。

2.2 並列プログラミング

現在、並列プログラミングを記述する方法として、(1) 従来型のプログラミング言語にコンパイラ指示文を挿入する方法、(2) 従来型のプログラミング言語において標準的通信ライブラリ（PVM, MPI など)を用いてプロセッサ間のデータ受渡し・同期を行う方法、(3) 並列処理向けの様々な言語を用いる方法などがある。(1) は既存のプログラムを大きく変更せずに並列化ができるというメリットがあるが、主にループレベルの並列化に留ま

り、小規模並列マシン向けの手法となっている。(2) は大規模なメッシュを領域分割して、それぞれの領域をプロセッサに割り当てる並列化手法などで用いられ、分散メモリ型の大規模並列機の性能を引き出すことができる。しかしながら、プログラム設計が難しく、また正しく動作することの検証が容易でない。(3) は並列計算モデルを反映した言語を用いてプログラミングする手法であり、プログラム論理の正しい並列動作が処理系によって保証されているが、並列処理の細かい部分をプログラマが指示できないため、現状では動作するプログラムの性能が不十分である。今後とも、それぞれのプログラミング方法の改良が必要である。[16]

2.2.1 分散メモリ並列プログラミング

(1) HPF(High Performance Fortran)

Fortran をベースとした並列言語は、Fortran D、Vienna Fortran、PCF-Fortran、CM-Fortran、DAP-Fortran などさまざまなものが提案されてきた。そういったものを統一しようと、Super Computing '91 で HPF の規格原案が生まれた。

HPF 2.0 は Fortran 95 に基づいており、!HPF$で始まる指示文を挿入することにより並列化が出来るようになっている。HPF は分散メモリ用のプログラム言語であるが、アーキテクチャに依存しない仕様となっているため、分散メモリでも共有メモリでも使用可能となっている。

表 2　HPF の代表的な指示文

宣言指示文		実行指示文	
PROCESSOR	プロセッサの宣言	INDEPENDENT	DO ループなどの独立実行
ALIGN	データの整列	REALIGN	データの動的な整列
DISTRIBUTE	データの分散	REDISTRIBUTE	データの動的な分散
INHERIT	データマッピングの継承	ON BEGIN	計算の分散指示
DYNAMIC	動的なマッピングの許可	RESIDENT	データの局地製の保証
SHADOW	通信バッファ領域の宣言	TASK REGION	タスク並列の支持

(2) PVM (Parallel Virtual Machine)

PVM (Parallel Virtual Machine) はフリーで利用ができ、移植性があるメッセージ通信ライブラリで、大部分はソケットベースで実装してある。

PVM がサポートしているのは、プロセッサが 1 つのマシンや SMP Linux マシン、ソケットが利用可能なネットワーク (例えば SLIP、PLIP、イーサネット、ATM)に接続している Linux マシンによるクラスタである。PVM は、プロセッサやシステム構成、使用している物理的なネットワークが異なっているさまざまなマシン構成 (異機種間クラスタ) であっても実際に動作する。また、PVM はクラスタ全体に渡って

並列に実行しているジョブを制御する機能も持っている。そして、何よりも PVM は長い年月に渡り何の制限も受けずに利用されているため、結果として数多くのプログラミング言語やコンパイラ、アプリケーション・ライブラリ、そしてプログラミングやデバックのためのツール等がある。そしてこれらの成果物を「移植性のあるメッセージ通信を開発するためのライブラリ」として使用している。

しかし、PVM のメッセージ通信を呼び出すと標準的なソケット処理の遅延の大きさに加えて、さらにかなりのオーバーヘッドが加わってしまうことに注意が必要である。その上、メッセージを扱う呼び出しそのものがとりわけ「親しみやすい」プログラミング・モデルではないことにも注意が必要である。

表 3　PVM の代表的な通信関数

pvm_send（送信先 ID、メッセージタグ）	送信(1 対 1 通信)
pvm_recv（受信先 ID、メッセージタグ）	受信(1 対 1 通信)
pvm_reduce(通信と同時に行う演算を定義、送信バッファの返しアドレス、データの要素数（int)、データタイプ、グループ名、演算結果を知るプロセス番号)	リダクション (1 対多通信)

（3）MPI（Message Passing Interface）

MPI（Message Passing Interface）は規格であり、PVM のようなライブラリではない。現在、分散メモリ型のプログラミングインタフェースとして、最も標準的に用いられている。仕様は MPIF（The MPI Forum）で検討され、1994 年 6 月に MPI1.0 がまとめられた。その後、並列 I/O やプロセスの動的生成・消滅、1 方向通信（One Sided Communication）などの機能をさらに追加した MPI2.0 の言語仕様が 1997 年 7 月に定められた。

日本では MPICH（MIP CHameleon）が一番普及している。MPICH は MPI1.1 に完全に準拠しており、移植性を考慮して設計してある。LAM と同様に、MPI プログラムがスタンドアローンの Linux システムや UDP や TCP ベースのソケット通信を使った Linux システムで構築したクラスタ上で動作する。しかし MPI が効率的かつ目的に対して柔軟に対応することに重点をおいているのは間違いない。この MPI の実装を移植するには、「チャネル・インターフェース」の 5 つの関数とパフォーマンス向上のために MPICH ADI（Abstract Device Interface）を実装することになる。

表 4　MPI の代表的な通信関数

MPI_Send （送信バッファの開始アドレス、データの要素数(int)、データタイプ、受信先(int)、メッセージタグ(int)、コミュニケータ)	送信(１対１通信)
MPI_Recv （受信バッファの開始アドレス、データの要素数(int)、データタイプ、送信先(int)、メッセージタグ(int)、コミュニケータ)	受信(１対１通信)
MPI_Reduce （送信先の送信バッファの開始アドレス、受信先の受信バッファの開始アドレス、データタイプ、演算のハンドルコミュニケータ)	リダクション (１対多通信)

2.2.2 共有メモリ並列プログラミング

(1) スレッド

スレッド並列化記述インタフェースとは、複数の手続きを別々のスレッドで非同期に実行することにより並列処理を実現する枠組みである。並列化制御ライブラリを用いた Fork ～ Join のプログラミングスタイルとなり、共有メモリマシン用のインタフェースである。ある意味では、データ転送を共有メモリアクセスと同期制御で行うメッセージパッシングととらえることもできる。Unix 上では、pthread ライブラリがこのインタフェースの代表である。また、CRAY や SX などの共有メモリベクトル並列マシンでも「マクロタスキング」として、同様の機能がサポートされている。

(2) OpenMP

OpenMP は、1997 年にアメリカの OpenMP Architecture Review Board が、Fortran をベースとして API の仕様で開発したものである。以後、C/C++などにおいても API の仕様で開発された。特徴として、Fortran/C/C++などをベースに、ループ、タスクの並列化部分に指示文を挿入することで、並列プログラムを作成できる。これにより逐次プログラムを並列プログラムを同じソースで管理できる。分散されたメモリを 1 つの共有メモリとして扱えるため、データの受け渡しを考慮しなくて良い。現在、共有メモリ型並列計算機は、広範囲に広まっている。そのため、これらのシステムで容易にプログラムの移植が可能な方法の 1 つである[1][4]。

次の図 2 は OpenMP プログラミングの実行モデルの fork-join モデルである。

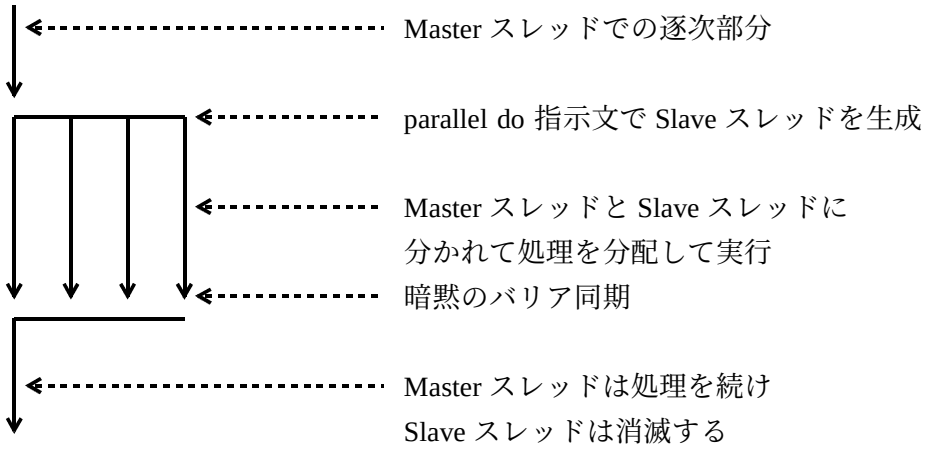


図 2　fork-join モデル

3. PC クラスタの構築

PC クラスタで分散メモリ環境を使用できるように PVM を、分散共有メモリ環境を使用できるように OpenMP をサポートしているクラスタソフトウェア SCore を使用する。

3.1 クラスタ構築の推移

始めに、Kondara Linux 1.1 でクラスタを組んだ。PC8 台を Ethernet と Myrinet で接続し、その上に SCore3.3.2 をインストールした。これらはカーネルの再構築までする必要があったために構築し終えるまでに時間がかかった。

そこで次に、その時点のバージョンでは Easy Installation Tool を使用することのできた Turbo Linux 6.1 でクラスタを組んだ。バイナリインストールされていたこともあって、カーネルの再構築の手間もはぶけたことに加え、Compute Host に OS のインストールの手間がはぶけ、インストール時の手間を大幅に短縮することができた。

しかし、動作が不安定でプログラムの実行時間がまちまちであったので、クラスタのバージョンを最新のものに変えることにした。SCore4.2 では RedHat のみ対応との事だったので、RedHat でクラスタを構築した。PC8 台で組み、うまくいった後、16 台で組むことにした。SCore に PVM は入っていたが、不安定ということを聞き、PVM をインストールした。

現在の PC クラスタの構築は以下の手順で行った。

・PC を Ethernet と Myrinet で接続。
・SCore をインストール。
・PVM をインストール。
・システムテスト

なお、Server Host を Computer Host のように動作させることも可能である。Server Host は制御のみを行い、Computer Host は計算のみを行っている。Computer Host うちの 1 台に Server Host の役割をかぶせることはシステムの不安定につながるので、Server Host と Computer Host は分けている。また、スレッド間のデータ通信は Myrinet と Ethernet の両方で行うことが出来るが、スレッドの制御に使っているのは Ethernet のみである。

3.2 システム構成

研究室の PC クラスタは図 3 のように接続されている。

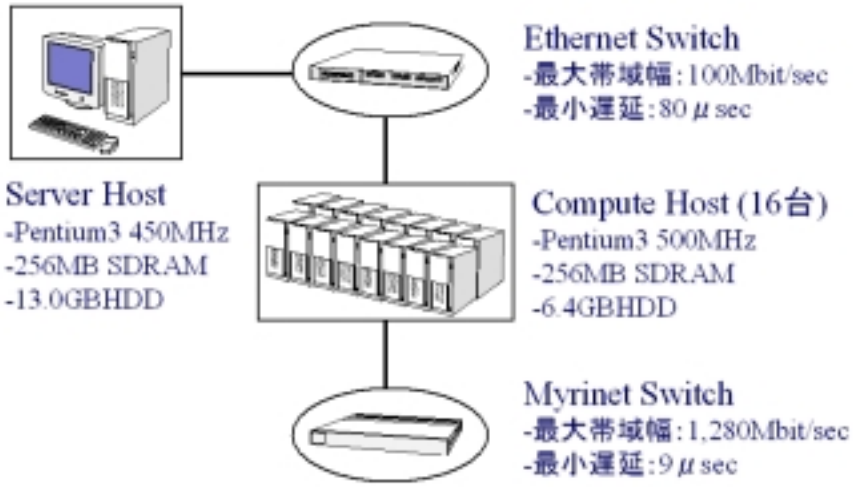


図 3　PC クラスタの構成

Compute Host として 16 台、それらを管理する Server Host として 1 台設置する。Ethernet で Server と Cluster16 台を、Myrinet で Cluster16 台を接続する。

3.3 Linux の移植

OS には RedHat　Linux　7.1 を使用する。3.3 で後述するので端折るが、RedHat をインストールするのは Server　Host だけでよい。モードは expert を選択する。Kondara8 台でクラスタを組んだときには必要最低限と思われるパッケージを選択できたが、それと同様に選択したところうまくいかず、何度か追加もしくはやり直す状態になった。最終的にパッケージは Everything ですべてインストールした。注意する点はファイアウォールの設定を NoFirewall を選択する点である。

インストール後に以下の基本設定を行った。

・IP
・マウント
・DNS
・シェル
・ドメインネーム
・サービス
・スイッチ

RedHat ではデフォルトではすべてのサービスが off になっており、Kondara が大抵の必要なサービスが on になっているのと対称的である。そのため、必要なサービスを on にし

ておく必要がある。作業を終了したらリブートする。

3.4 クラスタシステムソフトウェア SCore の移植

今回使用した SCore のバージョン(4.2)から RWC SCore Easy Installation Tool（EIT）が標準で入っているのでこれを利用する。これを利用すると、インストールにかかる作業が大幅に短縮される。

RWC の HomePage から SCore を落として CD に焼く。その後、CD をマウントし、インストールを行う。

```
# mount /mnt/cdrom
# cd /mnt/cdrom/
# ./Install
```

実行後、/opt/score　ディレクトリが生成され、ドキュメントと EIT がインストールされる。次に EIT を起動する。EIT がインストール用のブートフロッピーを生成するので、それを利用して Compute Host にインストールする。

```
# /opt/score/bin/eit
```

EIT の表示に従い情報を入力する。Compute Host Installation の"Make Boot Image" で Compute Host のブートフロッピーをを作成する。ブートフロッピーを Compute Host に入れて電源を投入してしばらく経つとアラームが鳴り出すので、フロッピーを取り出す。Server のディスプレイにアナコンダのウィンドウ画面が出るので"Next"を押すとアラームが 5 回なり、インストールが開始される。このとき、Compute　Host の画面を表示しなくても良いが、表示すると RedHat と必要な SCore のファイルがコピーされているのがわかる。アナコンダのウィンドウはすべての Compute　Host がインストールを終えるまでは消さないほうが良い。すべての Compute　Host のインストールが終わったのを確認して、すべてのウィンドウを閉じてログアウトし、ログインしなおす。

SCore 自体のインストールは終了しているわけだが、home のマウント先が RedHat ではなく、Solaris を利用している関係上シェルの位置等の違いがある。そこで Server　Host と同様の基本設定を Compute Host で設定する。

3.5 システムテスト

クラスタが正常に動作させ、また、それを確認するためにシステムテストを行う。するテストは以下の 4 つである。

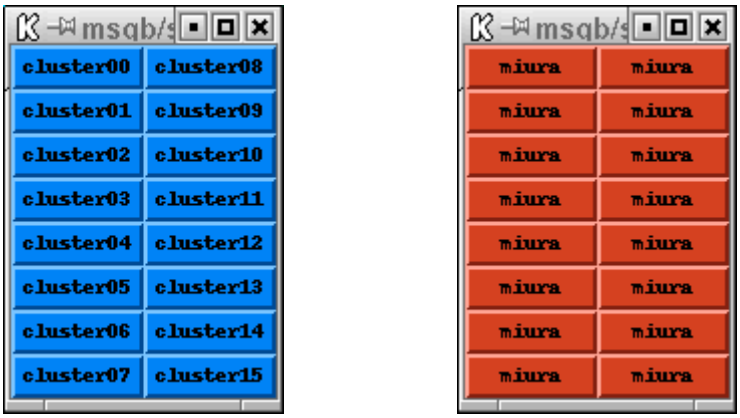
(1) SCOUT テスト

- 環境変数 SCBDSERV・PATH を設定
- クラスタデータベースの作成
- ComputeHost の確認
- SCOUT コマンドの実行

・Single-User 環境でのサンプルプログラムの実行

SCore 用シェルの動作の設定と確認をするテストである。SCOUT は、Compute Host の状態を監視するメッセージボード(Compute Host Lock Client)上で、これから動かそうとするノードグループが動作するか確認するときのコマンドである。これはそれが正常に動作するためのテストである。

メッセージボードの起動時の画面を図 4(左)に示す。図 4(右)はノードが専有されており、"miura"が使用していることを示している。



msgb 通常時（青）　　msgb 実行時（赤）

図 4　Computer Host Lock Client

(2) PM/Myrinet テスト

・ループバックテスト(ping)

・Poing-to-Point テスト(メッセージ・ゼロコピー)

・総合テスト(ストレステスト)

Server Host と Compute Host 間の通信を行う PM II 通信ライブラリが Myrinet を通して正常に動作するか確認するテストである。

(3) PM/Ethernet テスト

・etherpmctl コマンドの動作確認

・Poing-to-Point テスト(メッセージ)

・総合テスト(ストレステスト)

Server Host と Compute Host 間の通信を行う PM II 通信ライブラリが Ethernet を通して正常に動作するか確認するテストである。

(4) SCore テスト

・環境変数 SCBDSERV・PATH を設定

・Compute Host Lock Client の作動

・Single-User 環境でサンプルコマンドを実行

・MPC++ MTTL プログラムのコンパイルと実行

・MPICH-SCore プログラムのコンパイルと実行

・Single-User 環境の終了
・Multi-User 環境のために SCore-D operating system を起動
・Multi-User 環境下で MPC++ MTTL プログラムを実行
・Multi-User 環境で MPICH-SCore プログラムを実行
・Multi-User 環境の停止
・SCOOP GUI tool を使用して PC cluster を監視

SCore が正常に動くか確認する総合テストであり、実際にプログラムを動作させるときのコマンドを一通り確認するテストである。

(5) デモンストレーション

・マンデルブロープログラムの表示

このデモンストレーションはテストとは直接関係ないが、スレッド数の変化で計算速度の違いが目に見えて体感できる。このマンデルブロープログラムは、表示されたウィンドウを拡大縮小したりマウスでリサイズすることにより、画像を矩形分割して、それぞれの領域を指定されたスレッド数で計算を行う。

図 5(左)は実行した直後のスナップショットである。それを数回リサイズした後の状態が図 5(右)である。

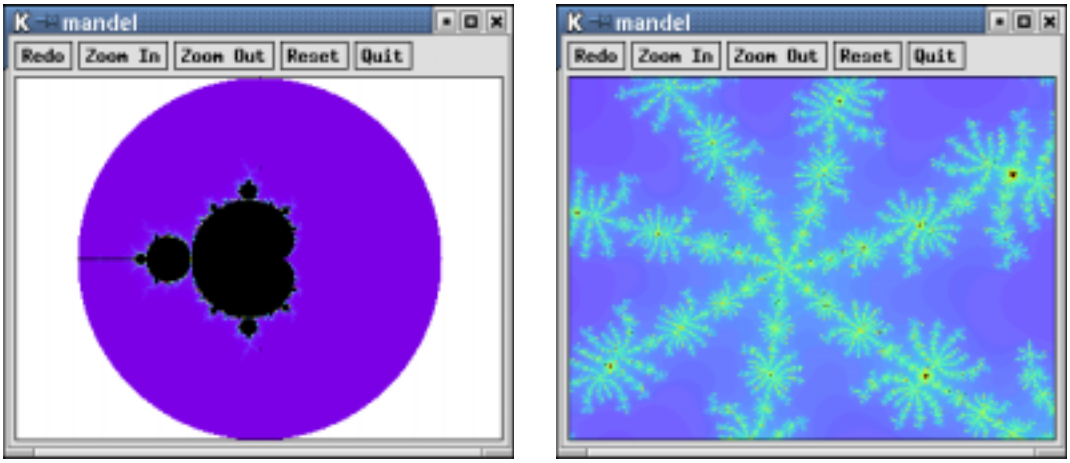


図 5　マンデルブロー

4. 並列プログラミング

4.1 サイクロイド

4.1.1 問題定義

PVM と OpenMP の比較には「サイクロイド」を取り上げる。「サイクロイド」とは円を転がしたときの、円の一点が描く軌跡を表したものの事で、今回取り上げたのは、直線上で転がしたときの直線とサイクロイドの間の面積を求めることにした。

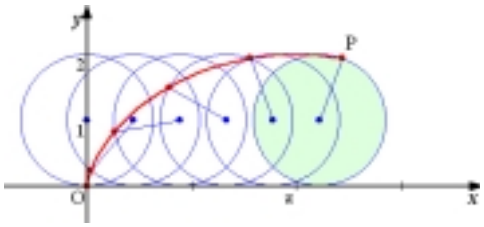


図 6　サイクロイド

サイクロイドの公式は以下の式で表されるが、これは時間(t)の経過とともにxとyの値が変化していくことを意味している。

$$A=\int_0^1 y\,dx=\int_0^{2\pi}(1-\cos t)(1-\cos t)\,dt=\left[\frac{3}{2}t-2\sin t+\frac{1}{4}\sin 2t\right]_{t_1}^{t_2}$$

$x=t-\sin t,\ y=1-\cos t\ \ (0\leqq x\leqq 2\pi,\ 0\leqq y\leqq 1)$

4.1.2 分割方法

面積を求めるときにはｘの値は均等に分断する必要がある。そこで 0 ≦ x ≦ 2 πの区間を N で分割する。

分割方法はブロック分割を用いた。PVM ではスレーブにて x ＝ 2 π/N ごとの面積を算出し、それをマスターに集めて回収する。

4.1.3 実行結果

PVM と OpenMP でそれぞれ分割数 1000 万、5000 万、1 億、5 億で実行した。

この実験の結果は表 5 ～表 8、図 7 に示す。なお、OpenMP の実行には Ethernet と Myrinet を使用したが、サイクロイドのプログラムが小規模すぎたためか、実行時間に差が出なかった。したがって実行時間と速度向上比は 1 つしかのせていない。

表 5　PVM 実行時間(sec)

分割数＼ Slave	1	2	4	8	16
1000 万	9.80	5.06	2.73	1.52	0.70
5000 万	48.52	24.43	12.42	6.38	3.54
1 億	96.91	48.64	24.50	12.62	6.53
5 億	484.16	242.25	121.38	61.91	32.24

表 6　PVM 速度向上比

分割数＼ Slave	1	2	4	8	16
1000 万	1	1.93	3.58	6.44	14.00
5000 万	1	1.98	3.90	7.60	13.70
1 億	1	1.99	3.95	7.68	14.84
5 億	1	1.99	3.98	7.82	15.01

表 7　OpenMP 実行時間(sec)

分割数＼ Thread	1	2	4	8	16
1000 万	12.02	6.02	3.01	1.49	0.73
5000 万	60.10	30.78	15.77	7.42	3.64
1 億	120.19	60.81	30 79	15.57	7.28
5 億	600.97	301.17	150.95	74.88	37.12

表 8　OpenMP 速度向上比

分割数＼ Thread	1	2	4	8	16
1000 万	1	2.00	3.99	8.07	16.47
5000 万	1	1.95	3.81	8.10	16.51
1 億	1	1.98	3.90	7.72	16.51
5 億	1	2.00	3.98	8.03	16.19

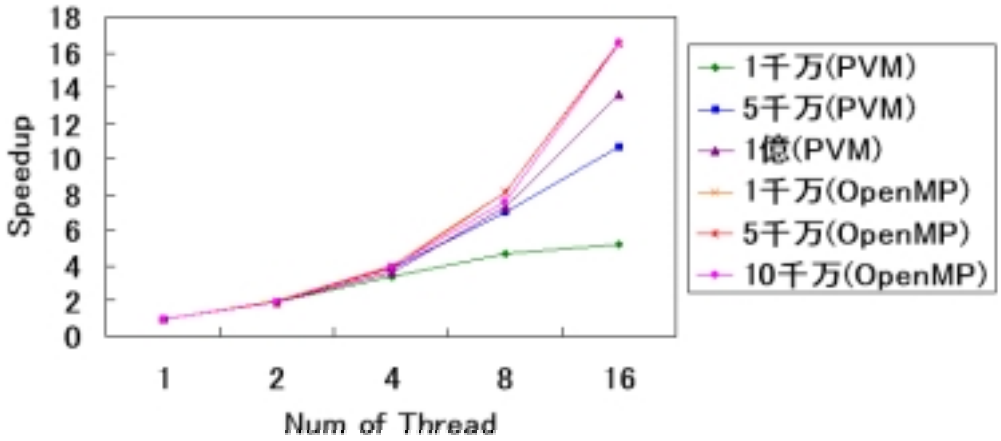


図 7　サイクロイド速度向上比

このアルゴリズムでは PVM の方が実行時間が短いという結果が出たが、これは小規模の OpenMP のプログラムがロックとアンロックを使用しているために、PVM のプログラムよりも処理に手間取ったためだと思われる。PVM と OpenMP の比較事例はまだ少ないため、より規模の多いもので比較することが今後の課題となる。

4.2 既存の OpenMP プログラミング

4.2.1 問題定義

当研究室で作成された既存の OpenMP プログラムの中で動作した 4 つのプログラムを Myrinet と Ethernet で比較する。比較する OpenMP プログラムにはラグランジェ補間、ランレングス圧縮、文字列照合(KMP 法、BM 法)を用いた。

ラグランジェ補間は何組かの x、y データが与えられているとき、これらの点を通る補間多項式をラグランジェ補間により求め、データ以外の点を求めるという問題である[18]。

ランレングス圧縮法はランレングス符号化とも呼び、符号化法のひとつである。ファイルの中で同じ文字が繰り返される連は最も単純な冗長性である。連が長い場合はランレングス圧縮により、高い圧縮率を持つことになる[19]。

KMP 法は D. E. Knuth 、J. H. Morris 、V. R. Pratt による文字列照合アルゴリズムを指し、BM 法は R. S. Boyer 、J. S. Moore による文字列照合アルゴリズムのことを指す[19]。

4.2.2 実行結果

ラグランジェ補間、ランレングス圧縮、KMP 法、BM 法の 4 つのプログラムを PVM、Myeinet(Ethernet/Myrinet)で動作させた。ラグランジェ補間は座標数 4000 と 20000 で実行した。ランレングス圧縮は比較的長い連のアルファベットで構成された 1 万、10 万、100 万文字のテキストファイルを作成し、それを圧縮させた。KMP 法と BM 法は短い連のアルファベットで構成された 1 万、10 万、100 万文字のテキストファイルを作成し、照合文字列を 7 文字として実行した。

この実験の結果は表 9 ～表 16、図 8 ～図 11 に示す。

表 9　ラグランジェ補間/Myrinet 実行時間(millisec)

座標数＼ Thread	1	2	4	8	16
4000	46.0	24.2	13.2	8.4	7.3
20000	230.4	116.3	59.3	31.5	18.9

表 10　ラグランジェ補間/Ethernet 実行時間(millisec)

座標数＼ Thread	1	2	4	8	16
4000	46.1	34.8	39.8	49.1	76.9
10000	230.4	130.4	80.5	70.8	86.7

表 11　ランレングス圧縮/Myrinet 実行時間(millisec)

文字数＼Thread	1	2	4	8	16
1 万	0.97	1.79	1.94	2.76	4.24
10 万	9.9	8.5	5.6	4.6	6.3
100 万	103	74	39	23	23

表 12　ランレングス圧縮/Ethernet 実行時間(millisec)

文字数＼Thread	1	2	4	8	16
1 万	0.97	13.68	18.74	30.83	52.62
10 万	9.9	24.2	31.8	36.8	54.8
100 万	102	163	132	135	445

表 13　KMP 法/Myrinet 実行時間(millisec)

文字数＼Thread	1	2	4	8	16
1 万	0.97	2.26	2.49	3.10	4.72
10 万	9.4	8.1	5.3	4.8	6.4
100 万	101	62	43	33	30

表 14　KMP 法/Ethernet 実行時間(millisec)

文字数＼Thread	1	2	4	8	16
1 万	0.97	13.6	18.7	30.8	52.6
10 万	9.4	33.6	38.3	55.8	64.8
100 万	101	120	1338	1798	1988

表 15　BM 法/Myrinet 実行時間(millisec)

文字数＼Thread	1	2	4	8	16
1 万	0.52	1.78	2.14	2.72	3.96
10 万	4.7	5.7	4.0	4.0	5.5
100 万	55	47	36	31	28

表 16　BM 法/Ethernet 実行時間(millisec)

文字数＼Thread	1	2	4	8	16
1 万	0.52	11.42	21.06	31.48	53.73
10 万	5.0	25.4	33.3	39.9	58.0
100 万	55	127	120	1355	2612

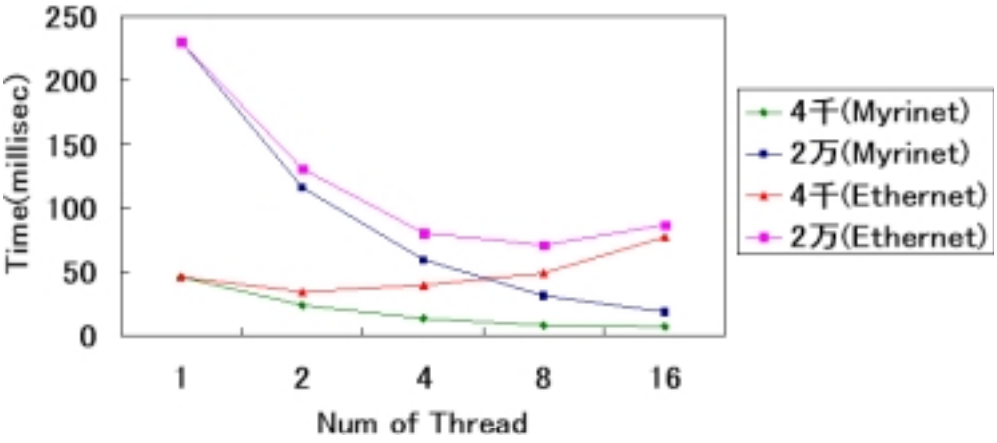


図 8　ラグランジェ補間 実行時間

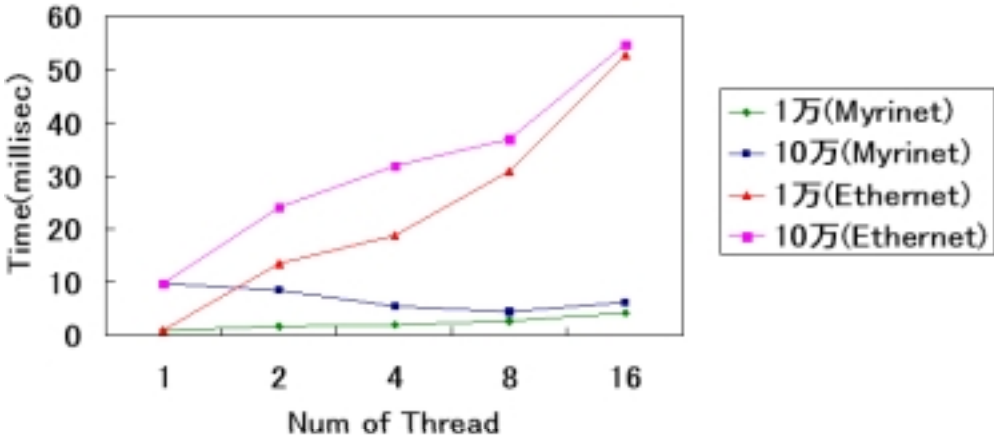


図 9　ランレングス圧縮 実行時間

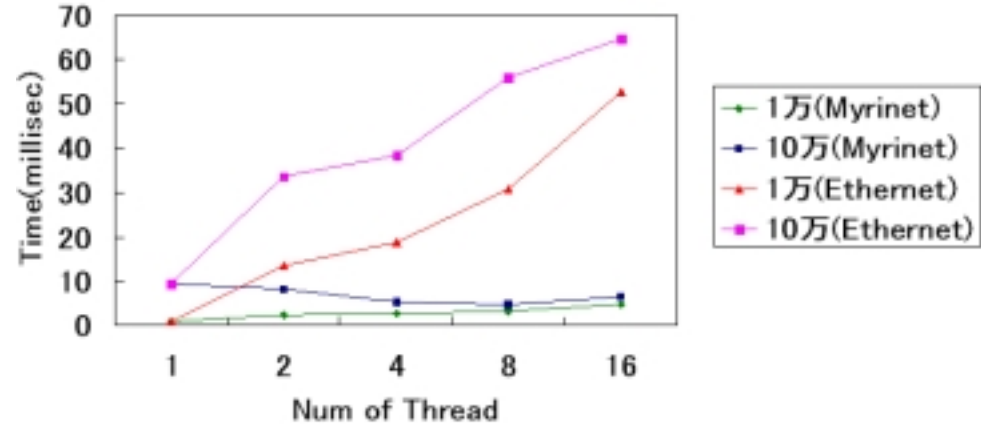


図 10　KMP 法 実行時間

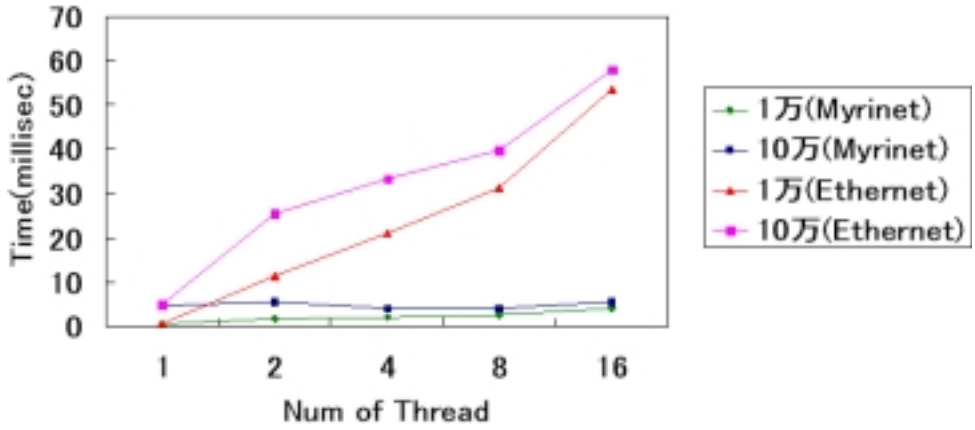


図 11　BM 法 実行時間

Myrinet と Ethernet の比較では、ラグランジェ補間、ランレングス圧縮、KMP 法、BM 法の 4 つとも Ethernet よりも Myrinet の方が実行時間は短かかった。サイクロイドで差が出なかったことを考慮すると、規模が大きいプログラミングになればなるほど通信には Myrinet を使用した方がいいことがわかる。

Ethernet では並列効果はラグランジェ補間を除きほとんど出なかった。これはオーバーヘッドがかかり過ぎているためと推測できる。

データは複数回取り、平均を載せてあるが、Myrinet のデータはばらつきが少なかったが、Ethernet のデータにはばらつきが目立った。

これらのことから、OpenMP プログラミングでは Myrinet を使用した方が効率のいいことがわかる。

5. おわりに

本研究では、16 台の PC クラスタの構築を行った。16 台を接続して認識させた段階では分散メモリ型並列計算機だが、SCore 等のクラスタソフトウェアを利用することにより、分散共有メモリ方並列計算機を構築することができた。私自身の Linux や Solaris、それに並列プログラミングに対する知識不足もあり、遅々とした進み具合であったように思う。思ったよりも手間取ったのは、Linux、特に Kondara や Turbo や RedHat と Solaris の違いであった。home ディレクトリを Solaris からマウントしているために起こるエラーの対処が一番大変だった用に思う。クラスタをネットワークから切り離して構築すればもっとすんなり進んだのかもしれないが、ユーザビリティの面からも避けたかった。

今後の課題として 3 つあげることができる。まず 1 つ目はクラスタのノード数を増やしての構築である。クラスタの構築は台数を増やすごとに構築が難しくなるといわれており、それはノード数が増えれば増えるほど管理が大変になることと、スイッチングハブの設定が複雑になるためである。2 つ目は、より大規模な問題での PVM と OpenMP の比較である。今回サイクロイドの比較では PVM の方が実行時間が短いという結果になったが、より大規模な問題ではどうなるのかを調べたい。3 つ目は画像処理などの大規模な問題を OpenMP で実行したい。OpenMP は従来の並列プログラミング言語に比べて簡単であること、SMP アーキテクチャに対応している点が有用であり、今後主流になっていくと思われる。実際に視覚を用いて確認することが出来る画像処理などの問題は並列効果を体感するのにもわかりやすく、大学院ではこれらに挑戦したいと考えている。

謝辞

本研究の機会を与えてくださり、日頃ご指導頂きました山崎勝弘教授に心より感謝を致します。また、本研究にあたり、さまざまな面で貴重なご意見や、励ましを頂きました Tran Cong So 様、松井 誠二様、大村 浩文様、そして本研究室の皆様にも心より感謝致します。

さらに、クラスタ構築にあたって数々の助言を頂きました住商エレクトロニクスの柴崎 勝憲様、RWC の石川 裕様、SCore-users のメーリングリストの方々に心より深く感謝します。

参考文献

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

[2] Linux Parallel Processing HOWTO: http://www.linux.or.jp/JF/JFdocs/Parallel-Processing-HOWTO.html

[3] R. Chandra, L. Fagum, D. Kohr, D. Maydan, J. McDonald, R. Menon:"Parallel Programming in OpenMP", MORGAN KAUFMANN PUBLISHERS, 2000.

[4] 飯塚 肇, 緑川 博子 共訳:"並列プログラミング入門", 丸善, 2000.

[5] 湯浅 太一, 安村 通晃, 中田 登志之 編:"はじめての並列プログラミング", bit 別冊, 共立出版, 1999.

[6] 佐藤 三久:"JSPP'99 OpenMP チュートリアル資料", RWCP, 2000.

[7] 三木 光範 他:"PC クラスタ超入門 1999", 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, 1999.

[8] 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.

[9] 男澤 昌哉 監修・著, 新井リンダ, 鈴木賢剛, 森蔭政幸 著："LINUX HA クラスター"、O'REILLY、2001.

[10] 森 眞一郎, 富田 眞治:"並列計算機アーキテクトから見た計算機クラスタ", 情報処理学会誌, Vol.39, No.11, pp.1073-1077, Nov. 1998.

[11] 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, Nov. 1998.

[12] 長島 雲兵, 関口 智嗣:"大規模計算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, Nov. 1998.

[13] 喜連川 優:"データベースとデータマイニングにおける並列処理", 情報処理学会誌, Vol.39, No.11, pp.1089-1094, Nov. 1998.

[14] 石川 裕, 堀 敦史, 手塚 宏史[新情報処理開発機構]:"RWCP におけるクラスタ開発記", 情報処理学会誌, Vol.39, No.11, pp.1095-1099, Nov. 1998.

[15] 先端情報技術研究所（AITEC）:"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", http://www.icot.or.jp/FTS/REPORTS/H11-reports/H1203-AITEC-Report7/AITEC0003R7-Frame-1.htm

[16] 先端情報技術研究所（AITEC）:"ハイエンドコンピューティング技術に関する調査研究 I", http://www.icot.or.jp/FTS/REPORTS/H11-reports/H1203-AITEC-Report3/index.htm

[17] 小柳 義夫:"ユーザからみた HPF 言語", 情報処理学会誌, Vol.38, No.2, pp.86-89, Feb. 1997.

[18] 内田 大介:"OpenMP による並列プログラミング 1", 立命館大学理工学部情報学科卒業論文, 2000.

[19] 土谷 悠輝:"OpenMP による並列プログラミング 2", 立命館大学理工学部情報学科卒業論文, 2000.

[20] 青地 剛宙:"PC クラスタ上での並列プログラミング環境の構築", 立命館大学大学院理工学研究科修士論文, 2001.

付録 1 PVM サイクロイドマスター(cyc_master.c)

```
#include <stdio.h>
#include <signal.h>
#include <math.h>
#include "defs.h"
#include "pvm3.h"
#include "second.c"

int Nslave;                 /* 起動したスレーブ数 */
int Tids[SLAVE];            /* 各スレーブのタスク ID */

/* シグナルの処理 */
void exit_handler(void)
{
    int sn;

    /* 全スレーブを終了させる */
    for(sn = 0; sn < Nslave; sn++)
        pvm_kill(Tids[sn]);

    pvm_exit();    /* PVM から離脱 */
    exit(1);       /* プログラム異常終了 */
}

int main()
{
    int i;
    double start, step, end , s_start, s_end;
    double qt, rm;
    double   lsum, sum = 0;
    double   at,bt;
        double    width = 2 * M_PI / N;

    /* スレーブ・プログラムを起動 */
    Nslave = pvm_spawn("sum_slave_s", (char**)0, 0, "", SLAVE, Tids);
    if(Nslave <= 0){
        fputs("PVM: Can't start slaves¥n", stderr);    /* エラーの表示 */
        return 1;                                      /* プログラム異常終了 */
```

```
}
/* シグナルの処理 */
signal(SIGINT, (void*)exit_handler);
signal(SIGTERM, (void*)exit_handler);

bt = seconds();

/* 処理を各スレーブに割り当てる */
qt = N / Nslave;    /* 商 */
start = 0;
for(i = 0; i < Nslave; i++){
    step = qt;

    end = start + step - 1;          /* end をセット */

    s_start = start * width;
    s_end = end * width;

    pvm_initsend(PvmDataDefault);    /* 送信初期化 */
    pvm_pkdouble(&s_start, 1, 1);    /* start をパック */
    pvm_pkdouble(&s_end, 1, 1);      /* end をパック */
    pvm_pkdouble(&width, 1, 1);              /* width をパック */
    pvm_send(Tids[i], 11);           /* スレーブに送信 */

    start += step;                   /* 次回の start をセット */
}


/* 全スレーブから結果を回収する */
for(i = 0; i < Nslave; i++){
    pvm_recv(Tids[i], 21);                   /* スレーブから受信 */
    pvm_upkdouble(&s_start, 1, 1);   /* start をアンパック */
    pvm_upkdouble(&s_end, 1, 1);     /* end をアンパック */
    pvm_upkdouble(&width, 1, 1);     /* width をアンパック */
    pvm_upkdouble(&lsum, 1, 1);              /* lsum をアンパック */

    /* 部分和を表示 */
    printf("%f - %f: %f    (TID:%x)¥n", s_start, s_end, lsum, Tids[i]);

    sum += lsum;    /* 部分和を合計 */
```

```
    }

    printf("          Total: %f¥n", sum);    /* 計算結果を表示 */

    at = seconds();

    printf(" time= %lf ¥n", at - bt);

    pvm_exit();    /* PVM から離脱 */
    return 0;      /* プログラム正常終了 */
}
```

付録 2 PVM サイクロイドスレーブ(cyc_clave.c)

```
#include <signal.h>
#include "defs.h"
#include "pvm3.h"
#include <math.h>
/* シグナルの処理 */
void sigterm_handler(void)
{
    pvm_exit();    /* PVM から離脱 */
    exit(1);       /* プログラム異常終了 */
}

int main()
{
    int ptid;
    double i;
    double s_start, s_end;
    double width;
    double lsum;
    /* シグナルの処理 */
    signal(SIGTERM, (void*)sigterm_handler);

    ptid = pvm_parent();    /* マスターのタスク ID を取得 */

    pvm_recv(ptid, 11);             /* マスターから受信 */
    pvm_upkdouble(&s_start, 1, 1);    /* start をアンパック */
```

```
    pvm_upkdouble(&s_end, 1, 1);       /* end をアンパック */
    pvm_upkdouble(&width, 1, 1);       /* width をアンパック */

    /* start から end までの和を計算 */
    lsum = 0;
    for(i = s_start; i <= s_end; i += width){
        lsum += (3/2*width - 2*sin(i + width) + 2*sin(i) + 1/4*sin(2*(i + width)) - 1/4*sin
(2*i));
      }
    pvm_initsend(PvmDataDefault);   /* 送信初期化 */
    pvm_pkdouble(&s_start, 1, 1);          /* start をパック */
    pvm_pkdouble(&s_end, 1, 1);            /* end をパック */
    pvm_pkdouble(&width, 1, 1);            /* width をパック */
    pvm_pkdouble(&lsum, 1, 1);             /* lsum をパック */
    pvm_send(ptid, 21);                /* マスターに送信 */

    pvm_exit();   /* PVM から離脱 */
    return 0;       /* プログラム正常終了 */
}
```

付録 3 PVM サイクロイド Makefile(Makefile.aimk)

```
.o:
        $(CC) $(CFLAGS) -o $@ $< $(LDFLAGS) $(LDLIBS)
        cp -p $@ $(HOME)/pvm3/bin/$(PVM_ARCH)

VPATH  =        ../

CC      =       gcc
OPTIONS         =       -pipe -O
CFLAGS =        $(OPTIONS) -I$(PVM_ROOT)/include $(ARCHCFLAGS)
LDFLAGS         =       -L$(PVM_ROOT)/lib/$(PVM_ARCH)
LDLIBS  =       -lm -lpvm3 $(ARCHLIB)

all:    cyc_master cyc_slave

clean:
        rm -f cyc_master cyc_slave *.o
```

```
cyc_master:	cyc_master.o
cyc_slave:	cyc_slave.o
```

付録 4 PVM サイクロイド(defs.h)

```
#define SLAVE		4	/* 起動するスレーブ数 */
#define N		10000000	/* サイクロイドの分割数 */
```

付録 5 OpenMP サイクロイド

```
#include <stdio.h>
#include <omp.h>
#include <math.h>

int main()
{

   int ID, thread;
   double width =2 * M_PI / N;
   int start, end;
   double sum = 0;
   double i;
   double is, ie;



#pragma omp parallel private(start, is, ie) shared(width) reduction(+ : sum)

      ID = omp_get_num_threads();
      thread = omp_get_thread_num();

      is = start * width;
      ie = (ID + 1) * width * width;

      for(i = is; i <= ie; i += width){
      sum += 3/2 * width - 2 * sin(i + width) - 2 * sin(i) + 1/4 * sin(2 * (i + width)) - 1/4 *
sin(2 * i);
      }

}
```

付録 6 時間関数(second.c)

```
#define  _INCLUDE_HPUX_SOURCE
#include <sys/time.h>
/*
        Fortran callable micro-second clock

        Revised to run on HP's workstations
*/
double second ()
{
    double t;
    struct timeval buffer;
    struct timezone dummy;

    gettimeofday (&buffer, &dummy);
    t = (double)buffer.tv_sec + ((double)buffer.tv_usec*1.0e-6);

    return (t);
}
```