



超並列計算研究会
Society for Massively Parallel Processing

PCクラスタ超入門

PCクラスタ型並列計算機の構築と利用

2000



会場：同志社大学 田辺キャンパス
日時：2000年9月7日(木)
9月8日(金)

主催：超並列計算研究会

講習会「PC クラスタ超入門」の開催にあたって

PC クラスタは通常使用されているパーソナルコンピュータを複数台ネットワーク接続することにより構築できるメモリ分散型の並列計算機です。近年の CPU の性能の大幅な向上に伴い、場合によっては、スーパーコンピュータにも匹敵する性能を持つシステムを構築することが可能です。また、フリーウェアを使用し、これまでの大型の汎用機やワークステーションにおいて最もコストがかかっていた人件費と保守料を、ある程度のノウハウを持つことで削減することが可能であるため、コストを大幅に削減することができます。

この 1 年で PC クラスタシステムをとりまく環境は大きく変化したように思えます。最も大きく変化した点は、PC クラスタが急速に普及したことでしょう。8 台程度の小さな規模のクラスタシステムは珍しくなくなっています。また、256 台を超えるような大規模なクラスタシステムも日本においてもいくつか見られるようになってきました。このような状況において、PC クラスタに興味を持つ方も増えてきました。

本講習会は昨年度の同じ名称の講習会に引き続いて 2 年目の開催となります。昨年度の講習会は、全くこの分野の知識のない近所の大学の学生を対象として、せいぜい 20 人も集まれば良いという目論見でスタートしましたが、結果的には大盛況で、100 名近くの方々が全国からご参加いただきました。これは望外の幸というより大誤算で、ご参加いただきました多くの方のレベルは非常に高く、我々講師陣が質疑応答で窮する場面もあったかと思います。今年は第 2 回目でもあり、また内容も充実させるために 2 日間の開催としました。講師陣として昨年度は同志社大学工学部知識工学科の知的システムデザイン研究室の関係者がほとんどでしたが、今年は外部の方にも多く入っていただき、より充実した内容としました。ただ、この講習会の目的はあくまでも「超入門」であり、対象者は「PC クラスタに興味を持っているけれどもまだ設置できない」という初心者です。なぜなら、我々講師陣も、一部の特別の専門家を除いて、PC クラスタを立ち上げたのは最近であり、まだまだ完全に使いこなすまでには至っていないからです。むしろ、今回の講習会も、昨年同様、最近 PC クラスタを立ち上げたので、不十分ではありますがそのノウハウを公開し、皆さんとご一緒に議論しましょうという趣旨で開催させていただくものです。どうぞこの 2 日間、我々講師陣は精一杯頑張りますので、皆様も一緒に PC クラスタの技術と利用法を考えていただければと思っています。

2000 年 9 月

三木 光範（超並列計算研究会代表、同志社大学工学部）

超並列計算研究会のご案内

本講習会を企画しました超並列計算研究会は1994年5月に設立され、並列処理の技術と利用法を議論するため、毎年数回の研究会、見学会、あるいは講習会を行っている任意団体です。どこの学会の研究会でもないのには理由があります。それは、この研究会が学会横断的な性格を持っているからです。我々は、並列処理の技術と利用法を考えるのが目的であり、情報処理の並列化技術も重要であり、一方、機械、化学、あるいは電気・電子などの各工学分野、さらには経済やビジネスの分野での並列処理の応用を考えることが重要であると思っています。特に、並列化のために計算のモデルが逐次処理の時とは異なるというように、用いる計算機環境と、解くべき問題との相互関係にも大きな興味があります。是非、こうした分野にご興味のある方は超並列計算研究会にご参加いただきますようお願い申し上げます。超並列計算研究会のホームページ（<http://is.doshisha.ac.jp/SMPP/>）をご参照ください。

なお、超並列計算研究会への入会および年会費は無料です。メーリングリストに登録していただくだけで各種の案内が届きます。各研究会などへの参加費はその都度お支払いください。参加には会員と非会員との区別はありません。どなたでも自由にご参加ください。

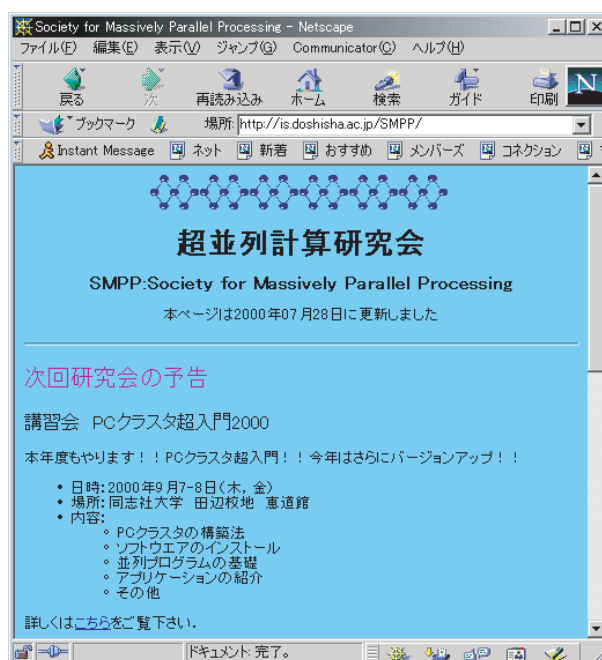


図 1: 超並列計算研究会ホームページ

目次

1PC クラスタの作り方	1
1.1 はじめに	1
1.2PC クラスタとは何か？	1
1.3PC クラスタの構築 超超入門	8
1.4PC クラスタの構築 –ハードウェア編–	19
1.5PC クラスタの構築 –ソフトウェア編–	26
1.6 実際の運用	29
1.7PC クラスタ再考	30
1.8 最近の動向	30
1.9 参考文献	32
2MPIによる並列プログラミングの基礎	33
2.1 はじめに	33
2.2 並列プログラミングの基礎知識	33
2.3 インストールに関して	39
2.4 並列プログラミング	43
2.5 1対1通信	47
2.6 グループ通信	52
2.7 その他	57
2.8MPI のプログラムの実行	63
2.9LAM と MPICH の通信速度	66
2.10 おわりに	68
2.11 参考文献	69
3Diskless Cluster 構築入門	70
3.1 はじめに	70
3.2Diskless Cluster とは	70
3.3Diskless Cluster のブートの仕組み	71
3.4 構築手順	72
3.5 より高機能・高性能に	84
3.6 おわりに	89
3.7Cluster の構築・運用に関するリソース	89
3.8 謝辞, およびフィードバック	90
3.9 参考文献	91
4PC クラスタにおけるベンチマークとジョブ管理	92
4.1 はじめに	92
4.2 ベンチマークとは	92

4.3 ベンチマークプログラムの分類 ¹⁾	93
4.4 ベンチマークプログラムのインストール	98
4.5 ベンチマークの測定結果と分析	102
4.6 ベンチマークを行う上での注意	105
4.7 クラスタシステムにおけるジョブ管理	107
4.8JMS によるジョブの管理と実行	108
4.9 ジョブマネージメントシステムの紹介	113
4.10Distributed Queuing System (DQS)	114
4.11 おわりに	118
4.12 参考文献	118
5ALINKA LCMを用いたPC クラスタシステムの構築	119
5.1 はじめに	119
5.2Aoyama Plus Project	119
5.3ALINKA LCM と Debian VT-PC Cluster Selection	124
5.4Debian GNU/Linux VT-PC Cluster Selection を用いた PC クラスタの構築	125
5.5ALINKA LCM ドキュメント	134
5.6 その他, 有用なソフトウェア	164
6 並列処理入門	166
6.1 はじめに	166
6.2 超並列計算機の発達	166
6.3 計算の高速化と並列処理	167
6.4 並列計算機の分類	168
6.5 並列計算機の構成方式	169
6.6 プロセッサ間ネットワーク (相互結合網)	170
6.7 並列処理のプログラム	172
6.8 並列処理の効率	172
6.9 数値計算の構造	174
6.10 数値計算の並列化	175
6.11 おわりに	175
6.12 参考文献	177
7 アプリケーションの並列化の例	178
7.1 はじめに	178
7.2 領域分割を用いた並列計算	178
7.3 遺伝的アルゴリズムの並列化	185
7.4 その他 (並列以外のツール)	185
7.5 おわりに	186
7.6 参考文献	186
8 並列化手法のご紹介と, クラスタを組む上での各種 Tips	194
8.1 はじめに	194
8.2 テスト環境	195
8.3 並列化の実際	196
8.4 ほんの少し高度な並列化	198
8.5 閑話休題	199
8.6 クラスタを組む上での Tips	200

8.7 参考文献	202
9 グローバルコンピューティング－ Condor と Globus Toolkit の紹介－	210
9.1 はじめに	210
9.2 Condor	211
9.3 Globus Toolkit	220
9.4 代表的なグローバルコンピューティングシステム	226
9.5 おわりに	228
9.6 参考文献	228
10 SCore 型クラスタ	229
10.1 はじめに	229
10.2 背景	229
10.3 RWC クラスタ開発の歴史	229
10.4 SCore Cluser System Software	230
10.5 アプリケーションの紹介	233
10.6 ユーザ	234
10.7 終わりに	234
10.8 参考文献	236

PC クラスタの作り方

同志社大学 工学部

廣安 知之

tomo@is.doshisha.ac.jp

■ はじめに

できるだけ安く高性能な計算機が欲しい。今、買うなら PC クラスタだと聞いている。なんだか面白そう。などなど、PC クラスタの構築の動機はいろいろあるでしょう。

本章では、PC クラスタを構築する上での必要事項をずらずらと並べていきます。後述する通り、PC クラスタを単純に構築するのは非常に簡単ですが、その性能を向上させようと考えれば、知らなければならぬこと、やらなければならぬことがほぼ無限にあります。ここでは、それら全てをもちろん網羅することはできません。そのため、この章の目的は、読者のみなさんに PC クラスタ関連の知識を理解してもらうことではなく、こういうことがあるんだな、こういうことが必要なんだな、ということを知ってもらうところにあります。いくつかの基本的事項については、次章からの詳しい解説を参考にして下さい。また、キーワードとそのポイントはできるだけ多く掲載しているつもりですので、それをきっかけにして、読者のマシンのバージョンアップを手助けできればと考えています。

■ PC クラスタとは何か？

□ クラスタとは

クラスタ (cluster) とは英語で「ブドウなどのふさ、同種類のものの群れ」という意味です。それに対して PC クラスタとは、一般的に使用されているパーソナルコンピュータ（いわゆるパソコン、PC）をネットワーク結合して構築した並列計算機の 1 種です。すなわち、PC がネットワークによってブドウのふさのように群れをなしているわけです。

Pfister によるとクラスタの定義は以下のようになります⁴⁾。

クラスタ

クラスタは単一で稼働するコンピュータの集まりで、一つの計算機資源として使用可能な並列もしくは分散システムである。

さらに、一般で市販されている部品やハードウェアのことをコモディティハードウェア、もしくは、off-the-shelf と呼んでいますが、それらのコモディティハードウェアから構成されているもののみをクラ

スタと呼んでいる人もいます。最近では、IBM やコンパックなどの各社のスーパーコンピュータは複数のコンピュータがネットワーク結合されることにより一つのシステムを構成しているので、これらのシステムもクラスタシステムと言うことも言えますが、上記の定義では、これらのシステムはクラスタシステムではありません。

均一なマシンからなるクラスタをホモジニアスクラスタ、異機種混合なマシンからなるクラスタをヘテロクラスタと呼んでいます。



図 1.1: クラスタシステム (ヘテロ)

□ クラスタの例

世界のスーパーコンピュータのランキングを掲載しているのが、Top 500 Super Computer Sites (<http://www.top500.org/>) です。このリストにもいくつかの PC クラスタが名を連ねています。これらのクラスタは文字通り、スーパーコンピュータと同等の性能があると言えます。第 62 位の C Plant Cluster, 第 207 位の NT CLuster, 第 364 位の Avalon cluster などです。これらは、台数が非常に大きいものであったり、ネットワークが高価であったり、一般には手の届かないものです。

ここで、いくつかの、PC クラスタシステムを紹介しておきましょう。

国内のクラスタ

- RWCP クラスタ

[www://pdswww.rwcp.or.jp/](http://www.pdswww.rwcp.or.jp/)

日本で代表的な PC クラスタは新情報開発機構 (RWCP) の PC Cluster システムです。ここでは高性能なネットワークドライバ (Score) などを開発し公開するなどの活動を行っています。

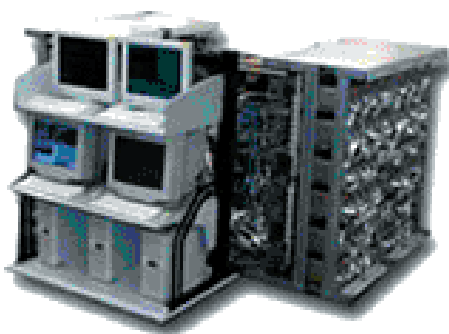


図 1.2: RWC クラスタ

このシステムにはアルファを CPU と採用したものと Pentium Pro を採用したものがあり、それらは Myrinet でネットワーク接続されています。

- 東京工業大学 松岡研究室 クラスタ

<http://matsu-www.is.titech.ac.jp/index-j.html>

日本でも代表的なクラスタ関連の研究を行っている研究室です。今年中に 200 台以上のクラスタが導入されるそうです。



図 1.3: 東工大 松岡研 クラスタ

- 東京大学 喜連川研 ATM クラスタ

<http://www.tkl.iis.u-tokyo.ac.jp/Kilab/Research/NEDO-100/index-j.html>

このクラスタの CPU の速度は位まではそれほどでもなくなりましたが, ATM 接続されており, 特徴的です.



図 1.4: 東大 喜連川研 クラスタ

- 青山学院大学 Aoyama+

<http://www.phys.aoyama.ac.jp/aoyama+/>

PC クラスタのブームを作ったともいえる青山学院大学のクラスタです. 詳しくは, 中田さんの章をご参考下さい.

- 同志社大学 知的システムデザイン研究室 クラスタ

<http://www.is.doshisha.ac.jp/cluster/index.html>

われわれのクラスタです. 現在は小規模なクラスタシステムが複数あります. 来年度には 200 台以上の PC からなるシステムが構築される予定です. 主に, クラスタに適した最適化アルゴリズムの開発を行っています.



図 1.5: 同志社大学 ISDL クラスタ

海外のクラスタシステム

- Avalon クラスタ

<http://cnls.lanl.gov/Frames/avalon-a.html>

図 1.2.2.2 に示すのは Los Alamos National Laboratory の Avalon と呼ばれる PC クラスタシステムです。



図 1.6: Avalon クラスタ

これはアルファマシンを 140 台、Myrinet と呼ばれる Giga bit ネットワークで接続した PC クラスタシステムです。これが非常に有名になったのは世界の Super Computer Top 500(<http://www.top500.com>) に PC クラスタシステムがランキングしたからです。これが PC クラスタシステムはコストパフォーマンスだけでなく、性能も超並列機と肩を並べることができることを証明したのです。しかしながら、このシステムを構成する PC は Pentium と比べると高価で FastEthernet と比べると更に高価なデバイスを使用しています。

また、この研究プロジェクトは Beowulf プロジェクトと呼ばれコモディティハードウェアによる PC とフリーのソフトウェアを走らせることの利点を研究しています。つまり、コンピュータパワーよりも価格性能比やプロセッサ性能比に興味を持っているのです。さらに Linux を使った場合の性能について大まかな指標を与えるのが目的ある Beowulf クラスと呼ばれるシステムの概念も提唱しています。

- CPlant クラスタ

<http://www.cs.sandia.gov/cplant/>

図 1.7 は先に紹介した世界の Super Computer Top 500 に現在（2000 年 6 月現在）62 位にランキングされている Sandia National Laboratory の Cplant PC クラスタシステムです。最近、node の増設が行われたようです。



図 1.7: Cplant クラスタ

このシステムも CPU としてアルファをネットワークも高速なものを使用しています。

- NCSA NT Cluster

<http://www.ncsa.uiuc.edu/General/CC/ntcluster/>

Web 上に図が公開されていないのが残念ですが、イリノイ大学の NCSA には OS として NT を採用したクラスタがあり、現在、Super computing Top 500 のリストのうち第 207 位に位置しています。このクラスタも Myrinet を使用し、通信ライブラリとして HPVM を使用しています。

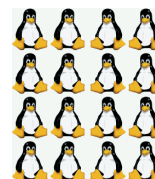
- LosLobos クラスタ

<http://www.ahpcc.unm.edu/Hardware/LosLobos.html>

LosLobos クラスタはニューメキシコ大学のクラスタです。500 台以上のクラスタを Myrinet で接続しています。Self Made でなく、IBM のマシンであることが特徴です。



図 1.8: Los Lobos クラスタ



□ これまでのコンピュータ，並列計算機と何が違うのか？

紹介したクラスタシステムを見ればわかる通り，いくつかの PC クラスタシステムはスーパーコンピュータと呼んでもおかしくないものがあります．それでは，PC クラスタシステムは，これまでのスーパーコンピュータとは何が違うのでしょうか？ここでは，次の2点を挙げておきます．

- 手にしているシステムをスケールアップ+チューンアップすることでスーパーコンピュータを構築することが可能である
- 価格を極力下げることが可能である

F1 のレースに乗用車で参戦することは物理的には可能です (レギュレーションにひっかかるのかな?). しかしながら，その乗用車をいくらチューンアップしたところで，レースに勝つことはできません．乗用車と F1 のマシン (ハードウェア) 自体がまったく異なっている上に，それを操作する腕前 (ソフトウェア) が違うからです．

今までは，計算機の世界でも同様でした．僕達が手にしている PC やワークステーションをいくらチューンアップしても，スーパーコンピュータにはならなかったのです．それは，例えば，使っている CPU は汎用製品で，一方は専用の製品だったからです．しかしながら，その状況が変わって来ました．すなわち，スーパーコンピュータで使用している部品も僕達が一般に使っている製品との差がなくなってきてしまったのです．CPU はあっと言う間にその動作速度を上げてきました．ハードウェアは大容量になり，アクセス速度も向上しました．さらに，開発コストの問題があり，スーパーコンピュータでも，汎用製品を使用していかなければ立ち打ちできなくなってきたのです．これは何を意味するのでしょうか．部品が一緒ですから，スケールアップやチューンアップすることによって，今使用しているシステムがスーパーコンピュータになっていくということです．自家用車で F1 のレースに参戦することが可能になったのです．

PC クラスタでは，極力コストを下げるができます．これまでのスーパーコンピュータの値段の中には，ハードウェアとソフトウェア，そして保守料金と人件費が入っていました (開発費も製品に付加するのかな?). そのコストをハードウェアだけに下げることが可能となります．これは価格破壊以外のなにものでもありません．まず，ソフトウェアはフリーのものを使用する．保守はしない．壊れたらそれまでと開きなおる．システム構築の人件費は助手や学生，その他のボランティアでこれも削減する．こうすることで，ハードウェアだけの値段にすることが可能になるのです．純粋なハードウェアだけの値段であれば，スーパーコンピュータも PC クラスタも変わらないと思います．だから，より性能の良いシステムの構築を求めて，有料のソフトウェアを使用し，保守を外部に委託し，構築も外部に委託していく形態にすると，非常に高価なシステムとなります．ですから，これまでのスーパーコンピュータと比較して，PC クラスタは安いというのは嘘です．価格設定がユーザーの都合でユーザーが設定できるようになっただけのことです．

□ PC クラスタを所有するメリットとは？

クラスタを所有すると以下のようなメリットがあります.

- 数値計算の向上 (「HPC(High Performance Computing)」と呼ばれる)
- 可用性の向上 (「HA(High Availability)」と呼ばれる)
- 計算機環境の専用 (スループットの向上)
- 古いマシンの有効活用
- 技術進展に追従したシステムのバージョンアップが可能
- たくさんのマシンを管理できる
- やることがいっぱいあってはまる
- 楽しい

■ PC クラスターの構築 超超入門

□ それでは作ってみましょう !!

ただ単に PC クラスターを構築するのはいたって簡単です。それは先に述べたように普通乗用車でも自動車レースに参加できるからです。ちょっと運転できれば、どこかで車を買って来てレースに参加すればよいのです。

PC クラスターでも同じことです。PC を複数台用意し (OS や開発環境, 並列ライブラリがすでに入っていたらなおよい), ネットワーク接続して, 必要なソフトウェアをインストールすればもうできあがりです。

ハードウェア編

まずは, ハードウェアを揃えましょう。PC 複数台にネットワークハブ, ケーブルなどが必要となります。

PC の購入方法はいくつかの方法があります。ポイントは後で説明するインストールする OS である Linux が正常に稼動し, 求める性能ができるだけ安く手に入ることです。また, クラスターなどを構築する場合には多くの PC を購入するわけなので, 初期不良などのトラブルの発生も多くなる可能性があります, そのトラブルに対するアフターケアが一番に問われる点であるかもしれません。

その 1 一般の組み立てられた製品を購入

大手のコンピュータメーカーが製造している PC を購入する方法がまずあげられます。ショップブランドと称してこれまで部品を販売していた店などが構築している AT 互換機もあります。さらに最近では通信販売などで売り上げをあげている会社も多く存在します。セット商品の場合には比較のお買い得なものが多く存在します。Windows 用のアプリケーションが多く添付されている場合があります。しかしながら, 部品が決定されているため, 特定の部品を使用したい場合 (得にネットワーク関係) には問題があります。

その 2 一般の組み立てられた製品を購入 (1 部の部品をグレードアップおよび交換)

ショップブランドの場合, 上記の製品は実は部品をユーザーが指定し, 差額を払えば購入できるシステムを導入している場合があります。この場合には使用したい部品や性能を満たす部品を注文すれば OK です。ただし, 選択が限られている場合もあります。もちろん, 部品に関する知識が必要です。また, 大手のショップブランドの場合, 部品に関する細かい問い合わせや指定を行うと途端に対応や納期が悪くなる場合があります。

その 3 Linux 対応の組み立てられた製品を購入

本講習会では OS として Linux の説明を中心に行います。問題となるのは, すべての製品が Linux 上で正常に稼動するとは限らないということです。ここが Linux を使用する際の大きな壁となります。Linux を使用する際の鉄則としては, Linux で稼動する部品を揃えるということでしょう。しかしながら, この情報を常に入手することは非常に大変なことです。最も簡単な方法は, Linux が稼動することを謳っている PC を購入することです。これらを扱っているメーカーやショップもここ 1 年でぐっと増えました。

その 4 上記その 2 およびその 3 の部品を購入して自作

少しでも価格を押さえるためには上記の購入方法で自作する方法があげられます。しかしながら, 価格調整を人件費で行っているような場合には, ユーザーが自作したからと言って安くならない場合もあります。また, 最大の問題は部品は購入先にて保証されるでしょうが, 全体のシス

テムとしては保証されない点です。この方法の最大のメリットは実は価格ではなくて PC の組み立てが楽しいというところにあるかもしれません。

その5 部品も自分で購入し自作

部品に対する知識も豊富で部品選びが面倒と感じず、逆に楽しいと思える人であれば、部品から購入し自作する方法もあります。部品から選択しているので、各パーツに関しては最も満足なものが手に入れられる方法です。しかしながら、部品に関する知識は必要であり、部品単体以外は全て自分の責任となります。

その6 PC クラスタ製品を購入

PC もネットワークも揃っていて (もちろんケースも特別の場合もある)、OS も開発環境も並列ライブラリもインストールされているような製品を購入する方法、これは究極の方法です。ユーザーは何もすることはありません。昨年まではアルファの CPU を搭載したものが中心でしたが、この1年でいくつかの会社が、AT 互換機のクラスタも扱うようになりました。来年は PowerPC もしくはアップル G4 Cube のクラスタが出てくるのでしょうか。

PC がそろいました。

モニタ、キーボード、マウスはインストールや障害が起こった際にのみ必要なもので、1 セットで十分でしょう。余裕があればそれらの切替え機があれば便利です。

ネットワーク関連製品はこの1年でぐっと安くなりました。100Base TX の8ポートスイッチングハブがついに1万円を切りました。ケーブルは、完全に完成したものを購入しても良いですが、設置場所によって長くなったり短くなったりするので、数が多い場合には、ケーブルの加工セットを用意しておくとても非常に便利です。

ソフトウェア編

ハードウェアがそろったので、次はソフトウェアです。

OS は Linux で、ディストリビューションを選択し、開発環境と並列ライブラリをインストールしましょう。

Linuxって何だ??

Linux はオープンソースなフリーウェアの Unix 互換 OS です。そもそもはフィンランドの大学院生が開発したものが、インターネットを通じて改良され、さらに、GNU などの優れたフリーウェアアプリケーションの存在により急速に発展し、多くのユーザーを獲得しました。現在では個人だけでなく企業でもネットワークサーバー機の OS として広く利用されています。その特徴は、Unix 互換のためネットワークの利用に特に適していること、フリーであるために必要なコストがほとんどかからないこと、サポートするユーザーが多数、強力でありかつインターネットなどで繋がっているためバグなどが発見されてもすぐに改善され、かつ、すぐに公開されることなどがあげられます。

何故 Linux なの??

PC クラスタシステムでは、OS として Linux を採用している場合がほとんどです。その理由は以下の様なものです。

- 価格が安い

そもそもオープンソースでフリーであるので、コストの点では商用の OS とは比較になりません。

- 安定性

フリーであるからと言って安定性に欠けるわけではありません。メモリプロテクトな OS であるので、ユーザーが作成したアプリケーションによってシステムがフリーズすることはありません。これはプログラムの開発段階では必須事項です。

- オープンソース

Linux はオープンソースであり情報交換がインターネットを通じて行われているため、開発が迅速である。とくにセキュリティホールが見つかった場合や多くのユーザーが希望しているような改善に対しては非常に敏速に対応される仕組みとなっています。また、カーネルとネットワーク関係がクラスタシステムの性能を大きく左右するため、この部分がオープンであることはクラスタシステムの開発には必須です。

- 性能

Linux のカーネルも十分高性能であると言えますが、Linux の性能を大きく支えているのが FSF (Free Software Foundation) のいわゆる GNU のアプリケーション群であらう。GNU C コンパイラや GNU Emacs といった Editor は Linux だけでなく UNIX そのものを支えていると言っても過言ではありません。さらに最近では Gtk+ という X Window のライブラリが作成され、デスクトップ環境や高度なマルチメディアアプリケーションが作成されています。

- プロセスの管理が可能

システムの性能を把握し向上させるためには、どのようなプロセスが起動していて不必要なプロセスが起動していないことが重要である。特にプログラムの開発段階では不必要なプロセスが発生しがちであり正確にプロセスを終了させることが望まれます。

- 多数の利用者

Linux には非常に多くのユーザーがいます。特にクラスタシステムを構築している関係者であればなんらかの形で Linux に興味があるはずで、これはクラスタに関連する技術がまず Linux 対応として出現するであろうことを示しています。また、ユーザーが多いためにインターネットを通じて協力ができます。

- その他

UNIX 互換であるためにそもそもネットワークに適した OS である。これはネットワークを利用したメッセージ通信を行うクラスタシステムとしては大きな長所です。

このような理由で Linux はクラスタシステムの OS の中心です。この Linux に対抗するのが Windows NT でしょうか。しかしながら、Linux は UNIX 互換であり、技術的にはいわゆる枯れた OS です。一方、Windows NT は比較的新しい OS であり、特にリリースされた Windows 2000 はさらに新しい OS であるので、今後問題が噴出して来るものと考えられます。そもそも Windows の長所としては統一的な GUI にあるわけですが、クラスタにより数値計算を行うような場合にはそれがあまり有効とはなりません。

このような Linux にも欠点がないわけではありません。これまであまりユーザーが着目していなかったネットワーク面では弱い点はいくつかあります。MPS での処理の問題や NFS などの性能などがそれです。よって同じフリーな OS でも Net BSDなどを好む開発者もいますが、これだけユーザーが多数となると今後飛躍的にこの方面の解決も行われることでしょう。

Distributionって何だ??

Linux は近年非常に有名になり広く使用されるようになってきました。しかしながら、厳密に言えば Linux とはカーネルのことを指します。カーネルだけではいろいろなサービスができず、インストールも難しいのです。そのため、いろいろなグループが「カーネル+アプリケーション群+インストーラー+パッケージ管理方法」をセットにして提案しています。これが Linux ディストリビューションです。よって、ディストリビューションが異なるとほとんど異なった OS であるといつていいくらいに異なっています。特に初心者が印象を大きくするのはやはりインストーラのできでしょう。実際の性能よりもインストールの容易さによって評価を決めてしまう場合が多くなります。

ディストリビューションは大きくわけて 3 種類あります。Slackware 系と RedHat 系、そして Debian GNU/Linux 系です。

1. Slackware 系

Slackware は Linux が普及し始めた初期のディストリビューションです。バイナリパッケージなどの方式を初めて普及させたディストリビューションでもあります。非常に自由度の高いディストリビューションで歴史的な経緯からバイナリパッケージの管理方式は RedHat や GNU/Linux Debian と比較すると依存関係などのチェックなどが行われないといったように初心者には難しいディストリビューションです。一方、これを基にした Plamo Linux という日本語ディストリビューションもあります。最近、ユーザー数は減っていると思われます。

2. RedHat 系

RedHat は現在最も普及しているディストリビューションでしょう。多くの商用のアプリケーションはまず RedHat に対応したものを発表しています。rpm は使いやすいパッケージ管理方式です。これを基にした日本語ディストリビューションも数多くあり、日本語 RedHat, Turbo Linux, Vine Linux, Laser5 Linux, Kondara などがあります。Vine Linux や Kondara などは安定しており、Windows ユーザーがとっつきやすいディストリビューションでしょう。

Extreme Linux と呼ばれるディストリビューションでは (<http://www.extremelinux.org/>)。これにはカーネルと各種ネットワークドライバ、MPI や PVM などが納められています。(しかしながらバージョンが少々古いため、次期バージョンの発表が待ち望まれるところであります。)

Softek と Turbo Linux は並行処理が行える enFuzion 6.0 を発売しています。

住商エレクトロニクスはコンパックのアルファマシンに Kondara をインストールしたモデルを販売しています。

RedHat をベースにしていますが、それぞれが完全に互換性がある保証はなく、また、独自に開発を進めると宣言しているパッケージもあります。

3. Debian Gnu/Linux 系

Debian GNU/Linux は非常に強力なパッケージ管理方式を持つディストリビューションです。筆者のお気に入りです。それは以下のような理由があります。

- 非常に強力な依存性などをチェックするパッケージ管理
- 一度インストールしたらバージョンアップすればよいだけという扱いの良さ
- 非常に多数のパッケージの存在
- パッケージの一元管理 (<http://www.debian.org>, <http://www.debian.or.jp>)
- apt によりシステムのバージョンアップや並列環境の構築が簡単
- Debian パッケージであれば、現在のところ必ず互換性がある

- RedHat 用のバイナリがあれば容易に Debian 用のパッケージに変換可能
- 必要なサービスはユーザーが開始するというポリシー
- ネットワークからのインストールやバージョンアップによって常に最新のシステムを維持することができる

インストールや設定の容易さとユーザーの変更の容易さとはトレードオフの関係にあります。確かに、日本語などがはじめから設定されているディストリビューションでは何もしなくても最初から基本的なソフトが使用できるようになっています。逆に、入らないアプリケーションが知らない間に起動していたり、思うように設定できづらい場合も多々あります。設定の多くがユーザーにまかされているディストリビューションでは初心者には導入が難しい面もありますが、逆に言えばユーザーの自由度が大きいということになります。

OS やディストリビューションの選択の幅は広く採用の決定はユーザーの趣味的にどうしてもなってしまう。クラスタシステムは数値計算を行う者にとっては大工のかなづちやノコギリにあたるものでしょう。それゆえに、身に覚えのないアプリケーションがインストールされているシステムではなく、一つ一つ、ユーザーが必要なものだけをインストールするようなシステムが筆者の趣味であります。

日本語の問題やインストールの問題は、Debian GNU/Linux をベースにした Omoikane の出現や、海外のディストリビューションである、Storm Linux や Corel Linux によって解決されつつあります。

並列ライブラリってなに??

開発環境はコンパイラなどです。アプリケーションを構築するには、C や Fortran, C++ などのコンパイラが必要となります。

さて、並列ライブラリとはなんでしょうか。

PC クラスタは並列計算機の種類で言うと、分散並列機になります。複数の CPU が存在し、かく CPU はそれぞれメモリを持っています。それぞれのメモリの中に格納されているデータは担当の CPU はアクセスが可能ですが、それ以外の CPU からはアクセスできません。それでは、他の CPU が管理するメモリ内のデータにアクセスしたい場合はどうするのでしょうか？

それはメッセージパッシングという方法を活用します。すなわち A という CPU と B という CPU が存在していると仮定し、A が B の管理するメモリのデータにアクセスしたい場合には、A が B に”お前のデータの hogehoge を送ってくれ”とメッセージを投げるのです。そうすると B は A にそのデータを送り返してやればよいのです (もちろん逆も可)。こうすることにより、システム内のデータにそれぞれの CPU がアクセスすることが可能になりました。

ただし問題は、ユーザが作るアプリケーション内にこのメッセージを明示してやらなければならないということです。一つ一つ最初から書いていたのでは、ユーザの負担が莫大なものになります。そのために用意されているのが、並列ライブラリです。この並列ライブラリで定義する関数を利用することによって、ユーザは並列処理が行えます。

並列ライブラリとしては、PVM や MPI などいくつかの種類があり、また、実装もいくつか存在します。

どうやってインストールするの？

一般に Linux のインストールは CD-Rom によって行われます。それぞれのディストリビューションを購入することもできますし、雑誌に付録として付いていることもあります。インターネットなどを通じてダウンロードすることもできます。

CD-Rom によつてのインストールの長所は非常に高速にインストールが可能であるという点です。しかしながら、ハードディスクの容量が巨大化し同時に価格が低下している現在、ローカルにミラーサイトを構築してそこから ftp にてインストールをする方法をお勧めします。これは例えば 1 日に 1 度、夜中などに各ディストリビューションのバイナリやソースをサイトより入手しローカルのバイナリやソースを常に新しいバージョンのものにしておくのです。こうすることにより、最新のものがかつ必要なアプリケーションが常に手に入ることとなります。これは実際に試してみると非常に便利です。

それじゃあ ちょっとインストールしてみましょう !!

ディストリビューションのインストール

基本的なディストリビューションのインストールは各ディストリビューションによつて多少は異なりますが大まかに言えば以下の流れで行われます。CD-Rom から起動させ、質問に答えていけば良いようになっています。

1. キーボードの選択
2. ハードディスクのパーティションの設定と初期化
3. 基本アプリケーションのインストール
4. ネットワークの設定
5. 時刻など基本的な設定
6. Root のパスワードの設定
7. ユーザーアカウントの作成

詳しいインストール方法は各種見られる入門書を参考にして頂きたいと思います。

このインストールの際に知っておかなければならない事項としてはハードディスクのパーティションとネットワーク関連の事項でしょう。

● ハードディスク関連

Linux では通常のファイル領域と swap と呼ばれるファイル領域が必要です。よつてハードディスクを少なくとも 2 つに分割しなければなりません。スワップは実メモリが足りなくなった場合に仮想記憶として使われるのですが、メモリが大量に載っているマシンではあまり問題となりません。また、スワップを頻繁に使用するとその処理速度は低下すると言われています。

さらにスワップ領域以外の領域を複数に分割して利用しても良いし、その分割した領域に複数の OS をインストールすることも可能です。

ハードディスクには MBR(Master Boot Record) というブート領域があります。ここではハードディスクの領域管理も行っています。この MBR が管理できる領域は Primary (物理) パーティションが 4 つまでで、それ以上に使用するためには Logical (論理) パーティションを使用しなければなりません (16 個まで)。

ハードディスクでは起動時には BIOS からローダーを呼び出し、そのローダーが Linux カーネル本体を呼び出すという多段階の仕組みになっています。Linux の標準ローダーが LILO(Linux LOader) です。

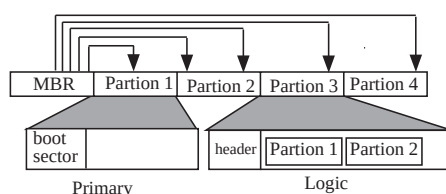


図 1.9: ハードディスクのパーティション

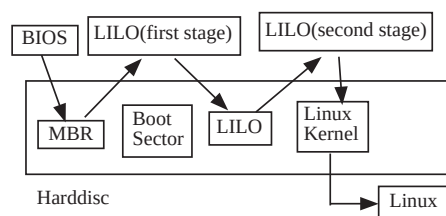


図 1.10: 起動の流れ

● ネットワーク関連

Ethernet 上で通常行う通信では、TCP/IP と呼ばれるプロトコル (約束ごと) に従って通信が行われています。まず、各通信機器には世界で一意に定まる名前が付けられています。これが IP アドレスで、192.168.1.2 といったような形で記されるのです。このアドレスでユニークな点は名前だけでなく所属するネットワークも同時に付記されている点でしょう。このネットワークはネットマスクと呼ばれるマスクで知ることができます。例えば上記のアドレスが 255.255.255.0 というサブネットマスクを持っている場合、192.168.1.* というアドレスは同一のネットワークにあることを示しています。さらに他のネットワークへ情報を伝えたい場合には外のネットワークと繋がっている機器を知っておかなければいけません。これがデフォルトゲートウェイです。デフォルトゲートウェイとなっている機器が自身のネットワーク以外の相手に情報を伝えたいという情報を得た場合、そのデータは外のネットワークに渡す仕組みとなっています。しかしながら、この IP アドレスは人間にはなかなか覚えにくいものです。一方でニックネームであれば覚えやすいものです。そこで通信機器に tiger であるとか dog といったようなニックネームをつけます。そしてそのニックネームと IP アドレスとの対応表をある機器が持っていて、その機器に対して問い合わせることでそのニックネームを持つ機器の IP アドレスを知るのです。この対応表を持つ機器を DNS(Domain Name Server) と呼びます。しかし、一つのマシンが世界中のすべての対応を知る仕組みでは対応表が巨大になり、かつ、ネットワークも渋滞してしまうので、これに対しては分散してデータを持つ仕組みになっています。例えば同志社大学の A というサーバーは同志社内の対応表を持っています。同志社以外のアドレスの問い合わせがあった場合には、B という日本の大学のサーバーのリストを持ったサーバに問い合わせます。これは同志社という領域が大学という領域の下位に設定しているからです。これがドメインシステムです。同志社内のマシンはすべて *.doshisha.ac.jp という名前が正式名であり、A.doshisha.ac.jp というサーバーが対応表を保持しています。doshisha.ac.jp は上位のドメイン、ac.jp のサーバーである B に問い合わせることができます。例えば、早稲田大学のサーバーは C であるとする。同志社から早稲田大学のマシン D のアドレスが問い合わせされた場合、まず、A.doshisha.ac.jp に尋ねられ、上位の B.ac.jp へ渡されます。B は C.waseda.ac.jp を紹介し、C.waseda.ac.jp では D.waseda.ac.jp の情報を持っているのでそれを答えとして返す。という仕組みになっています。

さらに詳しい解説はネットワーク関連図書に譲ります。

表 1.1: ネットワーク設定事項

設定事項	説明
ip address	マシンの番号
subnet mask	グループを示す番号
default gate way	他のネットワークへの出口となるマシンの ip address
name server	ip address とマシン名を対応づけるデータを有する ip address

バイナリパッケージによる並列ライブラリのインストール

それではバイナリパッケージを使って並列ライブラリをインストールしてみましょう。ここではパッケージのインストールについて説明しますが、実際の使用にはいくつかの設定が必要です。mpich などの設定については別章を参考にして下さい。

パッケージを利用する場合、依存関係という問題があります。これは、あるパッケージをインストールするために、すでに存在しなければならないパッケージがあるという問題です。RedHat や Debian GNU/Linux では依存関係を満たしていない場合には注意をしてくれます。

ここでは RedHat と Debian GNU/Linux の場合における、pvm と mpi, lam の例を紹介しましょう。

RedHat の場合

RedHat にはソースとバイナリの 2 種類のパッケージが存在します。ソースの方にはファイル名に srpm が含まれ、バイナリの方には、i386 などのアーキテクチャーを示す記号が含まれています。ソースパッケージはユーザが自分の環境に確実に合ったバイナリパッケージを作成してからインストールしいと思った場合に、利用するものです。ただしコンパイルが必要となるので時間がかかります。バイナリパッケージはコンパイルがなされたファイルがまとめられたもので rpm コマンドを使って簡単にインストールできるものです。

rpm の主要コマンドを表 1.2 にまとめます。

表 1.2: rpm コマンドでの主要なオプション

コマンド	説明
rpm -ivh xxxx.i386.rpm	インストール
rpm -Uvh xxxx.i386.rpm	アップデート
rpm -e xxxx.i386.rpm	削除
rpm -qvp xxxx.i386.rpm	ファイルリスト
rpm -qip xxxx.i386.rpm	パッケージ情報
rpm -q xxxx.i386.rpm	インストール済かの確認
rpm -K xxxx.i386.rpm	パッケージの検証

表 1.3 に使用すると便利な rpm 管理ツールを示します。

それでは pvm, mpich, lam をインストールしてみましょう。

<http://www.redhat.com/apps/download/> から各パッケージをダウンロードしてきましょう。

表 1.3: rpm 管理ツール

ツール名	説明
rpm2cpio	RPM パッケージから特定のファイルを取り出す
rpmfind	依存関係のあるファイルを収集する
gnorpm	RPM パッケージを扱う GUI ツール
alien	コンバートツール

```
ftp://ftp.redhat.com/pub/redhat/redhat-6.2/i386/RedHat/RPMS/pvm-3.4.3-4.i386.rpm
ftp://ftp.redhat.com/pub/redhat/powertools/6.2/i386/i386/mpich-1.2.0-5.i386.rpm
ftp://ftp.redhat.com/pub/redhat/redhat-6.2/i386/RedHat/RPMS/lam-6.3.1-4.i386.rpm
```

各パッケージのインストールを行いましょ う。

```
# rpm -ivh pvm-3.4.3-4.i386.rpm
# rpm -ivh mpich-1.2.0-5.i386.rpm
# rpm -ivh lam-6.3.1-4.i386.rpm
```

Debian GNU/Linux の場合

Debian GNU/Linux の場合もほとんど RedHat と同様です。パッケージをインストールするの コマンドは dpkg です。

表 1.4 に主な dpkg コマンドをまとめます。

表 1.4: dpkg コマンドでの主要なオプション

コマンド	説明
dpkg -i xxxx.deb	インストール
dpkg -l xxxx.deb	パッケージ情報

パッケージを取得しましょ う。

```
ftp://ftp.debian.or.jp/debian/dists/woody/main/binary-i386/devel/pvm_3.4.2-5.deb
ftp://ftp.debian.or.jp/debian/dists/woody/main/binary-i386/devel/pvm-dev_3.4.2-5.deb
ftp://ftp.debian.or.jp/debian/dists/woody/main/binary-i386/devel/mpich_1.2.0-3.deb
ftp://ftp.debian.or.jp/debian/dists/woody/main/binary-i386/devel/lam2-dev_6.3.2-6.deb
```

パッケージをインストールしましょ う。

```
# dpkg -i pvm_3.4.2-5.deb
# dpkg -i pvm-dev_3.4.2-5.deb
# dpkg -i mpich_1.2.0-3.deb
# dpkg -i lam2-dev_6.3.2-6.deb
```

これで終了です。

ただし、最近開発された `atp-get` コマンドは非常に便利なコマンドです。依存関係を判断し、必要なファイルを自動でダウンロード＋インストールしてくれます。削除の時も同様です。依存関係が複雑なパッケージは削除が非常に面倒です。そんな時には `apt-get` です。システムのバージョンアップの際にも `apt-get` を使用しましょう。

まず、`/etc/apt/source.list` にファイルを取得する方法場所を記しておきます。

```
deb ftp://ftp.debian.or.jp/debian-jp/ woody main contrib non-free
deb ftp://ftp.debian.or.jp/debian-jp/ woody-jp main contrib non-free
deb ftp://ftp.debian.or.jp/debian-non-US/ woody/non-US main contrib non-free
e
```

上記のファイルは `ftp` 経由で `ftp.debian.or.jp` から `woody` バージョンのソフトをダウンロードして使用することを意味しています。

下記にまとめた `apt` コマンドを使用することができます。

表 1.5: 主要な `apt` コマンド

コマンド	説明
<code>apt-get update</code>	パッケージ情報のアップデート
<code>apt-get install hogehoge</code>	hogehoge というパッケージのインストール
<code>apt-get remove hogehoge</code>	hogehoge というパッケージのアンインストール
<code>apt-get -f dist-upgrade</code>	依存関係を考慮しながら現在あるすべてのパッケージをバージョンアップする
<code>apt-get -s update</code>	バージョンアップのテストを行う
<code>apt-cache search hogehoge</code>	hogehoge に関連するパッケージを探索する
<code>apt-cache show hogehoge</code>	hogehoge というパッケージの詳細を表示する

これで `pvm-dev` をインストールしてみましょう。

```
# apt-get install pvm-dev
```

これで `pvm-dev` だけでなく、依存関係も処理して `pvm` もインストールされます。

さらに Debian には便利なパッケージが用意されています。

`task-parallel-computing-dev` と `task-parallel-computing-node` がそれです。前者は並列計算環境を構築するのに必要なアプリケーション (`pvm` や `mpich` など) が含まれています。後者は各ノードに必要なライブラリのパッケージです。

その他、細かいオプションなどやインストールする際の技術は数多くありますが、それに関連した図書や雑誌での記事も多く見られます。詳しくはそちらを参考にして下さい。

Windows でプログラミングしたいんですけど??

先にも述べたとおり、クラスタにて数値計算を行わせる場合には Windows で計算を行わせなければならない理由がこれと見当たりません。しかしながら、日頃使い慣れた Windows 上でどうしても PVM や MPI を利用したい人のためにいくつかライブラリが開発されています。残念ながら筆者は全く Windows に疎いため、ここでは、その情報源を紹介します。ほとんどの場合がバイナリをダウンロードすればインストール可能なようになっており、これらは VC++ を利用することが前提となっています。

- PVM

PVM はバージョン 3.4 からは Windows 95 および NT 版も用意されている。

<http://www.netlib.org/pvm3> からバイナリファイルを取得できる。

- MPI 関連

- mpich

本家の mpich のチームが NT での稼働にも対応した実装です。

<http://www-unix.mcs.anl.gov/mpi/mpich/> を参照して下さい。

ちなみに Linux と NT の混在環境でも mpich は使用可能です。

<http://www.lfbs.rwth-aachen.de/users/joachim/MP-MPICH/>

- WMPI

WMPI はポルトガルの Coimbra 大学のグループが MPICH を Win32 へ移植したものです。最大の特徴は mpich のバージョンに対して開発維持が現在もされていて mpich が稼動している Windows 以外のマシンとも共存ができる点です。現在バージョンが 1.5 となり、開発が Coimbra 大学から Critical Software

(<http://www.criticalsoftware.com/wmpi/home/index.html>)

へ移ったようです。

- MPICH/NT

MPICH/NT は Mississippi 州立大学の High Performance Computing Lab が開発している NT 上の mpich です。 <http://WWW.ERC.MsState.Edu/labs/hpcl/projects/mpi/mpiNT-download.html> にて情報とダウンロードができるようになっています。現在、バージョンが 0.92 であるが開発は中止されているようです。

- MPI Pro

MPI Pro は MPI Software Technology によって開発されている MPI の実装で Windows NT 用や Linux 用など幅広く開発しています。情報源は <http://www.mpi-softtech.com/> です。

Linux バージョンはソースのダウンロードがフリーのようです。

■ PC クラスタの構築 –ハードウェア編–

□ ハードウェアにこってみる

前章では、PC クラスタを簡単に構築してみました。Linuxなどをインストールした経験のある人であれば1日で完成するかもしれません。これで、並列計算がスタートできます。ただし、これは、市販の車でレースに出場するようなものです。もう少し、自分のマシンをチューンアップしてみましょう。まずは、ハードウェアからです。

PC クラスタのハードウェアを大雑把に整理すると、PC 単体とネットワークからなります。そのため、ハードウェアのチューンアップはPC 単体とネットワーク周りということになります。

PC 単体

PC 単体の構成はPC クラスタを構築する場合、次のようなものになります。

- CPU
- マザーボード
- メモリ
- ハードディスク
- ビデオカード
- ケース
- FDDドライブ
- CD-Romドライブ
- (マウス, キーボード)
- (ネットワークカード)

最近はサウンドカードやDVDを搭載したPCも増えてきていますが、数値計算などを主に使用する場合には、必要ありません。

1. メモリ

まずはメモリについて述べておきます。

メモリはCPUやハードディスクに比べて、価格の低下も性能もあまり変化がありません。価格に至っては高くなる傾向が最近あります

これまで市場ではPC/100 SDRAMというメモリが主流でした。データ幅が64Bitで100MHzでアクセスできるものです。これが次第に、PC133 SDRAMに変わりつつあります。

今年このメモリが大幅に変わる予定でした。それはIntelが新たなメモリである、Direct SDRAMに移行すると宣言し、マザーボードもそれに対応するものを発表したからです。

Direct SDRAMは転送速度が266MHz, 333MHz, 400MHzと非常に高性能なものです。

しかしながら、メモリの変更は行われなかったのです。

その理由としては、Intelの発売したマザーボードにいくつもの不具合が発見されたこと。まだまだDirect SDRAMの製造コストが高かったことなどがあげられます。

しかし、HPCを行っているものにとって不運なのは、このような Intel のつまづきによって、まだまだ 現行のメモリが続いていきそうだということです。これまで Intel は Pentium IV では Direct DRAM にすると言っていましたが、つい先日そのロードマップが変更され、SDRAM もサポートすることになったのです。

そうこうするうちに ADM は DDR SDRAM という別のメモリを次期のメモリとすることにしてしまいました。

ユーザ、特にクラスターのユーザとしては、高性能な部品が安定して供給されるのが望ましいわけですが、いきなり状況は混沌としてきました。

2. マザーボード

Linux において問題となるのは、マザーボードのチップセットです。新しいチップセットを使用しているようなマザーボードでは問題が生じる場合があります。長く使用されている 440BX という Intel のチップセットであればまず問題はありません。

しかしながら、CPU の進化に対して、440BX のチップセットは思った以上に長生きだったので、今年はメモリのところでも述べたように、Intel 820 がデビューする予定だったのですが、大きくくずれてしまいました。それに付け込むかのように Athlon の攻勢がありました。そのため、混沌感がひろがっています。Intel は失敗した Intel 820 の代わりに Intel 815 というちょっと中途半端なものを出してきました。これが、果たして 440BX の後継となるのでしょうか。

ソケットに関してはこれまでいろいろな形状がありましたが、Intel のものに関しては FC-PGA に落ち着きそうです。

3. CPU

CPU はパソコンの中心部です。最近、いろいろな選択枝が選べるようになりました。大まかに言えば Intel などの x86 系のプロセッサとコンパックのアルファプロセッサですが、Power PC など注目浴びるようになってきました。

これまで性能の高いクラスターと言えばほとんどがアルファプロセッサでした。しかし、今年、Intel や ADM は 1GHz の壁を越え、コストパフォーマンスの良さから Intel のものでも高性能なクラスターが作られるようになってきました。

x86 系のプロセッサも最近では事実上 Intel と AMD のものしかありません。最も難しい選択はこの中でどのプロセッサを選択するかということです。例えば、Intel では Pentium III と Celeron があり、AMD には Athlon と Duron がありますが、これらをクラスターの性能を考慮して決定することは極めて難しくなっています。

最近では AMD が非常に攻勢をかけていて、クロック対決では常に Intel の先を行っています。3D Now の技術を使用すると Athlon は非常に高性能という報告もあります。

(<http://japan.cnet.com/News/2000/Item/000526-5.html?rm>)

一方、Intel はまだまだブランド力があります。

いずれにせよ、今後も CPU に関してはその性能の向上は続くでしょう。また、最新でなければ、非常に価格が安くなっています。

4. ケース

ケースによって Linux が動かなかったり動いたりする場合があります。ただし、稼働させている時間が長かったり、CPU などの発熱が多い場合が考えられるので、冷却の十分行われるケースが必要です。ファンのちゃんと付いているものを購入しましょう。

大きさは、マザーボードの形式と拡張性によって決定されます。拡張性がほとんど必要無い場合には、最近では非常に小さいものも存在しています。

その他には電気容量や防音の問題があります。また、実はデザインが大きな問題であるかもしれません。というのも苦労してクラスタを構築してもケースが悪くて大したものを感じられない場合もあるからである。

最近では、場所の問題や見ためからラックタイプのものも注目されるようになってきました。ラックタイプのものは通常のケースと比較すると、コストがかかりますが、ラックは他の部品と異なり、すぐに性能が下がることがなく、長く使用は可能です。

5. ネットワークカード (Ethernet 100 Base TX)

現在広く使用されているのが Ethernet 100 Base TX でしょう。Linux によりクラスタシステムを構築する際に最も問題となるのが、このネットワークカードです。製造会社の問題ではなく、そのカードに使用されているチップセットの種類によって安定的に使用できるかできないかが問題となります。

ネットワークはハードウェアと OS の間にあるドライバの性能が大きく影響します。性能の良いネットワークドライバを使用したいところです。Linux OK と表示されているネットワークカードでもこのドライバの性能のために不安定になる場合もあるので要注意です。

さらに Ethernet にも最近、Giga bit 級の製品が見られるようになってきました。もちろん、特別なドライバが必要です。

Myricom 社が開発している Myrinet は多くのクラスタシステムで使用されている Giga bit ネットワークとスイッチングハブのシステムです。これは Ethernet とは異なるものです。特別なドライバを用意することで極めて高速に通信ができるようになります。

6. ハブ

複数台の AT 互換機をネットワークで接続するためにはハブが必要となります。ここ数年で最も値段が下がったものの一つにスイッチングハブがあります。

今後1年で Giga のスイッチングハブも値段が下がっていくことでしょう。

7. ハードディスク

ハードディスクはここ数年で最も値段の激減したものの一つです。また、そのアクセス速度や容量も増大しました。10G や 20G のハードディスクが標準となっていて数年前では想像もつかないことです。

AT 互換機で現在利用されているハードディスクには大きく分けて2種類があります。IDE と呼ばれるものと SCSI と呼ばれるものです。それぞれの規格での理論性能を次表にまとめてみます。

表 1.6: データ転送理論性能

Standard	SCSI/IDE	Transfer Speed [MB/sec]
Fast SCSI(SCSI2)	SCSI	10
Ultra SCSI(SCSI3)	SCSI	20
Ultra ATA/33	IDE	33
Ultra Wide SCSI	SCSI	40
Ultra ATA/66	IDE	66
Ultra 2 SCSI	SCSI	80
Ultra ATA/100	IDE	100
Ultra 160 SCSI	SCSI	160
Ultra 320 SCSI	SCSI	320

データはメモリとハードディスクの間をいったりきたりしています。このデータ転送には2種類あります。CPUがデータ転送を制御する方法であるPIO転送とCPUの代わりにデータ転送を制御する特別なバスマスタと呼ばれるものを利用して行うDMA転送です。もちろんDMA転送の方がCPUパワーを損なうことなく高速に転送できます。これまでSCSIはDMA転送でIDEはPIO転送であったので、データ転送速度には差があり、そのため多少高額でもサーバー機ではSCSIが使用されていました。

それに対して最近、IDEでもDMA転送を行う技術が開発され、使用が普通になりました。それがUltra ATA/33やUltra ATA/66です。もちろんこれを使用するにはそれに対応したマザーボードもしくは拡張ボードが必要となります。またLinuxではカーネルなどの対応が必要となります。

現在のところコストパフォーマンスはIDEの方が比較にならないほどに良好です。

また、最近の話題ではRAIDと呼ばれる複数台のハードディスクを分散書き込みを行うことにより、高速にアクセスしたり、障害に強くしたりする手法があります。これはハードウェア的な処理とソフトウェア的な処理があります。

ハードディスクヘデータの書き込みが多いクラスタでは、最近のIDEの大容量化とコストの低下の傾向から大きなキャッシュを有するRAIDディスクを使用したいところです。

また、最近の選択のポイントとして容量や性能だけでなく、静音という点も注目されるようになりました。

8. ビデオカード

LinuxではウィンドウシステムとしてXが動作します。XではX serverとハードウェアが問題となります。フリーのXF86や商品のAccelarateXがビデオカード上のチップに対応しているかが問題となるのです。また、最近話題の3DにはLinuxでは対応しているサーバーが存在しないため高価なビデオカードは必要ありません。さらにクラスタでは、telnetやsshでの操作で行うことが多くウィンドウシステムが必要ない場合が多く、そのような場合にはビデオカードは何を選んでも問題はありません。

9. マウスとキーボード

LinuxでXを使用する場合にはマウスはPS/2の3ボタンマウスを用意したいところです。2ボタンでも3ボタンとしてエミュレートして使用することが可能です。

キーボードは日本語版や英語版、大きさの違い、キーバインディングの違いなど様々なものがありますが、ユーザーが好みで選択すればよいと思います。

キーボードもマウスもシステム構築後はメンテナンスの際に使用するのみですから、1セットでよいでしょう。

最近、USB接続用のものが普及してきました。Linuxも最新のカーネルではUSBに対応しています。

ネットワーク

ネットワークはPCクラスタにとって非常に重要な部分の一つです。一つのPCに何枚のネットワークカードを挿入するかといった問題も重要です。ネットワークの性能はハードウェアだけでなく、使用するドライバにも大きくよってしまいます(Scoreの章参照して下さい。)。

規模が大きくなるとハブやネットワークをどのようなものをいくつ用意し、どのようなトポロジで配置するかも大きな問題になります。

Ethernet に対する代替の有力候補はミリネットです。これはミリコム社 (<http://www.myricom.com>) のギガビットのネットワークで様々なクラスタで利用されています。ミリネットに関する資料やドライバなども数多く存在します。

□ それでは組み立ててみましょう !!

それでは組み立ててみましょう。ここでは IBM PC/AT 互換機を例に説明します。

IBM PC/AT 互換パソコン（以下、AT 互換機）は一般に汎用パソコンとして使用されているもので 1984 年に IBM より発表された PC/AT を元祖としています。IBM はこの機器のデバイスを全て自社で開発するのではなく、基本使用を公開し、拡張ボードを使用することによって製品化しました。この特徴により、各デバイスを様々な会社が開発し、拡張ボードを利用して様々な周辺機器を開発することにより、AT 互換機の性能は向上し、かつ、価格は下がり、また、応用範囲も広がるという特徴を持つにいたりました。そのため、現在では、AT 互換機は価格は安くても性能的には十分満足でき、かつ、安定性にも優れているのです。

下の手順で AT 互換機を組み立てます。

1. ケースをあける



図 1.11: ケース

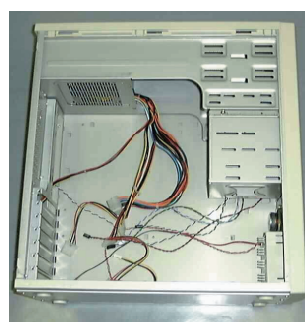


図 1.12: ケースを開けた状態

2. CPU の設定の用意

本講習会では Celeron(Socket370) にアダプタを付加して Slot 1 に設置する。

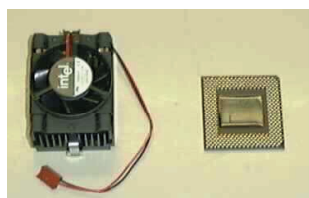


図 1.13: CPU(celeron)



図 1.14: アダプタ

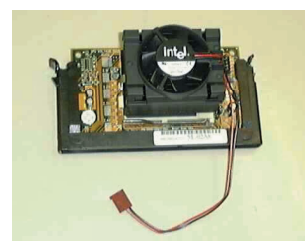


図 1.15: CPU を組み立てた状態

3. マザーボードに CPU, メモリを設置

4. マザーボードをケースに設置

マザーボードの下にはスペーサーをつける。マザーボードによってはアクセス周波数をジャンパーピンで設定するものもある。本講習会で使用するものは BIOS によって設定する。

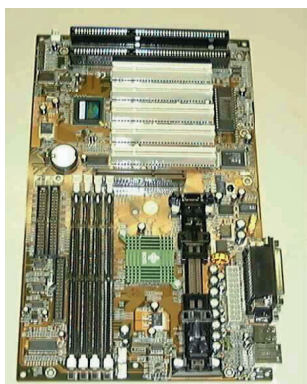


図 1.16: マザーボード
(ATX,slot1)

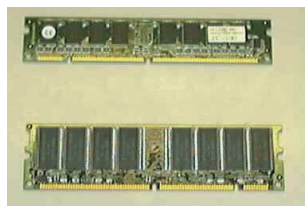


図 1.17: メモリ

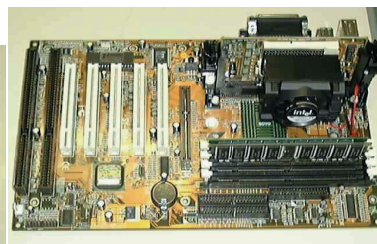


図 1.18: マザーボードにメモ
リと CPU を設置



図 1.19: マザーボードとケース

5. フロッピーディスクドライブ, CD-Romドライブの設置



図 1.20: フロッピードライ
ブ



図 1.21: CD-Romドライブ

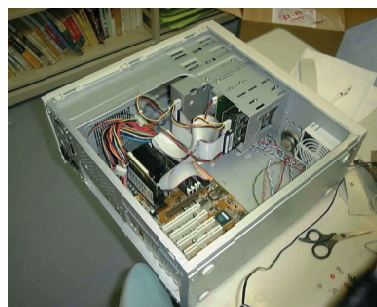


図 1.22: フロッピーと CD-
Romドライブを装着した
状態

6. ビデオカード, ネットワークカードの設置

7. BIOS の設定

BIOS により基本周波数や CPU の動作周波数, メモリの種類, 起動ディスクの選択などを行う。

8. ネットワークへの接続

Ethernet をネットワークカードとハブに接続しクラスタを構築する。

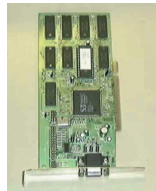


図 1.23: ビデオカード



図 1.24: ネットワークカード



図 1.25: ネットワークとビデオカードを装着した状態

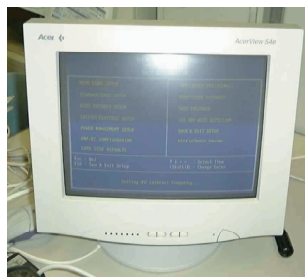


図 1.26: BIOS 設定画面



図 1.27: スイッチングハブ

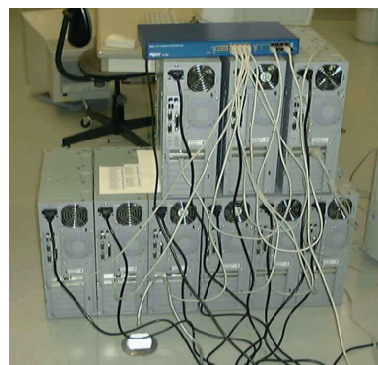


図 1.28: 完成したクラスタ

■ PC クラスタの構築 –ソフトウェア編–

□ ソフトウェアにこってみる

PC クラスタを構成するソフトウェアは図 1.29 に示す通りです。

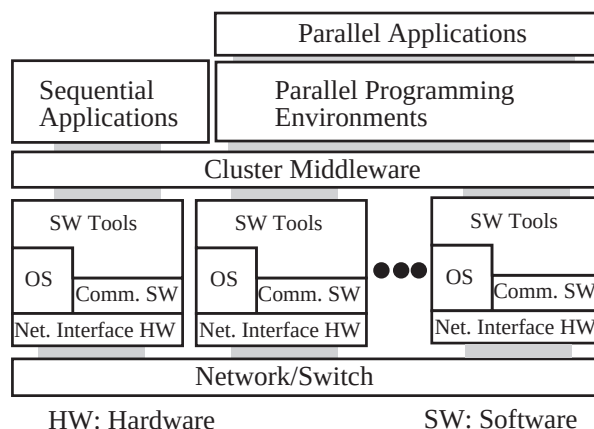


図 1.29: PC クラスタの構成

ネットワークの上に、PC を構成するソフトウェア群、クラスタとして機能するためのミドルウェア群、その上に逐次のアプリケーションもしくはパラレルのアプリケーションが載ることになります。

オペレーティングシステム (OS) の種類

ここまでクラスタの OS として Linux を中心に説明してきました。しかしながら、最近では、多くの商用 OS がフリーになってきて、選択肢も広がっているといえます。

Linux 以外の OS としては、次のようなものがあげられます。

- Free BSD <http://www.freebsd.org/>
- Net BSD <http://www.netbsd.org/>
- Solaris <http://www.sun.com/solaris/>
- UnixWare <http://www.nihonsco.co.jp/freeUW7/>
- BeOS <http://www.be.com/products/free/bveos/>
- Darwin <http://www.publicsource.apple.com/projects/darwin/>

この中でも特に BSD 系と Solaris はネットワークの性能が優れているために、クラスタの OS としては適しているかもしれません。

カーネルの再構築

特に Linux において、カーネルの再構築は必須事項です。その理由としては

- 不要なデバイスドライバや機能を付加しておく、そのためにメモリ空間の一部が占有されてしまう

- システムの起動に時間がかかる
- Linux は開発途上の OS であり、特にクラスタで重要なネットワーク周りの開発はこれからであるため、最新のバージョンのものを利用する効果が高いと考えられるため
- PC クラスタで頻繁に使用するソフトウェアは特定のバージョンの OS と依存関係があり、かつ、特定の機能を必要とするため。

などがあります。

必要なモジュールのみでカーネルを作成することで必要メモリの少ないカーネルを作成することが可能です。

通常のカーネルの再構築の手順は以下の通りです。

1. ソースの入手と展開
2. menuconfig などによるコンフィギュレーション
3. make
4. パッケージ化
5. パッケージのインストール

最近では、Vine や Debian GNU/Linux のようにカーネルさえもパッケージとなっているものも増えてきました。

また、ここでは詳しく述べませんが、再構築には、静的に構築を行う手順と動的に行う手順があります。

開発環境

Linux は近年発達してきたために、Linux 上ではどのようなコンパイラがあるのかというのもよくある質問です。PGI 社製 <http://www.softek.co.jp/SPG/compiler.html> や VATS/f90 <http://www.psrv.com/lxf90.html> , および Absoft 社製 <http://www.absoft.com/> などは定評のあるところです。富士通製 <http://www.fqs.co.jp/fort-c/> も OpenMP に対応し、HPC での技術を Linux 上で稼動する形でクラスタにポーティングしつつあります。

mpich のソースからのインストール

mpich をソースからインストールしてみましょう。

まずはソースを以下のサイトから入手します。

<ftp.mcs.anl.gov/pub/mpi/mpich-1.2.0.tar.gz>

mpich のインストール

1. ソースファイルを展開する。mpich ができる。
2. mpich に移動し、configure, make をする。
3. make PREFIX=/usr/local/mpich install を実行しインストールする。(これは /usr/local/mpich にインストールする例)

詳しくは MPI の章をご覧ください。

管理するために必要なツール

クラスタを管理するためにいくつか必要なツールがありますが、NFS、NIS、NTP は良く使用されるアプリケーションです。

- NFS(Network File System)

NFS はサーバで供給されるファイルをクライアントで共有するものです。ユーザのディレクトリを NFS でクライアント側で共有することで、どのクライアントでも共通のファイルを参照することになります。

NFS を利用しない場合には、クライアントすべてにファイルを転送しなければならず、非常に面倒な作業が発生します。

一方、ディスクアクセスが頻繁に生じるようなアプリケーションの場合には、NFS がボトルネックとなるような場合があります。高速な NFS が望まれます。また、ネットワークカードを 2 枚利用することで、並列処理でのデータ交換と NFS のデータを分離する方法もあります。

- NIS(Network Information Service)

NIS を利用することにより、ユーザは共通のログイン名、パスワード、グループ名などを使用することができるようになります。NIS もサーバ・クライアント型のシステムです。

- NTP(Network Time Protocol) MPICH などを利用する場合、make ファイルなどを利用するために、タイムスタンプが各ノードで異なると問題が生じてしまいます。

この問題を NTP です。このシステムもサーバ・クライアント型で、クライアントの時間をサーバの時間に合わせます。

プロファイラ

並列アプリケーションの作成における難しい点の一つはデバッグが簡単に行えないという点です。各ノードで間違い無く動くアプリケーションでも通信が発生すると正常に動作しない場合があるからです。しかも複数ノードを効率よくデバッグを行うためにデバッガは GUI 表示できることが必須です。TotalView <http://www.dolphinics.no/> は広く使用されている GUI ベースのデバッガです。一般的に使用されている言語、メッセージパッシングライブラリに対応しています。

一方、どの程度の通信遅延がどこで生じているか、あるいはどの関数でどれだけの時間がかかっているのかを知ることはプログラムを高速化するために必要なこととなります。それらを教えてくれるのがプロファイラと呼ばれるツールです。フリーウェアでは MPICH に含まれている MPE や Paradyn <http://www.cs.wisc.edu/paradyn> が比較的有名です。市販のものとしては Vampir <http://www.pallas.de/pages/vampir.htm> が広く使用されています。

リソースマネージャー

構築したクラスタを有効に利用するためには Resource Management and Scheduling (RMS) の導入が必要になります。RMS はプロセスの配分、ロードバランスの調整、複数タスクのジョブスケジューリングを行うため、ユーザーの助けを必要とせず、各ノードが自動的に効率良く使用されることになります。CONDOR <http://www.cs.wisc.edu/condor/> や DQS <http://www.scri.fsu.edu/pasko/dqs.html> はフリーウェアして代表的なものであり、LSF <http://www.platform.com/> や CODINE <http://www.genias.de/products/codine/> は高価ですが、非常に高機能な市販のシステムです。

ジャーナリングファイルシステム

現在、Linux は Ext2 というファイル形式が広く使われています。ジャーナリングファイルシステムは次世代のファイルシステムです。

突然の停電やなんらかの障害で、システムがダウンした際には、早い復旧が望めます。また、これまでの Ext2 では、突然の障害には弱いため、ファイルが壊れてしまうことが多々ありました。特に台数の多いクラスタシステムでは特にこれらの問題がネックとなります。

ジャーナリングファイルシステムはデータベースで使用されていた技術をファイルシステムにも適用したものです。データベースでは耐故障性を得るためにトランザクション単位で処理が行われます。ジャーナリングファイルシステムではファイルシステムのディレクトリ構造や管理情報に対してデータベースと同様のトランザクション処理を行います。ファイルやディレクトリに対する更新履歴 (ジャーナル) を別に保存することで、たとえクラッシュが発生しても電源が落ちる前の一貫性を保った状態に戻せるというわけです。また、ファイルシステムの一貫性が保証されているため、クラッシュ後にファイルシステムをチェックする必要がありません。これが、ダウン後からの復旧が早い理由です。

ジャーナリングファイルシステムの実装はいくつかありますが、その中で現在、最も完成度が高いものが、ReiserFS <http://devlinux.com/projects/reiserfs/> でしょう。

Linux ディストリビューションの一つである SuSE Linux は時期バージョンは ReiserFS を標準サポートすると表明しています。また、Kondara MNU/Linux Server でも ReiserFS を採用しています。

■ 実際の運用

本章では一通りインストールからチューンアップまでを考えてきました。

インストールの際は、解説したように 1 台、1 台に OS やアプリケーションをインストールする方法がありますが、今後は、クラスタシステムとしてインストール可能なアプリケーションが増えてくることでしょう。Alinka LCM(<http://www.alinka.com/download.htm#lcm>) や FAI(<http://www.informatik.uni-koeln.de/fai/>) はそのようなシステムです。

前述した通り、NFS や NIS などを運用した方が管理が楽になります。そのため、サーバが必要となります。また、Beowlf 型のクラスタではユーザのログインポイントが必要となります。そのため、サーバーとなりログインポイントなるマシンを 1 台用意した方が管理が簡単です。このマシンには計算などの処理はさせないようにします。サーバーの外側はグローバルアドレスを有し、サーバの内側とクライアントはローカルアドレスを持つように設定します。

計算させるノードはディスクレスにするのも手です。これは別章を参考にして下さい。

インストール後は実際の運用になります。

まず最初に行うのは、起動です。マシンの数が多い場合には、いちいち一つずつ手動で起動しては面倒ですので、Wake up Lan の機能を使用して、ネットワーク経由で起動します。

実際にアプリケーションを動かす場合には RMS を活用しましょう。

停電の場合など不意の事故に備えることが大事です。

万が一、ハードウェアの故障に備えてクラスタを構成する部品の予備を準備する必要があります。コモディティハードウェアは性能やコストで良い面がある半面、そのサイクルが早く、数ヵ月で同一の部品を手に入れることができなくなることもあるからです。

■ PC クラスタ再考

□ 共有メモリ型と分散メモリ型

一般に PC クラスタの場合、共有メモリ型の PC をいくつか含んだ分散メモリ型の並列計算機となります。メモリを持たない CPU が複数バスにつながりメモリからみて対称に配列しているものを SMP(Symmetric MultiProcessor) と呼ばれます。

SMP 内では thread や Open MP(<http://www.openmp.org>) を利用したプログラミングが効率的です。

分散メモリでは、pvm や mpi などによるメッセージパッシングによりプログラミングすることになります。

□ PC クラスタのセキュリティ

PC クラスタのセキュリティを考えた場合、クラスタシステムの外部からアクセスできるマシンは 1 つにした方が以下のいくつかの理由により無難です。すなわち、rsh や nfs といったセキュリティに問題があるサービスを利用するからです。これらのサービスは閉じたネットワーク内でローカルで行うべきです。

また、ログインポイントにアクセスしたりデータをダウンロードする必要性が多々ありますが、そのような場合には、セキュアなツールを使用すべきです。telnet などではなく ssh(<http://www.ssh.org/>) などを利用しましょう。

□ Single System Image (SSI)

Single System Image (SSI) はクラスタのような複数台なるシステムを 1 台のマシンのように使えるシステム概念です。この SSI に望まれるサービスは以下のようなものがあります。

- 単一のエントリーポイント
- 単一のファイル階層
- 単一の制御ポイント
- 単一のバーチャルネットワーク
- 単一のメモリ空間
- 単一のジョブマネージャー
- 単一のユーザインターフェース
- その他

筆者は完全な SSI の実現は難しいと考えています。例えば、これまでに自動コンパイラの研究が盛んに行われてきましたが、OpenMP のように、結局、並列を明示する tag を挿入する方法に落ち着いてきています。すなわち、完全に複数のマシンを 1 台のマシンと同等に使うのは便利そうですが、その実現方法が現実的ではありません。上記の項目のいくつかを達成し、ユーザが使いやすくなればそれで良いのではと考えています。

■ 最近の動向

- CPU はどうなるのか？

CPU の種類はいくつかありますが、基本的には Intel の 32bit の CPU, i386 を基本としていました。いよいよ今年には Intel から次世代の 64bit OS, Itanium が発売されます。すでに Linux は 64bit に対応していますが、どうなるでしょうか。HPC の世界はどうなるのか。非常に興味深いところです。

参考 <http://www.linux64.org/>

CPU は急激な勢いで高性能化を遂げ、汎用 CPU は Intel と AMD の製品に収束しつつあります。しかし、その高性能化の代償として回路が複雑になり、消費電力や発熱が大きく増加するという弊害が生じたわけです。その問題を解決したのがトランスメタ社 <http://www.transmeta.com/Crusoe> です。この CPU は演算性能を高める一部の機能をソフトウェアで代替することによってトランジスタ数を大幅に削減し、高性能と低消費電力を両立しているのです。

早く実物が見たいものです。

- MB のチップセットは メモリはどうなる？

ハードウェアの項目にも書きましたが、意外にしぶとく 440BX のチップセットがそのまま残りました。いよいよ今年は転換機でしょうか。

チップセットの問題は、CPU やメモリと密接な関係があり、かつ、チップセットの主流が変更されれば、ネットワーク関係への影響も必死です。特に、数値計算の分野では、メモリのアクセス速度の向上が望まれているわけですが、値段や性能が足踏みしています。この 1 年でどうなるでしょうか。

- ネットワークはどうなるのか？

マッキントッシュにも標準で搭載されるようになり Giga がいよいよ普及してきました。今のところ、ハブの値段がまだまだ高いですが、急速にこれも安くなる雰囲気です。今年は、giga net が普通に使われるようになる元年なのかもしれません。

もう一つのポイントは無線 Lan です。

これも最近普及してきた製品です。クラスタ内を無線でつなぐことはないでしょうが、複数のクラスタ間を無線で使うことはあるかもしれません。

一方ミリネットは Myrinet 2000 へ進化し 2Gb となりました。

- グローバルコンピューティング

最近、特に注目されているのがグローバルコンピューティングです。

グローバルコンピューティングはインターネット上のあらゆるコンピュータシステムを一つのシステムとして利用しようというものです。Globus や Ninf などといったシステムがこれまでに提案されてきました。

しかし、ここにきてユニークなシステムが登場しました。Napster (<http://www.napster.com/>) や Gnutella (<http://gnutella.wego.com/>) と呼ばれるシステムです。これは、全世界のコンピュータのファイルをあたかも共有しているように見えるシステムです。最初は、MP3 のためのシステムでしたが、特に Gnutella はそのファイル形式にこだわっていません。これから、Music と Netscape が世界を変えたのと同様に、世の中を変えていく可能性を匂わせています。

■ 参考文献

- 1) Beowulf Underground <http://www.beowulf-underground.org/>

テキスト

- 2) Cluster Computing White Paper <http://www.dcs.port.ac.uk/~mab/tfcc/WhitePaper/>
- 3) Beowulf How To <http://www.linuxdoc.org/HOWTO/Beowulf-HOWTO.html>
- 4) G. F. Pfisher In search of CLuster Prentice Hall 1998
- 5) R. Buyya High Performance Cluster Computing: Architectures and Systems Volume 1 Prentice Hall 1999
- 6) R. Buyya High Performance Cluster Computing: Programming and Applications Volume 2 Prentice Hall 1999
- 7) T. Sterling, J. Salmon, D. J. Becker and D. F. Savarese How to Build a Beowulf The MIT Press 1992
- 8) B. Wilkenson and M. Allen Parallel Programming Prentice-Hall 1999
- 9) 青山チューニング技法虎の巻 golgo13@jp.ibm.com 1999
- 10) 青山並列プログラミング虎の巻 golgo13@jp.ibm.com 1999
- 11) 湯浅他はじめての並列プログラミング 共立出版1998

国際会議

- 12) SC2000 <http://www.sc2000.org/>
- 13) PDPTA <http://ceng.usc.edu/~hjin/cc-tea2000.html>
- 14) Cluserter 2000 <http://www.tu-chemnitz.de/informatik/RA/cluster2000/index.htm>
- 15) CCGrid 2001 <http://www.ccgrid2001.qut.edu.au/>

研究会 および メイリングリスト

- 16) 超並列計算研究会 <http://www.is.doshisha.ac.jp/SMPP/>
- 17) IEEE tfcc <http://www.dgs.monash.edu.au/~rajkumar/tfcc/>
- 18) HPC メイリングリスト http://www.softtek.co.jp/hpc_mlist.html

MPIによる並列プログラミングの基礎

同志社大学 大学院 工学研究科

渡邊 真也

sin@mikilab.doshisha.ac.jp

■ はじめに

ここでは、MPIを用いた並列プログラミングについて初歩的な基礎事項を含めて解説する。MPI(Message Passing Interface)とは、分散メモリ環境における並列プログラミングの標準的な実装である。その名の通りメッセージパッシング方式に基づいた仕様であり、MPIの仕様に準じた実装ライブラリは、複数存在する。その中の幾つかはフリーで配布されており、UNIX系を中心としてWindows, Macとほぼ全てのOS,アーキテクチャに対応している。そのため、どのようなクラスタ環境においてもMPIはフリーで 사용할 ことができる。以下では並列プログラミングを始めようとする初心者の方を対象として、MPIを用いた簡単なプログラム作成への手引き書として解説を行っていく。

■ 並列プログラミングの基礎知識

具体的なプログラミングの説明に入る前に、MPIを中心とした幾つかの基礎事項について説明する。

□ メッセージパッシング方式とデータ並列方式

クラスタのような分散メモリシステムにおいて、並列化インタフェースは大きく2つに分類される。一つは、**メッセージパッシング方式**であり他方が**データ並列方式**である。以下、MPIが用いているメッセージパッシング方式について説明する。

メッセージパッシング方式

メッセージパッシング方式とは、プロセッサ間でメッセージを交信しながら並列処理を実現する方式である。並列処理では、複数のプロセッサが通信を行いながら同時に処理を進めていく。そこで問題になるのがプロセッサ間の通信であるが、メッセージパッシング方式ではプロセッサ間での通信をお互いのデータの送受信にて行う。

そのため、

- ・プロセッサ i 上でプロセッサ j にデータを送る。
- ・プロセッサ j 上でプロセッサ i からデータを受ける。

が必ずペアとして成り立たなければならない。

メッセージパッシング方式は、分散メモリ環境において現在、最も主流の方式であり、かなり本格的な並列プログラミングが実現することができる。この方式をプログラムで実現するためのライブラリが、**メッセージパッシングライブラリ**である。その最も代表的なライブラリとして MPI と PVM がある。

□ MPI と PVM

現在、分散メモリ型のメッセージパッシングライブラリの代表としては MPI と PVM (Parallel Virtual Machine) があげられる。しかし近年、PVM が劣性に立たされており MPI の急速な伸びが目立つ。以下では、MPI と PVM について概説し、今後の両者の動向についての観測について述べる。

MPI

MPI は Message Passing Interface を意味し、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 仕様である。

MPI の経緯について説明する。MPI の標準化への取り組みは Supercomputing'92 会議において、後に **MPI フォーラム** として知られることになる委員会が結成され、メッセージ・パッシングの標準を作り始めたことで具体化した。これには主としてアメリカ、ヨーロッパの 40 の組織から 60 人の人間が関わっており、産官学の研究者、主要な並列計算機ベンダのほとんどが参加した。そして Supercomputing'93 会議において草案 MPI 標準が示され、1994 年に初めてリリースされた。

MPI の成功を受けて、MPI フォーラムはオリジナルの MPI 標準文書の改定と拡張を検討し始めた。この MPI-2 フォーラムにおける最初の成果物として、1995 年 6 月に MPI1.1 がリリースされた。1997 年 7 月には、MPI1.1 に対する追加訂正と説明がなされた MPI1.2 と、MPI-1 の機能の拡張を行った MPI-2 がリリースされた。MPI-2 の仕様は基本的に MPI-1 の改定ではなく、新たな機能の追加であるために、MPI-1 で書かれたプログラムが MPI-2 をサポートするプラットフォームで実行できなくなるということはない。これらの MPI 文書は、MPI フォーラムのサイト (<http://www.mpi-forum.org/>) から入手可能である。また MPI-1, MPI-2 の日本語訳が MPI-J プロジェクトにより公開されており、<http://www.ppc.nec.co.jp/mpi-j/> から入手可能である。

MPI-2 への主な取り組みとしては、

- ・ 入出力 (I/O)
- ・ Fortran90, C++ 言語向けの枠組み
- ・ 動的プロセス制御
- ・ 片側通信
- ・ グラフィック
- ・ 実時間対応

などが挙げられる。この中で特に注目すべきは動的プロセスの制御の導入である。これまで、PVM では実現されていた機能であるが MPI-1 では実現されていなかった。この機能実現は、単に MPI のプログラミングがより高度なことが実現できるようになったというだけでなく、PVM の優位性が一つ崩れたという意味でも重要である。

MPI にはその仕様に準じた幾つかのライブラリが存在する。実装によっては一部の MPI の関数が使えないものもあるが、ほとんどの主要な MPI 関数は利用することができる。ただし、現在はまだ MPI-2 を完全にサポートした実装はない。MPI がサポートされているシステムは、専用の並列計算機からワークステーション、PC に至るまで幅広い。下に示す表に、フリーに提供されているものとベンダによって提供されている主な実装を紹介する。

[Freeware MPI Implementation]

実装名	提供元
	ホームページ
	サポートされているシステム
CHIMP/MPI	Edinburgh Parallel Computing Centre(EPCC)
	ftp://ftp.epcc.ed.ac.uk/pub/chimp/release/
	Sun SPARC(SunOS 4,Solaris 5),SGI(IRIX 4,IRIX 5), DEC Alpha(Digital UNIX),HP PA-RISC(HP-UX),IBM RS/6000(AIX), Sequent Symmetry(DYNIX),Meiko T800,i860,SPARC,Meiko CS-2
MPICH	Argonne National Laboratory
	http://www-unix.mcs.anl.gov/mpi/mpich/
	MPP では IBM SP,Intel ParagonSGI Onyx, Challenge and Power Challenge,Convex(HP) Exemplar,NCUBE, Meiko CS-2,TMC CM-5,Cray T3D, TCP で接続されたネットワーク上では, SUN(SunOS,Solaris),SGI, HP,RS/6000,DEC Alpha,Cray C-90, その他多数のマシン
LAM	Laboratory for Scientific Computing,University of Notre Dame
	http://www.mpi.nd.edu/lam/
	Sun(Solaris 2.6.1,2.6),SGI(IRIX 6.2-6.5), IBM RS/6000(AIX 4.1.x-4.2.x),DEC Alpha(OSF/1 V4.0), HPPA-RISC(HP-UX B.10.20,B.11.00), Intel x86(LINUX v2.0,v2.2.x)
WMPI	Universidade de Coimbra - Portugal
	http://dsg.dei.uc.pt/wmpi/intro.html
	Windows 95,Windows NT

[Vendar MPI Implementation]

IBM Parallel Environment for AIX-MPI Library	IBM Corporation
	http://www.ibm.com/
	Risc System/6000,RS/6000 SP
MPI/PRO	MPI Software Technology
	http://www.mpi-softtech.com/
	Redhat Linux,Alpha アーキテクチャ,Yello Dog PPC(Machintosh G3 box), Windows NT,Mercury RACE
Sun MPI	Sun Microsystems,Inc.
	http://www.sun.com/software/hpc/
	すべての Solaris/UltraSPARC システム, またそのクラスタ

PVM

MPIはインターフェイスの規定であるのに対して、PVMは実装パッケージそのものである。PVMは、TCP/IP ネットワークで接続された何台ものコンピュータを仮想的に1台のマシンととらえて、並列プログラムを走らせることのできるメッセージ・パッシングの環境とライブラリを提供する。その歴史は、1991年にオークリッジ国立研究所とテネシー大学で開発に始まり、PVMプロジェクトとして発展をしてきた。PVMプロジェクトはその実験的性質から、副産物として科学者のコミュニティあるいはその他の分野の研究者に役立つようなソフトウェアをこれまで作り出している。元々は、ワークステーション・クラスタのためのTCP/IPベースの通信ライブラリであったが、現在では多くの並列計算機にも移植されている。

しかしながら、PVMでは各並列計算機ベンダが独自にチューニングを施した独自のPVMを開発しており、移植性に乏しいという欠点がある。これは、MPIフォーラムのような第三者的なしっかりとした決定機関を保有していないことに起因している。

かつては分散メモリ環境ではPVMが主流であったものの、現在はMPIに押されておりWeb上における情報量もMPIと比較して圧倒的に少ないのが現状である。

PVMの公式サイトはhttp://www.epm.ornl.gov/pvm/pvm_home.htmlである。ここでPVMの本体やチュートリアル、その他の様々な情報を得ることができる。PVMのソースコードはNetlibから取得可能である。Netlibは数値計算を中心とする科学技術計算に関するフリーソフトやドキュメントをアーカイブしているプロジェクトで、情報は<http://www.netlib.org/>から得ることができる。PVMのソースコードは<http://www.netlib.org/pvm3/index.html>から得ることができる。ここには、ソースコード以外にもドキュメントや仮想コンソールのGUIであるXPVM、テストユーティリティ、論文、ユーザグループの活動報告などPVMに関するあらゆるアーカイブが置かれている。

MPIとPVMの比較

MPIとPVMに関する議論は、これまでも幾つか行われている。最近の動向としては、MPIが新規規格であるMPI-2の仕様において、それまでのPVMにしか無かった「動的なプロセス管理」の機能を取り入れるなど活発な動きを見せており、MPIがかなり優勢な状況となっている。

ここで両ライブラリの利点・欠点についてまとめる。

PVMは、ワークステーションをクラスタにすることで再利用するという世界で育ってきたものである。そのために不均質ないくつものマシンやオペレーティング・システムを直接に管理する。そしてダイナミックに仮想マシンを形成することができる。PVMのプロセス管理の機能では、アプリケーションの中から動的にプロセスを生成したり、停止したりすることができる。この機能はMPI-1にはなく、MPI-2で採り入れられている。さらに利用可能なノードのグループを管理する機能では、ノード数を動的に増減したり調べたりすることができる。これは、MPIではまだ採り入れられていない。PVMの利用における最大の問題点は、先にも述べた移植性である。

それに対してMPIは、その実装がMPP (Massively Parallel Processors) やほとんど同一な特定のワークステーション・クラスタを対象としている。MPIはPVMの後に設計されたために、明らかにPVMにおける問題点を学んでいる。つまりMPIの方がPVMに比べて、高レベルのバッファ操作が可能であり、高速にメッセージを受け渡すことができる。そして、オープンなフォーラムによって達成された「標準」であるために、MPIで作成されたプログラムは非常に移植性が高い。ただしMPIの実装は数多くあるが、MPI-2を完全にサポートしたものはまだない。また、Webや書籍などでの情報もMPIの方が入手しやすく便利であるといえる。

ここで、MPIのライブラリが集まる<http://www.mpi.nd.edu/lam/lam-info.php3>では、「MPIがPVMより優位である10の理由」と題して興味深いリストが載っていたので紹介する。ここでは、MPIがPVMに勝っている点として、

1. MPIはフリーの実行可能かつ高品質なインプリメンションを1つ以上持っている。
2. MPIは第3者的な決定機関を持っている。

3. MPI グループは完全に非同期通信をサポートしている。
4. MPI グループはしっかりとした基盤と持ち能率的で決定論的な側面を持っている。
5. MPI は効果的なメッセージバッファを扱うことができる。
6. MPI における同時性は第 3 者的なソフトウェア機関により保護されている。
7. MPI は効果的な MPP, クラスタのプログラミングを行うことができる。
8. MPI は全体的にポータブルである。
9. MPI は正式に明示されている。
10. MPI が標準である。

を挙げている。上記の理由の主な論拠は、

- 複数の信頼性の高いライブラリの存在

- MPI フォーラムの存在

- MPI の移植性の良さ

などである。このことより、MPI において、MPI フォーラムの存在が MPI への信頼性の向上に大きく関与していることがわかる。また、PVM における最大の問題点である移植性の良さも MPI の最大の利点の一つとなっていることも分かる。

□ LAM と MPICH

MPI はインターフェースの規定であり、実装パッケージそのものではない。MPI の実装ライブラリとしてはフリー、ベンダ問わずに数多く存在する。その中でも代表的なフリーのライブラリとしては、LAM と MPICH があげられる。ここでは、LAM と MPICH の 2 つのライブラリについて簡単に述べる。もし、これらの MPI のライブラリに関するより詳細な情報に興味があれば <http://www.mpi.nd.edu/MPI/> を参考にして頂きたい。

LAM

LAM(Local Area Multicomputer) は、ノートルダム大学の科学コンピュータ研究室 (Laboratory for Scientific Computing, University of Notre Dame) が作成したフリーの MPI ライブラリである。LAM は、標準的な MPI API だけでなく幾つかのデバッキングとモニタリングツールをユーザに提供している。MPI-1 を完全にサポートしているだけでなく MPI-2 の標準的な幾つかの要素についても機能を提供している。最新バージョン 6.3.2 では MPI-2 における 1 方向通信、動的プロセスの管理に関する機能がカバーされている。

また、LAM は世界中のほとんどの UNIX 系ベンダの並列マシン上で利用することができる。ただし、Windows といった UNIX ベースでないベンダに関してはサポートされていない。

LAM は、XMPI との相性が良く XMPI を使用したいのであれば非常に便利である。LAM に関する詳細な情報は、<http://www.mpi.nd.edu/lam/> を参考にして頂きたい。非常に多くの有益な情報を得ることができる。

MPICH

MPICH は、アメリカのゴードン国立研究所 (Argonne National Laboratory) が模範実装として開発し、無償でソースコードを配布したライブラリである。移植しやすさを重視した作りになっているため盛んに移植が行われ、LAM 同様、世界中のほとんどのベンダの並列マシン上で利用することができ

る。特に，MPICH では UNIX 系に限らず Windows 系へのサポートも充実している。さらに，SMP，Myrinet などのハード面にも対応している上，DQS，Globus といった様々なツールを使用できることも大きな特徴の一つである。また，MPICH 1.2.0 では，MPI-1.2 の全ての機能をカバーしており，MPI-2 に関しても幾つかの機能についてはサポートしている。MPICH 1.2.0 における MPI-2 への詳細なサポート情報については <http://www-unix.mcs.anl.gov/mpi/mpich/mpi2-status.html> に載せられている。また，MPICH に関する詳細な情報は，<http://www-unix.mcs.anl.gov/mpi/mpich/> を参考にして頂きたい。ここでの Installation Guide では MPICH 導入に関する詳細な情報を得ることができる。

■ インストールに関して

ここでは、並列ジョブを実行するための最低限の準備 (環境設定) に関して説明する。

□ インストール

他の講義と重なっている部分があるが、ここでは LAM と MPICH のインストール、特にコンフィグについて述べる。尚、以下の内容はソースコードレベルからインストールを前提としている。

コンフィグ

ここでは、コンフィグをライブラリのインストールのための設定という意味で用いる。LAM や MPICH は、様々な環境や目的に対応するため多くの機能を持っている。しかし、単純にそれらのライブラリの機能を全てインストールするのでは、無駄な機能が多くなり容量が大きくなるだけでなく、自分の使っている環境とマッチせずライブラリとしての本来の機能を発揮できない恐れがある。

それを避けるためインストールの前準備として、コンフィグという作業を行い、一体自分はどういった環境でのライブラリ使用を求めているのか、ライブラリのどういった機能を実現したいのかを明示的に指定してやる必要がある。

具体的には、コンフィグによって得られた情報に基づき make ファイルが変更され、その結果、求める形態のライブラリがインストールされるという仕組みになっている。

LAM

LAM のインストール方法について示す。

ドキュメントでの推奨では、`/usr/local/lam-6.***/`へのインストールになっているが、特に指定は無い。LAM は MPICH と異なりプロセスの起動がマスターからの `rsh` を用いるといった方法では無い¹。そのため MPICH では、ホストマシンさえインストールされていれば良いのに対して LAM では、各マシン毎にライブラリをインストールするか、もしくはライブラリの部分を NFS で共有させてやる必要がある。尚、バージョンは 6.4-a3 を用いた。

・ アーカイブの入手

まず、LAM のアーカイブを <http://www.mpi.nd.edu/lam/download/> より入手する。

・ 展開, コンパイル

手に入れたソースを以下のようにコンパイル, コンフィグ, メイクを行う。

```
$ tar xvfz lam-6.4-a3.tar.gz
$ cd lam-6.4-a3
$ ./configure --prefix=/usr/local/lam-6.4-a3 --with-romio
// configure に関する詳細については, $./configure --helpless によって得られる。
```

```
-prefix      : インストールするディレクトリ
--with-romio : ROMIOをサポート。ROMIOを組み込むことにより、MPI-2からMPI-I/O
               の機能を組み込むことができる。
```

¹LAM と MPICH ライブラリは、MPI プロセス間での通信チャネルの起こし方が異なっている。クライアント to クライアントモデルにおいて、MPICH では要求によりコネクションが形成されるのに対して、LAM では初期状態において全てのネットワークがつながれる。そのためコネクションの立ち上がり時間において両者は若干の差がある。

```
$make
[ .. lots of output .. ]
```

MPICH

次に、MPICHにおけるインストール方法について示す。前述したとおりMPICHでは全てのマシンにインストールもしくはライブラリ部分をNFSで共有する必要はない。そのため、LAMに比べて多少インストールが楽と感じるかも知れない。ここでは、インストール先として、`/usr/local/mpich/`を想定する。

・アーカイブの入手

まず、MPICHのアーカイブを <http://www-unix.mcs.anl.gov/mpi/mpich/download.html> より入手する。尚、バージョンは1.2.0を用いた。

・展開、コンパイル

手に入れたソースを以下のようにコンパイル、コンフィグ、メイクを行う。

```
$su
#tar xvzf mpich.tar.Z
[ .. lots of output .. ]
#cd mpich-1.2.0
#./configure --prefix=/usr/local/mpich --with-device=ch_p4 --with-arch=LINUX
--with-romio -opt=-O2 -fc=g77 -flinker=g77
// LAMと同様、詳細については、#./configure --help|less によって得られる
--with-device:コミュニケーションデバイスの指定.
--with-arch :アーキテクチャの指定.
-opt       :この最適化オプションは、MPICHを構築するに辺り使用されるもの
           である。すなわち、mpicc,mpiCC,mpif77といったスクリプトには
           一切影響しない.
-fc        : fortran コンパイラの指定
-flinker    : fortran リンカ指定
[ .. lots of output .. ]
#make
[ .. lots of output .. ]
#make install
[ .. lots of output .. ]
```

・マシンの指定する

`/usr/local/mpich/share/machines.LINUX` を編集し、MPIで利用するマシン名の一覧を作成する。図2.1に示すモデルを参考にした場合について図2.2に例を示す。

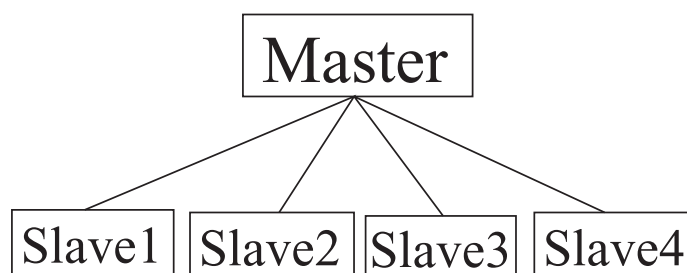


図 2.1: サンプルクラスタモデル

```

# Change this file to contain the machines that you want to use
# to run MPI jobs on.  The format is one host name per line, with either
#   hostname
# or
#   hostname:n
# where n is the number of processors in an SMP.  The hostname should
# be the same as the result from the command "hostname"
master.opt.isl.doshisha.ac.jp
slabe01.opt.isl.doshisha.ac.jp
slabe02.opt.isl.doshisha.ac.jp
slabe03.opt.isl.doshisha.ac.jp
.

```

図 2.2: /usr/local/mpich/share/machines.LINUX

□ パスの設定

各 Linux パッケージに対応したバイナリファイルからインストールした場合には、自動的に mpi 関連のコマンド (mpicc, mpiCC, mpirun など) のパスが通るため特に設定の必要ない。しかしソースファイルからコンパイルを行いインストールした場合には、各種パスの設定を行う必要がある。ここでは、bash の場合を例にコマンドラインへのパスの例を示す。

自分のホームディレクトリ内にある **.bashrc**, (MPICH ならば **.bash_profile** でも構わない) に以下の内容を書き加える必要がある (無論, 直接コマンドを打ち込んでも良い)。

[.bashrc (LAM の場合)]

```

export PATH="$PATH:/usr/local/lam-6.4-a3/bin"
export MANPATH="$MANPATH:/usr/local/lam-6.4-a3/man"

```

[.bashrc もしくは, .bash_profile (MPICH の場合)]

```

export PATH="$PATH:/usr/local/mpich/bin"
export MANPATH="$MANPATH:/usr/local/mpich/man"

```

□ ノードの指定

LAMでは、**作業ディレクトリ内**において `lamhosts` ファイルを作成し、使用する各ノードのホスト名を記入する。また、LAM・MPICH ともに実行には、プログラムの通信コマンドとして `rsh` を使用するため、**自分のホームディレクトリ**に `.rhosts` というファイルを作成し、使用する各ノードのホスト名を記入する必要がある。下記はその一例である。

[`lamhosts`]

```
master.opt.isl.doshisha.ac.jp
slabe01.opt.isl.doshisha.ac.jp
slabe02.opt.isl.doshisha.ac.jp
slabe03.opt.isl.doshisha.ac.jp
.
.
```

[`.rhosts`]

```
master.opt.isl.doshisha.ac.jp
slabe01.opt.isl.doshisha.ac.jp
slabe02.opt.isl.doshisha.ac.jp
slabe03.opt.isl.doshisha.ac.jp
.
.
```

□ SMP を含んだクラスタでは …

最新版の MPICH では、SMP クラスタのサポートも行われている。

SMP クラスタとしてサポートさせるには、システムの `machies.LINUX(/usr/local/mpich/share/machies.LINUX)` において、下記の様に記述すれば良い。ただしこれは、`mpich` のバージョンが 1.2.0 以降からの機能である。

[`/usr/local/mpich/share/machies.LINUX`]

```
master:2
slabe01:2
slabe02:2
slabe03:2
.
.
```

MPI ビルド時のオプションに SMP 用のものがある。SMP を含むクラスタにおいて実験を行ったところ、逆に計算時間が増加するという結果が得られたものの、完全な SMP 単体での MPI プログラミングを行う場合には有効であると思われる。その場合、インストール時の `config` のオプションに `-comm=shared` を加えれば良い。尚、この場合の `-comm` とはコミュニケーションタイプのことを意味する。

■ 並列プログラミング

以降より実際に MPI を用いて並列プログラムを組み、最低限必要であろうと思われる通信関数について解説を行う。MPI においてデータを交換するための通信関数には、任意の 2 つのプロセス同士だけが関わる **1 対 1 通信**（**2 地点間**）通信と、任意のグループに属するプロセス（プロセス全体も含む）が全て関わる **グループ通信**（**集団通信**）が用意されている。さらに、1 対 1 通信、グループ通信のそれぞれに **ブロッキング通信** と **ノンブロッキング通信** が用意されておりそれぞれを適所で使用することができる。ブロッキング通信とノンブロッキング通信に関して概念図を図 2.3, 2.4 に示す。

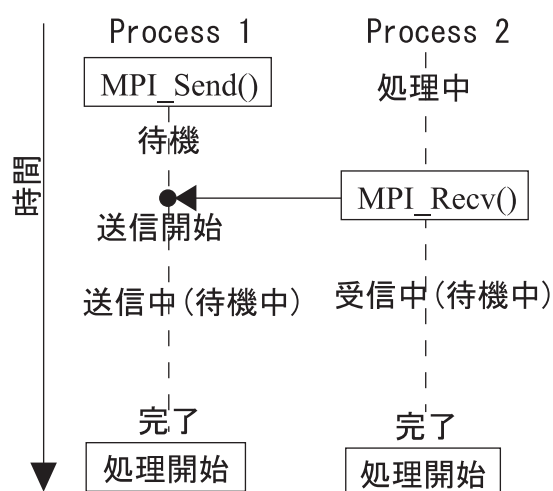


図 2.3: ブロッキング通信

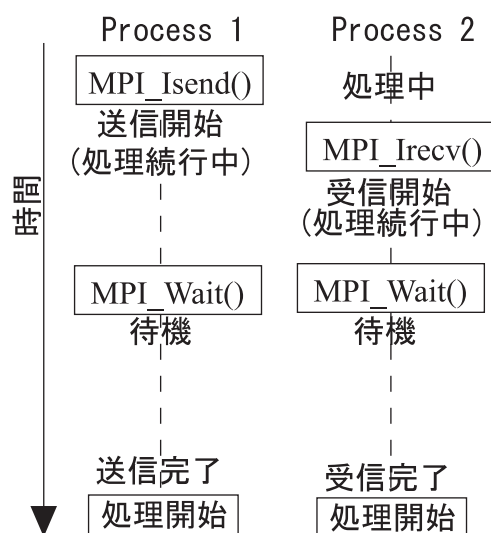


図 2.4: ノンブロッキング通信

□ ブロッキング

ブロッキングとは、操作が完了するまで手続きから戻ることがない場合のことを意味する。この場合、各作業はその手続きが終了するまで待たされることになり効率が悪くなる場合がある。具体的には、図 2.3 に示すように送信、受信が宣言されてから完了するまで処理は待機状態になるため非常に無駄が多くなってしまいうのである。ただし、各操作の完結が保証されているためノンブロッキングに比べ簡単である。

□ ノンブロッキング（非ブロッキング）

ノンブロッキングとは、操作が完了する前に手続きから戻ることがあり得る場合のことを意味する。ノンブロッキング通信を用いることによりより効率の良いプログラムを作成することができる。具体的には、図 2.4 に示すように、送信、受信が宣言されてからも処理を継続することができるためブロッキング通信に比べ通信待ちの時間が少なくなり処理時間の軽減を計ることができる。特に、非同期通信などを行う場合にはノンブロッキング通信は必要不可欠である。しかし、操作の完了が保証されていないため、ノンブロッキング通信を完了するための関数 (MPI_Wait()) を呼び出す必要がある。

尚、ノンブロッキング関数には接頭語として I(即座 immediate) が必ずついている。また、基本的には 1 対 1 通信のみにしかノンブロッキング通信は存在しない。

並列プログラムの実行方法についてはさらに後の節で説明するが、プログラムを書くにあたって並列プログラムの実行の様子について概念的に理解しておく必要がある。手順を述べると、まずユーザはクラスタの 1 つのマシン、あるいは専用の並列計算機のホストとなっているマシンにログインを行う。そこでユーザはクラスタ、あるいは専用の並列計算機に対してジョブの投入を行う。ジョブの投入のために MPI ではスクリプト、専用の並列計算機では専用のコマンドが用意されている。プログラムが無事終了すると、ファイル、あるいはユーザのモニタに結果が出力される。このように実際に並列に走る個々のプロセスの起動は、実行環境に任されている。プログラマはプログラムの中で、最初に通信ライブラリを利用するための簡単な手続きを記述し、その後で実際の並列処理の部分を書いていけばよいことになる。

□ MPI の初歩

MPI-1 には 127 個の通信関数が用意されている。しかし、20 程度の関数を知っていれば、通信をかなり細かく制御することができる。さらにいうなれば、最低限の制御関数 (MPI_Init(), MPI_Comm_size(), MPI_Comm_rank(), MPI_Finalize()) と 2 つの通信関数 (MPI_Send(), MPI_Recv) さえ使いこなせることができれば、単純な並列プログラムは組むことができる。

MPI では全てのプロセスを一斉に起動する。そして各プロセスは、他のほとんどの MPI 関数よりも先に MPI_Init() を呼ばなければならない。MPI を用いた並列プログラムの枠組みを図 2.5 に示す。

```
#include "mpi.h"    // ヘッダファイルの読み込み

int main(int argc, char **argv)
{
    int numprocs, myid;

    MPI_Init(&argc,&argv); // MPI ライブラリを利用するための準備（初期化）
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    // コミュニケータ内のプロセスの数を取得. この関数により numprocs
    にはプロセス数が入力される.
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    // コミュニケータ内の各プロセスが自分の rank を取得. この関数によ
    り myid には自分の rank 番号が入力される.

    /* 並列処理の記述 */

    MPI_Finalize(); // MPI ライブラリの利用の終了処理
    return 0;
}
```

図 2.5: プログラムの枠組み

rank : コミュニケータ内の全てのプロセスは、プロセスが初期化されたときにシステムによって示された ID をもっている。これは 0 から始まる連続した正数が割り当てられる。プログラマはこれを用いて、処理の分岐、あるいはメッセージの送信元や受信先を指定することができる。

コミュニケータ : お互いに通信を行うプロセスの集合である。ほとんどの MPI ルーチンは引数としてコミュニケータを取る。変数 `MPI_COMM_WORLD` は、あるアプリケーションを一緒に実行している全プロセスからなるグループを表しており、これは最初から用意されている。また新しいコミュニケータを作成することも可能である。

よく用いられていると思われる MPI 関数の主なものを表 2.1 に示しておく。それ以外にも有用なものは存在するが、それは各実装に付属のドキュメントや MPI の仕様書を参照されたい。特に、Web では IBM のオンラインマニュアル (Web Book) の AIX に関するマニュアル <http://www.jp.ibm.com/manuals/webbook/gc23-3894-01/index.html> が各 MPI 関連のルーチンに関して詳細に載っているので参考にして頂きたい。

表 2.1: 主な MPI 関数

サブルーチン	タイプ	内容
MPI_Init()	環境管理	MPI の実行環境の初期化
MPI_Comm_rank()	連絡機構	コミュニケータ内のランクを取得
MPI_Comm_size()	連絡機構	コミュニケータ内のプロセス数を取得
MPI_Finalize()	環境管理	MPI の実行環境の終了 (全ての MPI 処理を終了)
MPI_Send()	2 地点間	ブロッキング送信. 最も標準的な送信関数
MPI_Recv()	2 地点間	ブロッキング受信. 最も標準的な受信関数
MPI_Sendrecv()	2 地点間	ブロッキング送受信
MPI_Isend()	2 地点間	ノンブロッキング送信.
MPI_Irecv()	2 地点間	ノンブロッキング受信.
MPI_Iprobe()	2 地点間	source, tag, および comm と一致するメッセージが着信しているか検査する.
MPI_Probe()	2 地点間	source, tag, および comm と一致するメッセージが着信するまで待機する. MPI_Iprobe と非常に似ているが, 常に一致するメッセージが検出されてから戻るブロッキング呼出しであるという点が異なる.
MPI_TEST()	2 地点間	ノンブロッキング送受信操作が完了しているか検査. MPI_Iprobe にある種近い働きをする.
MPI_Wait()	2 地点間	特定の非ブロック送受信の完了を待つ
MPI_Waitall()	2 地点間	全ての非ブロック送受信の完了を待つ
MPI_Get_count()	2 地点間	メッセージの要素数を返す.
MPI_Barrier()	集合通信	コミュニケータ内でバリア同期をとる
MPI_Bcast()	集合通信	メッセージのブロードキャスト
MPI_Reduce()	集合通信	コミュニケータ内で通信と同時に指定された演算を行う
MPI_Type_extent()	派生データタイプ	特定のデータタイプのサイズを取得
MPI_Type_struct()	派生データタイプ	新しいデータタイプを作成
MPI_Type_commit()	派生データタイプ	システムに新しいデータタイプを委ねる
MPI_Type_free()	派生データタイプ	データタイプの削除

■ 1対1通信

MPI の 1 対 1 通信には、数多くの関数が用意されている。これにより、より細かく通信を制御することが可能である。しかしここでの説明は、基本的な送受信関数の紹介にとどめておく。ほとんどの MPI 関数のプロトタイプでは、成功した場合の戻り値は `MPI_SUCCESS` になる。失敗した場合の戻り値は実装によって異なる。

[1 対 1 通信関数]

図 2.6 に示すプログラムは、rank0 と rank1 の 2 つプロセスがお互いに "hello" というメッセージを送り合うプログラムである。用いる関数は、ブロッキング通信である送信関数 `MPI_Send()` と受信関数 `MPI_Recv()` の 2 つである。

```
int MPI_Send( void *buf, int count, MPI_Datatype datatype, int dest, int tag,
              MPI_Comm comm )
```

基本的なブロッキング送信の操作を行う。送信バッファのデータを特定の受信先に送信する。

void *buf: 送信バッファの開始アドレス (IN)
 int count: データの要素数 (IN)
 MPI_Datatype datatype: データタイプ (IN)
 int dest: 受信先 (IN)
 int tag: メッセージ・タグ (IN)
 MPI_Comm comm: コミュニケータ (IN)

```
int MPI_Recv( void *buf, int count, MPI_Datatype datatype, int source, int tag,
              MPI_Comm comm, MPI_Status status )
```

要求されたデータを受信バッファから取り出す。またそれが可能になるまで待つ。

void *buf: 受信バッファの開始アドレス (受け取ったデータの格納場所) (OUT)
 int source: 送信元 (`MPI_ANY_SOURCE` で送信元を特定しない) (IN)
 int tag: メッセージ・タグ (`MPI_ANY_TAG` でメッセージ・タグを特定しない) (IN)
 MPI_Status *status: ステータス (OUT)

データタイプ

ポータビリティを高めるために、MPI によって前もって定義されたデータ型である。例えば、`int` 型であれば `MPI_INT` というハンドルを用いることになる。プログラマは、新たなデータ型を定義することが可能である。基本データタイプを以下の表に示す。

タイプ	有効なデータタイプ引数
整数	<code>MPI_INT</code> , <code>MPI_LONG</code> , <code>MPI_SHORT</code> , <code>MPI_UNSIGNED_SHORT</code> , <code>MPI_UNSIGNED</code> , <code>MPI_UNSIGNED_LONG</code>
浮動小数点	<code>MPI_FLOAT</code> , <code>MPI_DOUBLE</code> , <code>MPI_REAL</code> , <code>MPI_DOUBLE_PRECISION</code> , <code>MPI_LONG_DOUBLE</code>
複素数	<code>MPI_COMPLEX</code>
バイト	<code>MPI_BYTE</code>

メッセージ・タグ : メッセージを識別するためにプログラマによって割り当てられた任意の整数である。ワイルドカードとして `MPI_ANY_TAG` が存在するが、より安全に送受信を行うためには任意の整数を指定するのが望ましい。

MPI_Status : 送信元のランクや送信時に指定されたタグの値を格納する構造体である。 `mpi.h` では以下のように定義されている。

```
typedef struct {
    int count; //受信されるエントリの数 (整数型)
    int MPI_SOURCE; //送信先の情報
    int MPI_TAG; //タグに関する情報
    int MPI_ERROR; //エラーコード
    int private_count;
} MPI_Status;
```

なお、返却ステータスである `MPI_Status` 内の各変数に直接アクセスすることはできない。そのため、例えば受信されているエントリの数 (データ数) などを知りたい場合には、 `MPI_Get_count` を用いて値を返してもらう。

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
    int myid, procs, src, dest, tag=1000, count;
    char inmsg[10], outmsg[]="hello";
    MPI_Status stat;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    count = sizeof(outmsg) / sizeof(char);

    if(myid == 0){
        src = 1;
        dest = 1;
        MPI_Send(&outmsg, count, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
        MPI_Recv(&inmsg, count, MPI_CHAR, src, tag, MPI_COMM_WORLD, &stat);
        printf("%s from rank %d\n", &inmsg, src);
    }else{
        src = 0;
        dest = 0;
        MPI_Recv(&inmsg, count, MPI_CHAR, src, tag, MPI_COMM_WORLD, &stat);
        MPI_Send(&outmsg, count, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
        printf("%s from rank %d\n", &inmsg, src);
    }

    MPI_Finalize();
    return 0;
}
```

図 2.6: Hello.c

□ もっとスマートに

先ほどのプログラム hello.c を、MPI_Sendrecv() を使うことによりもっと通信関数を減らすことができる。

```
int MPI_Sendrecv(void* sendbuf, int sendcount, MPI_Datatype sendtype, int dest,
                 int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype,
                 int source, int recvtag, MPI_Comm comm, MPI_Status *status)
```

基本的なブロッキング送信の操作を行う。送信バッファのデータを特定の受信先に送信する。

void *sendbuf: 送信バッファの開始アドレス (IN)

int sendcount: 送信データの要素数 (IN)

MPI_Datatype sendtype : 送信データタイプ (IN)
 int dest : 受信先 (IN)
 int sendtag : 送信メッセージ・タグ (IN)
 void *recvbuf : 受信バッファの開始アドレス (受け取ったデータの格納場所) (OUT)
 int recvcnt : 受信データの要素数 (IN)
 MPI_Datatype recvttype : 受信データタイプ (IN)
 int source : 送信元 (MPI_ANY_SOURCE で送信元を特定しない) (IN)
 int recvtag : 受信メッセージ・タグ (IN)

この MPI_Sendrecv() 関数は、先ほどの MPI_Send() , MPI_Recv() 関数の代わりに次のように用いることができる。

```

MPI_Recv(&inmsg,count,MPI_CHAR,src,tag,MPI_COMM_WORLD,&stat);
MPI_Send(&outmsg,count,MPI_CHAR,dest,tag,MPI_COMM_WORLD);

```



```

MPI_Sendrecv(&outmsg,count,MPI_CHAR,dest,tag,&inmsg,count, MPI_CHAR,src,tag,MPI_
COMM_WORLD,&stat);

```

□ ノンブロッキング通信を多用しよう

1 対 1 通信においては、ノンブロッキング通信を用いる方が望ましい。というのも、ブロッキング通信では送受信の手続きが完了してから次の操作を開始するため通信での遅延が直接プログラムの計算時間に跳ね返ってくるためである。逆にノンブロッキング通信では、送受信の手続きを行いながら次の操作も平行して行うことができるため、通信での遅延を軽減することができる。

ここでは、ノンブロッキング通信での基本的な送受信方法について例を示す。ノンブロッキング通信における 1 対 1 通信では、送信ルーチンとして MPI_Isend(), 受信ルーチンとして MPI_Irecv() を用いる。また、ノンブロッキング通信では、送受信の手続きの完了を待たずに次の操作が行われるため明示的に送受信の完了を宣言する必要がある。具体的には、MPI_Wait() により明示的な完了を宣言する。

以下では、rank 0 と rank 1 の 2 つのプロセス間での簡単なノンブロッキングの送信、受信プログラムの例を示す。

```

int MPI_Isend(void* sendbuf,int sendcount,MPI_Datatype sendtype, int dest,MPI_Comm
comm,MPI_Request *request)

```

基本的なノンブロッキング送信の操作を行う。

MPI_request : 通信要求 (ハンドル) を返す。 (OUT)

MPI_Request : ノンブロッキング通信における送信もしくは受信を要求したメッセージに付けられる識別子。整数である。

```

int MPI_Irecv(void *recvbuf,int recvcnt,MPI_Datatype recvttype, int source,int recv-
tag,MPI_Comm comm,MPI_request *request)

```

基本的なノンブロッキング受信の操作を行う。

int MPI_Wait(MPI_request *request, MPI_Status *status)

MPI_WAIT は、request によって識別された操作が完了した時点で手続きが終了する。このルーチンによりノンブロッキング通信は完了する。具体的には、ノンブロッキングでの送信、受信呼出によって作成された request が解除され、MPI_REQUEST_NULL が設定される。また、完了した操作に関する情報は status 内に設定される。

[isend_irecv.c]

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
    int myid, procs, src, dest, tag=1000, count;
    int date[100];
    MPI_Status stat;
    MPI_Request request;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    //略 (何らかのプログラム操作)

    if(myid == 0){
        src = 1;
        dest = 1;
        count = 100;
        MPI_Irecv(&date, count, MPI_INT, src, tag, MPI_COMM_WORLD, &request);
        //略 (何らかのプログラム操作)
        MPI_Wait(&request, &stat);
    }else{
        src = 0;
        dest = 0;
        count = 100;
        MPI_Isend(&date, count, MPI_INT, src, tag, MPI_COMM_WORLD, &request);
        //略 (何らかのプログラム操作)
        MPI_Wait(&request, &stat);
    }
    //略 (何らかのプログラム操作)
    MPI_Finalize();
    return 0;
}
```

■ グループ通信

□ π 計算のアルゴリズム

次にグループ通信について説明する．並列化を行う対象は π を近似的に求めるプログラムである． π は式 2.1 に示す計算式で求めることができる．またこれを逐次的に計算するプログラムは，積分計算が図 2.7 に示すように近似的に行われるので，0 から loop-1 個の区間の積分計算に置き換えられる．つまり，この積分計算のループ部分を並列化する．プログラム例では，複数のプロセッサで loop 個の区間の計算を分担するようなアルゴリズムを採用する．

$$\pi = \int_0^1 \frac{4}{1+x^2} dx \quad (2.1)$$

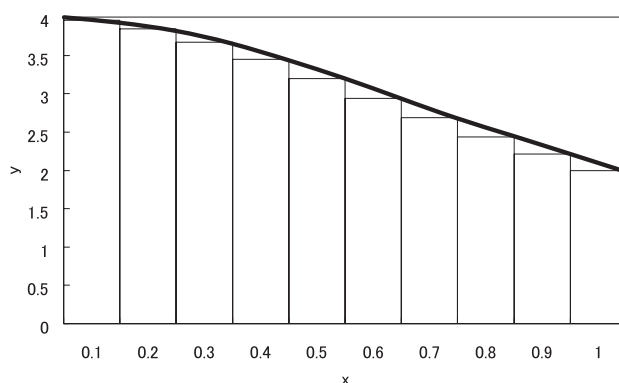


図 2.7: $y = \frac{4}{1+x^2}$

[逐次プログラム]

```
#include <stdio.h>

int main(int argc, char **argv)
{
    int i, loop;
    double width, x, pai=0.0;

    loop = atoi(argv[1]);
    width = 1.0 / loop;

    for(i=0; i<loop; i++){
        x = (i + 0.5) * width;
        pai += 4.0 / (1.0 + x * x);
    }
    pai = pai / loop;
    printf("PAI = %f\n", pai);
    return 0;
}
```

□ MPI のグループ通信

[グループ通信関数]

MPI には 1 対 1 通信の他にコミュニケータ内の全てのプロセスが参加するグループ通信が用意されている。MPI では 16 個のグループ通信のための通信関数が用意されているが、ここでは π 計算のプログラムで用いられている `MPI_Bcast()` と `MPI_Reduce()` について説明する。 π 計算のプログラムはこの 2 つの関数を用いて、図 2.8 に示されるような通信が行われる。まず最初に rank0 のプロセスにおいて、あらかじめ定義された、あるいはユーザによって入力された分割数を他の全プロセスに送信する。次に、それぞれのプロセスは自分の rank と分割数を照らし合わせて、各自が積分計算を行う区間を求め積分計算を行う。最後に計算結果を rank0 に送る。この通信においては、rank0 に各プロセスの計算結果を集めると同時にその総和をとり、 π の近似値を求めている。サンプルプログラムを図 2.9, 2.10 に示す。

int MPI_Bcast (void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

1 つのプロセスからコミュニケータ内の他の全プロセスにメッセージを一斉に送信する。

void *buf : 送信元では送信バッファの開始アドレス、受信先では受信バッファの開始アドレス
int count : データの要素数
MPI_Datatype datatype : データタイプ
int root : 送信元の rank
MPI_Comm comm : コミュニケータ

int MPI_Reduce (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int dest, MPI_Comm comm)

コミュニケータ内の全プロセスのデータのある 1 つのプロセスに集める。また同時に各データを足し合わせるなどの演算を行う。

void *sendbuf : 送信先 (コミュニケータ内の全プロセス) の送信バッファの開始アドレス
void *recvbuf : 受信先 (dest で指定された rank) の受信バッファの開始アドレス
MPI_Op op : 演算のハンドル

演算のハンドル : MPI によって前もって定義された演算を指定することで、通信と同時にどのような演算を行うかを指定する。例えば、各データの合計をとる場合には `MPI_SUM` と指定する。またプログラマは、`MPI_Op_create()` 関数によって独自の演算を定義することができる。代表的な縮約演算子を以下に示す。

縮約演算子	内容
MPI_MAX	最大
MPI_MIN	最小
MPI_SUM	合計
MPI_PROD	積
MPI_LAND	論理 AND
MPI_BAND	ビット単位 AND
MPI_LOR	論理 OR
MPI_BOR	ビット単位 OR
MPI_LXOR	論理 XOR
MPI_BXOR	ビット単位 XOR

double MPI_Wtime ()

このルーチンは、time の現行値を秒数の倍精度浮動小数点数として返す。この値は過去のある時点からの経過時間を表す。この時点は、プロセスが存在している間に変更されない。

MPI_Get_processor_name(char *name,int *resultlen)

呼出し時にローカル・プロセッサの名前を返す。この名前は文字ストリングで、これによって特定のハードウェアを識別する。name は少なくとも MPI_MAX_PROCESSOR_NAME 文字の長さの記憶域を表し、MPI_GET_PROCESSOR_NAME はこれと同じ長さまでの文字を name に書き込むことができる。また、実際に書き込まれる文字数は resultlen に戻される。

char *name：実際のノードに固有の指定子を返す。(OUT)

int *resultlen：name に戻される結果の印刷可能文字の長さを返す。(OUT)

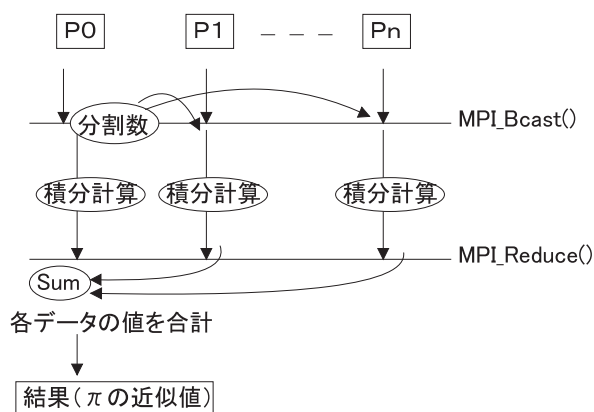


図 2.8: π の計算における通信

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>

double f( double a ){ return (4.0 / (1.0 + a * a)); }

int main( int argc, char *argv[])
{
    int done = 0, n, myid, numprocs, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;
    double startwtime, endwtime;
    int namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(processor_name,&namelen);

    fprintf(stderr,"Process %d on %s\n",myid, processor_name);
    n = 0;
    while (!done){
        if (myid == 0){
/*
            printf("Enter the number of intervals: (0 quits) ");
            scanf("%d",&n);
*/
            if (n==0) n=100; else n=0;
            startwtime = MPI_Wtime();
        }

        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0)
            done = 1;
        else{
            h = 1.0 / (double) n;
            sum = 0.0;

            for (i = myid + 1; i <= n; i += numprocs){
                x = h * ((double)i - 0.5);
                sum += f(x);
            }
        }
    }
}
```

図 2.9: pi.c

[pi.c の続き]

```
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (myid == 0){
        printf("pi is approximately %.16f, Error is %.16f\n",
            pi, fabs(pi - PI25DT));
        endwtime = MPI_Wtime();
        printf("wall clock time = %f\n",endwtime-startwtime);
    }
}

MPI_Finalize();
return 0;
}
```

図 2.10: pi.c の続き

■ その他

□ プログラム内の一部分の計測時間を計りたい

次のような場合、プログラム内の一部分の計測時間を測定することがある。

- (A) 主力計算を行っているタイムステップのループのみの経過時間を測定したい。
- (B) 並列化した部分の経過時間をプロセス毎に測定し、ロードバランスを調べたい。
- (C) 並列化した部分の計測時間と、それに伴う通信時間をプロセス毎に測定して比較したい。

並列プログラムでは、各プロセスでその進行が異なっているため、各プロセス毎で計算時間も異なる。ここでは、全プロセスの最大値を求める場合 (maxtime_ mpi.c) と各プロセス毎にそれぞれ経過時間を求める場合 (each_ proctime_ mpi.c) について示す。

maxtime_ mpi.c では、最大値の計算時間を、each_ proctime_ mpi.c では各プロセス毎にそれぞれ経過時間を求めている。ここで重要なのは、計測前に必ず MPI_Barrier() により同期をとることである。また、最大時間を求める場合には、任意の測定部分が終了した時点で再度同期を取り全てのプロセスの足並みをそろえる必要がある。

int MPI_Barrier (MPI_Comm comm)

すべてのプロセスが呼出しを完了するまでブロックする。プロセスは、すべてのプロセス・メンバの入力が完了するまで操作を終了することはできない。

[maxtime_ mpi.c]

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
    int myid, procs, src, dest, tag=1000, count;
    double s_time, e_time;
    MPI_Status stat;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    MPI_Barrier(MPI_COMM_WORLD); //← 測定前に同期をとる
    s_time = MPI_Wtime(); //← この部分以降からの時間を計測
    //略（測定部分）
    MPI_Barrier(MPI_COMM_WORLD); //← 全プロセスの足並みをそろえる。
    e_time = MPI_Wtime();
    printf("time = %f \n", e_time-s_time);
    MPI_Finalize();
    return 0;
}
```

[each_proctime_mpi.c]

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
    int myid, procs, src, dest, tag=1000, count;
    double s_time, e_time;
    MPI_Status stat;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Barrier(MPI_COMM_WORLD);
    s_time = MPI_Wtime(); //← この部分以降からの時間を計測
    //略（測定部分）
    e_time = MPI_Wtime();
    printf("time = %f \n", e_time - s_time);
    MPI_Finalize();
    return 0;
}
```

□ 受信データの要素数が分からない場合

サンプルプログラムの hello.c, cpi.c では、受信データの要素数が予め分かっていたためスムーズに受信を行うことができた。しかし、場合によっては予め受信するデータの要素数が分かっていない場合もある。

そのような場合、単純に実際の送信データを前もって受信先に教えてやるというのが最も単純な方法として考えられる。しかし、そのために余計な通信を行わなければならない無駄な通信のオーバーヘッドが生じてしまう。そのような場合、MPI_Probe(), もしくは MPI_Iprobe() などを用いて実際には受信操作を行わずに送信されたデータの要素数を把握するのが最も効率の良いやり方である。

以下にその例として、データタイプとして byte 型の要素が要素数不明の状態ですられてきた場合について示す。流れとしては、

1. MPI_Probe() によって送信されてきたデータ情報を MPI_Status に格納する。
2. MPI_Get_count() によって MPI_Status 内から送信データの要素数を把握する。
3. MPI_Recv() を用いて具体的な要素数が把握できた段階において、実際の受信を行う。

となる。

#ただし、以下のプログラムでは char 型が 1byte であると仮定のもとで行っている。

[getcount.c]

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
    int myid, procs, src, dest, msgt, count;
    unsigned char *rna_recv;
    double s_time, e_time;
    size_t n1;
    MPI_Status stat;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    // 相手が何らかのデータを送信した。
    MPI_Probe(MPI_ANY_SOURCE, msgt, MPI_COMM_WORLD, &stat); // 送信データの収集
    MPI_Get_count(&stat, MPI_BYTE, &size); // 送信データサイズの把握
    n1 = size * sizeof(unsigned char);
    if((rna_recv = (unsigned char *)malloc(n1)) == NULL){
        printf("Error \n");
        exit(1);
    }
    MPI_Recv(rna_recv, size, MPI_BYTE, MPI_ANY_SOURCE, msgt, MPI_COMM_WORLD,
             &stat); // 実際の受け取り
    MPI_Finalize();
    return 0;
}
```

□ 構造体を送信受信したい

実際に並列のプログラムをくみ出すと厄介になってくるのが構造体の送受信である。構造体のメモリを連続的に確保している場合には、Byte 型を用いて先頭アドレスを基準に構造体の size 分を指定し送受信を行う、もしくは `MPI_Type_contiguous()`、`MPI_Type_commit()` を使い新たなデータタイプを作成し登録することにより、より簡単に送受信を行うことができる。しかし、実際問題として連続的に構造体のメモリを確保することは難しい場合が多い。

そのため、構造体の中身を何度かに分けて送受信しなければならない状況が生じる。そのような無駄を極力省くため、送信前に構造体の中身を 1 つの Byte 型の変数に全てパックし、受信側で再度アンパックするという方法を用いる。この方法は、パック、アンパックという操作が生じるため小規模 (2 変数しか含まないような) の構造体の場合にはあまり効果は無いが、大規模の構造体では通信回数がかなり減るため計算時間の削減が期待できる。以下では、ある構造体を想定しこの構造体を送受信の際のパック、アンパックの一例を `pack_unpack.c` に示す。尚、本例は並列分散 GA ライブラリ GAPPA のソースを参考にしている <http://mikilab.doshisha.ac.jp/dia/GAPPA/>。

[pack_unpack.c]

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include "mpi.h"
5
6 typedef struct {
7     int a;
8     int *b;
9     double *c;
10 } Sample;
11
12 void all_malloc(struct Sample *s_sample, struct Sample *r_sample)
13 {
14     s_sample->b = (int*)malloc(sizeof(int) * 3);
15     s_sample->c = (double*)malloc(sizeof(double) * 3);
16     r_sample->b = (int*)malloc(sizeof(int) * 3);
17     r_sample->c = (double*)malloc(sizeof(double) * 3);
18 }
19
20 void free_malloc(struct Sample *a, struct Sample *b)
21 {
22     free(s_sample->b);
23     free(s_sample->c);
24     free(r_sample->b);
25     free(r_sample->c);
26 }
```

```
28 unsigned char *transcription(struct Sample *sample)
29 {
30     int i,j,n,mant;
31     unsigned char *buffer;
32     unsigned int mant_n,exp2;
33
34     mant = htonl(sample->a);
35     memcpy(buffer,&mant,4);
36     j=4;
37     for(i=0;i<3;i++){
38         mant = htonl(sample->b[i]);
39         memcpy(buffer+j,&mant,4);
40         j +=4
41     }
42     for(i=0;i<3;i++){
43         mant = (int)frexp(sample->c[i],&n)(1 << 30));
44         mant_t = htonl(mant);
45         exp2 = htonl(n);
46         memcpy(buffer+j,&mant_t,4);
47         memcpy(buffer+j+4,&exp2,4);
48         j += 8;
49     }
50     return buffer;
51 }
52
53 void reverse_transcription(const unsigned char *data,struct Sample *)
54 {
55     int n,i,mant,j;
56     double dum;
57
58     mant = ntohl(*(unsigned int *)(data));
59     sample->a = mant;
60     j=4;
61     for(i=0;i<3;i++){
62         mant = ntohl(*(unsigned int *)(data+j));
63         sample->b[i]=mant;
64         j+=4;
65     }
66     for(i=0;i<3;i++){
67         mant = ntohl(*(unsigned int *)(data+j));
68         n = ntohl(*(unsigned int *)(data + j + 4));
69         dum = ldexp((double)mant, n - 30);
70         sample->c[i]=dum;
71         j += 8;
72     }
73 }
```

```
75 int main(int argc, char *argv[])
76 {
77     int myid, procs, src, dest, msgt, count, size, partner;
78     unsigned char *rna_recv, *rna_send;
79     double s_time, e_time;
80     struct Sample s_sample, r_sample;
81     size_t n1;
82     MPI_Status stat;
83
84     MPI_Init(&argc, &argv);
85     MPI_Comm_rank(MPI_COMM_WORLD, &myid);
86
87     all_malloc(&s_sample, &r_sample);
88
89     ////略 (何らかのプログラム操作)
90     size = sizeof(int) * 4 + sizeof(double) * 3;
91     rna_send = (unsigned char *) malloc(size);
92     rna_recv = (unsigned char *) malloc(size);
93     rna_send = transcription(&s_sample);
94     if(myid == 0)
95         partner = 1;
96     else
97         partner = 0;
98
99     MPI_Sendrecv(rna_send, size, MPI_BYTE, partner,
100                 msgt, rna_recv, size, MPI_BYTE, MPI_ANY_SOURCE,
101                 msgt, MPI_COMM_WORLD, &stat);
102     reverse_transcription(rna_recv, &r_sample);
103     ////略 (何らかのプログラム操作)
104     MPI_Finalize();
105     all_free(&s_sample, r_sample);
106     return 0;
107 }
```

■ MPIのプログラムの実行

□ コンパイル

初めに MPI を用いたプログラムのコンパイル方法について説明する。プログラムのコンパイルには LAM の場合 `hcc` コマンド, MPICH の場合, `mpicc` コマンドを用いる (ここでは `pi.c` を想定)。

LAM

```
$hcc -O -o lampi test.c -lmpi
```

MPICH

MPICH では, `mpicc` を利用したコンパイルを行う。

```
$ mpicc -O -o chpi pi.c
```

□ プログラムの実行

プログラムの実行の前に LAM の場合, 実行の前にデーモンの起動を行わなければならない。何故, LAM にはデーモンの起動が必要で MPICH の場合には必要ないかというと, MPICH はプログラムの実行により `rsh` を用いて `root` から各プロセスを起動しているのに対して, LAM は初期段階から各プロセスを起動した状態でプロセスを走らせるというスタイルをとっているためである。

LAM

前述の通り, LAM では前段階としてデーモンの起動を行う必要がある。

```
$ recon -v lamhosts //lamhosts に記されているノードの確認
recon: -- testing n0 (duke.work.isl.doshisha.ac.jp)
recon: -- testing n1 (cohort1.mpara.work.isl.doshisha.ac.jp)
recon: -- testing n2 (cohort2.mpara.work.isl.doshisha.ac.jp)
recon: -- testing n3 (cohort3.mpara.work.isl.doshisha.ac.jp)
recon: -- testing n4 (cohort4.mpara.work.isl.doshisha.ac.jp)
recon: -- testing n5 (cohort5.mpara.work.isl.doshisha.ac.jp)
[ .. lots of output .. ]
$ lamboot -v lamhosts //lamhosts に記されているノード内からデーモン起動
LAM 6.3.2/MPI 2 C++ - University of Notre Dame

Executing hboot on n0 (duke.work.isl.doshisha.ac.jp)...
Executing hboot on n1 (cohort1.mpara.work.isl.doshisha.ac.jp)...
Executing hboot on n2 (cohort2.mpara.work.isl.doshisha.ac.jp)...
Executing hboot on n3 (cohort3.mpara.work.isl.doshisha.ac.jp)...
Executing hboot on n4 (cohort4.mpara.work.isl.doshisha.ac.jp)...
Executing hboot on n5 (cohort5.mpara.work.isl.doshisha.ac.jp)...
topology done
```

実行コマンドである mpirun は以下のように実行する。

[LAM の場合]

```
mpirun -v [マシン数 ex.n0-5 ] [プログラム名]
```

上記におけるマシン数の指定は, `n0-*` (*整数) のように行う. 例えば 4 台使用したい場合には, `n0-3` となる.

[MPICH の場合]

```
mpirun -np [マシン数 ex. 5 ] [プログラム名]
```

上記におけるマシン数の指定は, `*` (*整数) のように行う. 例えば 4 台使用したい場合には, 4 となる.

mpirun はいくつかのオプションを指定することができる. これらについては,「`mpirun -h`」や「`man mpirun`」を参照されたい. 以下に, π 計算のプログラムを実行した結果を示す.

π 計算の実行

[LAM の場合]

```
$ mpirun -v n0-8 lampi
22953 lampi running on n0 (o)
493 lampi running on n1
207 lampi running on n2
317 lampi running on n3
215 lampi running on n4
210 lampi running on n5
215 lampi running on n6
239 lampi running on n7
277 lampi running on n8
Process 0 on duke
Process 8 on cohort8
Process 1 on cohort1
Process 3 on cohort3
Process 5 on cohort5
Process 7 on cohort7
Process 2 on cohort2
Process 6 on cohort6
Process 4 on cohort4
pi is approximately 3.1416009869231245, Error is 0.0000083333333314
wall clock time = 0.006149
```

π 計算の実行

[MPICH の場合]

```
$ mpirun -np 8 chpi
Process 0 on duke.work.isl.doshisha.ac.jp
Process 1 on cohort10.mpara.work.isl.doshisha.ac.jp
Process 2 on cohort9.mpara.work.isl.doshisha.ac.jp
Process 5 on cohort6.mpara.work.isl.doshisha.ac.jp
Process 6 on cohort5.mpara.work.isl.doshisha.ac.jp
Process 7 on cohort4.mpara.work.isl.doshisha.ac.jp
Process 4 on cohort7.mpara.work.isl.doshisha.ac.jp
Process 3 on cohort8.mpara.work.isl.doshisha.ac.jp
pi is approximately 3.1416009869231245, Error is 0.0000083333333314
wall clock time = 0.016509
```

□ 計算後の処理

LAMでは、起動したデーモンを明示的に終了させてやる必要がある。具体的には、計算ジョブの終了を宣言して、並列計算用のデーモンプロセスを終了させるという作業を行う。

```
$ lamclean -v //計算ジョブの終了
killing processes, done
closing files, done
sweeping traces, done
cleaning up registered objects, done
sweeping messages, done
$ wipe -v lamhosts
LAM 6.3.2 - University of Notre Dame
Executing tkill on n0 (duke.work.isl.doshisha.ac.jp)...
Executing tkill on n1 (cohort1.mpara.work.isl.doshisha.ac.jp)...
Executing tkill on n2 (cohort2.mpara.work.isl.doshisha.ac.jp)...
Executing tkill on n3 (cohort3.mpara.work.isl.doshisha.ac.jp)...
Executing tkill on n4 (cohort4.mpara.work.isl.doshisha.ac.jp)...
[ .. lots of output .. ]
```

■ LAMとMPICHの通信速度

最後に、簡単な通信速度を測定するプログラムを用いて、LAMとMPICHの速度比較を行う。ここまで見てきたように、LAMとMPICHはコンパイル、実行において同じMPIという範疇でありながら若干異なっている。勿論、対応しているアーキテクチャ、DQSなどのジョブ管理システム(JMS)への対応など異なっている点も多く各々の目的に適したライブラリを選択する必要がある。

しかし、単純にIntel系のマシンを安価なハブでつなぎ合わせたクラスタにおいて最も重要になってくるのは、速度の違いであろう。並列化のその最大のメリットである速度について「通信速度」という観点から下記に示すような簡単なプログラムを用いて比較を行った。

プログラムについて、若干の解説を行う。本プログラムは、MPI_Sendrecv(),MPI_Bcast()をそれぞれ1000回行い、各通信関数にかかった時間を表示するというものである。LAMとMPICHではその仕組みの違いにより、立ち上がり時間に差が生じるが今回は、それを省く形で計測を行った。プログラム中の関数などはできる限り上記で説明したものだけを用いた。尚、MPI_Sendrecv()ではできる限り異なる送受信相手となるような仕組みを用いた。ちなみに、この送受信相手の選択方法は、ソートなどで良く用いられる方法である。

[test_mpi.c]

```
1 #include <stdio.h>
2 #include "mpi.h"
3 #include <stdlib.h>
4
5 int main(int argc,char *argv[])
6 {
7     int myid,procs,src,dest,tag=1000,count,nemelen,proc,msgt;
8     int i,j,avg_rank,range,dum,namelen;
9     char flag;
10    MPI_Status stat;
11    char processor_name[MPI_MAX_PROCESSOR_NAME];
12    double startwtime, endwtime,midwtime;
13
14    MPI_Init(&argc,&argv);
15    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
16    MPI_Comm_size(MPI_COMM_WORLD, &procs);
17    MPI_Get_processor_name(processor_name,&namelen);
18    printf("proces_name %s rank %d \n",processor_name,myid);
19    MPI_Barrier(MPI_COMM_WORLD);
20    startwtime = MPI_Wtime();
21    i=0;
22    flag=0;
```

[test_mpi.c の続き]

```
23     while(!flag){
24         avg_rank = procs/2;
25         range=procs;
26         for(j=0;;j++){
27
28             if(myid<avg_rank){
29                 dest=myid+range/2;
30             }
31             else{
32                 dest=myid-range/2;
33             }
34             msgt=100;
35             MPI_Sendrecv(&j,1,MPI_INT,dest,msgt,&dum,1,MPI_I
NT,MPI_ANY_SOURCE,msgt,MPI_COMM_WORLD,&stat);
36             range /= 2;
37             if(range==1)
38                 break;
39             if(myid<avg_rank)
40                 avg_rank = avg_rank - range/2;
41             else
42                 avg_rank += range/2;
43         }
44         i++;
45         if(i>1000)
46             flag=1;
47     }
48     MPI_Barrier(MPI_COMM_WORLD);
49     midwtime=MPI_Wtime();
50     if(myid==0)
51         printf("Sendrecv 1000 clock time = %f\n",midwtime-startwt
ime);
52     for(i=0;i<1000;i++)
53         MPI_Bcast(&dum,1,MPI_INT,0,MPI_COMM_WORLD);
54
55     MPI_Barrier(MPI_COMM_WORLD);
56     endwtime=MPI_Wtime();
57     if(myid==0)
58         printf("Bcast 1000 clock time = %f\n",endwtime-midwtime);
59     MPI_Finalize();
60     return 0;
61 }
```

□ 結果

前ページのプログラムを実行した結果、下記のような値が得られた。

```
[LAM]
Sendrecv 1000 clock time = 1.242967
Bcast 1000 clock time = 0.282649

[MPICH]
Sendrecv 1000 clock time = 3.580716
Bcast 1000 clock time = 4.023609
```

上記の結果より、予想以上に LAM が高速であることが分かる。MPI_Sendrecv(), MPI_Bcast() ともに倍以上に差が開いており、特に MPI_Sendrecv() に至っては 5 倍近くの差が生じている。まだ私自身の勉強不足により、この原因はハッキリとは分かっていない。この点については、至急調査を行う予定である。

■ おわりに

MPIによる並列プログラミングの基礎と題して進めてきたが、まだまだ私自身が知識不足で満足いく内容にはほど遠かったものの、MPIを始める初心者にとって一つの手がかりにはなったのではないだろうか。

今回のドキュメントはあくまでも初歩的な分野を広く浅く触れているだけに過ぎない。ここで、今回の講習を受講された皆さんに、知識の確認と向上のために下記に示す文献、URLを一読されることを強く勧める。紹介する文献、URLに簡単な紹介文も併記しておくので参考にさせていただきたい。

<http://www.mpi.nd.edu/MPI/>

MPIの各種フリー、ベンダのライブラリの情報が載っている。そこから各ライブラリのメインのページへのリンクが張ってるためリンク集としても非常に価値のあるページである。

<http://www.mpi.nd.edu/lam/>

LAMのメインのページである。非常に良く整備されている上、LAMさらにはMPIに関する様々な情報を得ることができる。LAMを使用しないユーザーでも一読する価値はある。

<http://www.jp.ibm.com/manuals/webbook/gc23-3894-01/index.html>

IBMのオンラインマニュアル（Web Book）のAIXに関するマニュアル。各MPI関連のルーチンに関して詳細に載っている。MPI関数に関する使用方法、機能などを調べたいときは大変重宝する。

青山幸也,「虎の巻シリーズ」,(日本アイ・ビー・エム株式会社,1999)

IBMの青山さんが執筆されたドキュメント。日本語の中では、恐らく最も充実したMPIに関するドキュメントである。基本的な並列プログラミングの考え方から、MPIを用いた数値計算プログラミング例、MPIの基本的な関数の使用方法までを網羅しておりかなり豊富な知識を得ることができる。MPIを本格的に学ぶ上では必読と言える。

<http://www.ppc.nec.co.jp/mpi-j/>

上記よりMPI-1, MPI-2の日本語訳ドラフトを手に入れることができる。ドラフト自体を読んでも量が膨大な上、使用方法に関して最低限の記述してしていないため少々読みづらい面は否めない。ただし、このドラフトが全ての基本となっているため手元にあると心強い。

■ 参考文献

- 1) 1999 年超並列講習会「PC クラスタ超入門」テキスト, 「PC クラスタ超入門」, (<http://is.doshisha.ac.jp/SMPP/report/1999/990910/index.html>)
- 2) 1998 年並列処理講習会テキスト, 「並列処理の基礎と実習」, (日本機械学会, 1998.8)
- 3) 湯淺太一, 安村通晃, 中田登志之, 「はじめての並列プログラミング」, (共立出版, 1998.6)
- 4) Linux Japan 特集 2, 「並列処理とクラスタリング」, (レーザー 5 出版局, 1998.7)
- 5) Rajikumar Buyya, 「High Performance Cluster Computing」, (Prentice Hall PTR, 1999)
- 6) 「MPI Standard (日本語訳ドラフト)」, (MPI-J ML, 1996.5)
- 7) 「MPI-2: Extensions to the Message-Passing Interface (日本語訳ドラフト)」, (MPI-J ML, 1999.7)
- 8) 青山幸也, 「虎の巻シリーズ」, (日本アイ・ビー・エム株式会社, 1998)
- 9) Blaise M.Barney, 「The Message Passing Interface, ISHPC 97 Tutorial」, (Maui High Performance Computing Center, 1997.11)
- 10) G.A.Geist, J.A.Kohl, P.M.Papadopoulos, 「PVM and MPI: a Comparison of Features」, (1996.3)

Diskless Cluster 構築入門

同志社大学 大学院 工学研究科

谷村 勇輔

tanisuke@mikilab.doshisha.ac.jp

■ はじめに

PC Cluster は、汎用的な PC を多数集めて構築される並列計算機のことです。PC Cluster はコストパフォーマンスが高いことから、最近では数多く作られるようになりました。それに伴い、Cluster を構築・運用する際の技術も充実しつつあり、用途に合わせた Cluster を構築することが可能になりました。ここでは PC Cluster の中でも、プロセッサ数が比較的小規模（8～32）な場合においてしばしば用いられる Cluster のモデルの 1 つ、**Diskless Cluster** について述べたいと思います。

■ Diskless Cluster とは

PC Cluster は一般に、**サーバマシン** と呼ばれる Cluster 全体を管理するためのマシンと **計算マシン** と呼ばれる実際に計算を行うマシンの 2 種類からなります。おおよそサーバマシンは 1 台であり、計算マシンは複数台用意されることになります。図 3.1 に示すような Cluster を想像して下さい。Diskless Cluster では、この計算マシンのディスク（主に HDD）がありません。必要なファイルは **NFS (Network File System)** などを利用して、サーバマシン上のファイルをネットワーク経由で参照します。これらの特徴より Diskless Cluster の利点は、

- HDD がいないために障害が発生しにくく、かつ低価格で構築できる。あるいは、他の OS が動作しているマシンを Cluster の計算マシンとして使用できる。
- カーネルやファイルシステムをサーバマシンが持つことにより、一元管理しやすい。

であるといえます。逆に欠点としては次のことが考えられます。

- ネットワークの負荷が高くなるために、性能の低下が起きる。
- ローカルに HDD をもたないので swap 領域をとれない。
- 実行時にファイル入出力を頻繁に行うアプリケーションでは、性能が著しく低下する。

実際に大幅な性能低下が起きる可能性があるのは、全計算マシンからネットワーク経由でファイルアクセスがある時です。実行する並列アプリケーションが、頻繁にファイルアクセスしないのであれば、それほど性能の低下を招くことはないと思います。ただし並列アプリケーション実行時には、各計算マシンにおいてネットワーク経由で実行モジュールを読み出さなければなりません。これは、アプリケーションが立ち上がるまでの遅延を引き起こします。また、計算マシンを一斉起動する場合に

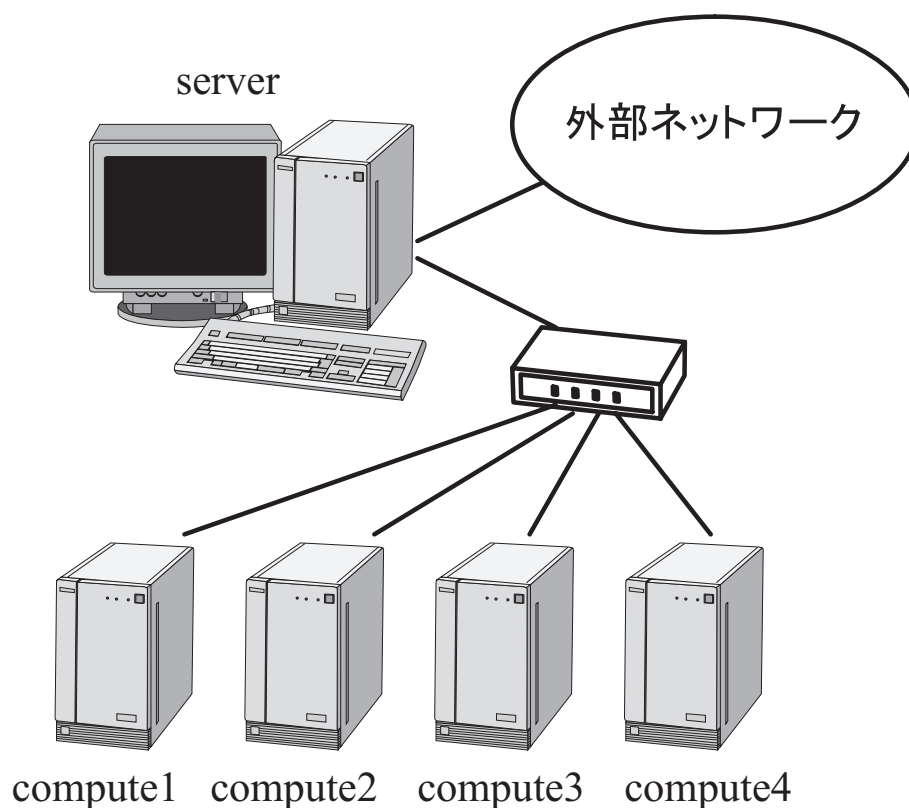


図 3.1: Cluster の概観

もネットワークやサーバマシンに負荷が集中する可能性があります。こうした影響は、Cluster の規模が比較的大きくなった場合に問題になってきます。

■ Diskless Cluster のブートの仕組み

Diskless Cluster は、計算マシンのブートをどのように行うかによって次の 3 つに分類できます。

- FD を用いる
 - (1) FD にカーネルを書き込んでおく
 - (2) サーバマシン上にあるカーネルを読み出す
- ブート ROM を用いる
 - (3) サーバマシン上にあるカーネルを読み出す

FD を用いる方法では、カーネルを FD に書き込むかサーバマシン上に置いておくかの 2 通りの選択肢があります。いずれの場合もブート後は FD を使用しないため、ブートの手順だけが異なります。(1) の場合には次のような手順となります。

電源 ON → (FD より) カーネルのロード → ネットワークの設定 → ルートファイルシステムのマウント (ネットワーク経由) → 各種サービス開始

これを見れば分かるように、(1) は FD で動く Linux (<http://saturn.fokus.gmd.de/linux/linux-distrib-small.html>) を発展させた形であるといえます。FD で動く Linux との違いは、ルー

トファイルシステムをサーバマシン上に持つことです。(2)では、サーバマシン上にあるカーネルをネットワーク経由で読み出します。FDからロードされて最初に動くプログラム(起動ROMバイナリ)が、ネットワークの設定を行います。この種のプログラムとしてはNetbootやEtherboot、NILOなどが挙げられます。これらについての詳細は後述します。以下に(2)のブート手順を示します。

電源 ON → ネットワーク設定 → カーネルのロード(ネットワーク経由) → ルートファイルシステムのマウント(ネットワーク経由) → (改めて)ネットワークの設定 → 各種サービス開始

(1)と(2)の比較では、カーネルの変更があっても管理しやすいこと、カーネルイメージのサイズをあまり気にしなくて良いことなどの理由から、(1)よりも(2)の方が扱い易いと思います。(3)では、(2)で用いた起動ROMバイナリをBIOSに組み込んでおきます。これにより計算マシンではFDDすらも必要でなくなります。ブートROMへの書き込みにはROMライターが必要ですが、On Board書き込みをすることも可能です。ただし、これらの作業には常にBIOS破壊の危険が伴います。ROMライターを用いてあらかじめバックアップをとっておくこと、それなりに覚悟を決めて各自の責任の元で作業を行うようお願いします。最近の高機能のNICには、こうした起動ROMバイナリと同じような機能(BOOTPやDHCPなどの機能)を付加した製品もあります。そのようなNICを利用できる場合には、さらに構築の手間を省くことができると思います。ここで紹介した中から、今回は(2)を用いてDiskless Clusterを構築する方法について説明します。また(3)の手順を用いる場合にも、(2)で十分に動作確認をした後にROMへの書き込みを行うのが良いかと思います。

■ 構築手順

□ 流れ

本章ではDiskless Clusterの構築の手順をおおまかに示します。OSにはDebian/GNU Linuxを用いるため、各種サービスの設定はDebianのものを記述します。その他のディストリビューションやOSを用いる場合には、適宜読みかえて下さい。以降の節では、次のような手順でDiskless Clusterを構築していきます。

●サーバマシンの構築

Linuxのインストール
↓
カーネルの再構築
↓
BOOTPサーバの設定
↓
TFTPサーバの設定
↓
NFSサーバの設定

●計算マシンの構築

計算マシン用ディレクトリの作成
↓
計算マシン用の設定ファイル、起動・終了スクリプトの作成
↓
計算マシン用カーネルの作成
↓
計算マシン用起動ディスクの作成
↓
Diskless Clusterの動作確認

Cluster の構成としてはサーバマシンが 1 台、計算マシンが複数台あるものとします。図 3.1 に示すようにサーバマシンの名称を server、計算マシンの名称を compute1、compute2 のように付けていきます。各マシンの IP アドレスは以下に示す通りです。参考までに、表 3.1 に使用するマシンスペックを示しておきます。

```
server    : 202.23.xx.xx (グローバル)
           192.168.1.1  (Cluster 内)

compute1 : 192.168.1.11
compute1 : 192.168.1.12
```

表 3.1: 各マシンスペック

CPU	Celeron (Mendocino)
Clock	466MHz
Memory	64MB
NIC	Intel EtherExpress Pro/100+
OS	Debian GNU / Linux 2.2 (potato)
kernel version	2.2.15

□ サーバマシンの準備

まず最初に、Diskless Cluster のサーバマシンを構築します。このサーバマシンは通常の PC Cluster のサーバマシンに、ディスクレスである計算マシンのためのサービスを付加したものになります。以下の説明では、既にサーバマシンに Linux がインストールされ、コンパイラやエディタなどの開発環境が整っているものとします。

カーネルの構築

計算マシンのネットワーク設定を一元管理するために BOOTP を用います。計算マシンのファイルシステムをネットワーク経由で提供するためには NFS を用います。これらを利用するために、以下に示す項目にチェックをしてカーネルを構築する必要があります。

```
CONFIG_INET_RARP=y  # RARP をサポートする
CONFIG_NFS_FS=y     # NFS をサポートする
CONFIG_NFSD=y       # NFS サーバをサポートする
```

BOOTP サーバの設定

BOOTP は「自ら OS を持たない端末がサーバに接続するためのプロトコル」です。サーバマシン上の /etc/bootptab に MAC アドレスとそれに対応する IP アドレスを記述しておくことで、静的に IP アドレスを割り当てることができます。DHCP を利用することも可能ですが、今回は BOOTP を利用します。Debian において BOOTP サーバは bootpd パッケージに含まれています。BOOTP の設定は、/etc/bootptab に計算マシンの NIC の MAC アドレスとそれに割り当てる IP アドレスを記述します。以下に設定例を示します。

/etc/bootptab の記述

```
.default:\n    :sm=255.255.255.0:\n    :gw=192.168.1.1:\n    :ds=202.23.xx.xx:\n    :ht=ethernet:\n    :bf=bootImage:\n    :hn:to=-18000:\n\ncompute1:ha=00A0A0A0A0A1:ip=192.168.1.11:tc=.default:hd=/tftpboot/192.168.1.1\ncompute2:ha=00A0A0A0A0A2:ip=192.168.1.12:tc=.default:hd=/tftpboot/192.168.1.1
```

記述の主な意味

sm: サブネットマスク

gw: ゲートウェイ

ds: DNS サーバ

bf: ブートファイル

ha: IP アドレスを割り当てる NIC の MAC アドレス

ip: 割り当てる IP アドレス

tc: グローバル・テンプレートの読み込み

hd: ブートファイルの置かれているディレクトリ

設定後に bootpd を起動する必要がありますが、一般的に bootpd は inetd 経由で起動されるようになっています。/etc/inetd.conf に以下の記述があるかどうかを確認し、なければ追加します。i オプションは inetd モードであることの指定、t オプションは BOOTP パケットを待つタイムアウト値を設定しています。

/etc/inetd.conf の記述

```
bootps  dgram  udp  wait  root  /usr/sbin/tcpd  /usr/sbin/bootpd -i -t 120
```

TFTP サーバの設定

TFTP (Trivial File Transfer Protocol) は、計算マシンが BOOTP によりネットワーク設定を行った後に、サーバマシン上にあるカーネルをダウンロードするために用います。TFTP は FTP と同様なファイル転送プロトコルですが、非常に単純化されています。Debian において TFTP サーバは tftpd パッケージに含まれています。一般的に tftpd も bootpd と同様、inetd 経由で起動されるようになっています。/etc/inetd.conf に以下の記述があるかどうかを確認し、なければ追加します。引数には TFTP によるアクセスを許可するディレクトリを指定します。

/etc/inetd.conf の記述

```
tftp  dgram  udp  wait  nobody  /usr/sbin/tcpd  /usr/sbin/in.tftpd  /tftpboot
```

NFS サーバの設定

Diskless Cluster では、各計算マシンが NFS を利用してサーバ上のファイルシステムをマウントします。Debian において NFS サーバは nfs-server パッケージに含まれています。インストール後、NFS サーバの設定ファイル (/etc/exports) の中に、各計算マシンにどのディレクトリをマウントさせるかを記述しておきます。これは、計算マシン間でどの程度ファイルを共有するかにも依存します。今回は、以下のように設定ファイルを記述します。ファイル中の no_root_squash は、root ユーザが NFS のク

クライアントマシン上においても root 権限を持つことができることを指定しています。さらに、root、home ディレクトリについては NFS 経由の書き込み権限を与えている点に注意して下さい。また、書き込み権限は必要なディレクトリに対してのみ与えるべきです。

/etc/export の記述

```
/usr          192.168.1.0/255.255.255.0(ro)
/bin          192.168.1.0/255.255.255.0(ro)
/lib          192.168.1.0/255.255.255.0(ro)
/sbin        192.168.1.0/255.255.255.0(ro)
/root        192.168.1.0/255.255.255.0(rw,no_root_squash)
/home        192.168.1.0/255.255.255.0(rw,no_root_squash)
/tftpboot/template 192.168.1.0/255.255.255.0(ro)
/tftpboot/192.168.1.11 192.168.1.11/255.255.255.0(rw,no_root_squash)
/tftpboot/192.168.1.12 192.168.1.12/255.255.255.0(rw,no_root_squash)
```

□ 計算マシン用のディレクトリの作成、およびシステムの設定

各計算マシン用のディレクトリをサーバマシン上に構築します。Cluster の管理を容易にするために、共有できるファイルはできる限り共有し、特定マシンのファイルを少なくするようにします。Cluster Computing の分野ではこうした Cluster の一元管理のように、相互接続されたコンピュータの集合を単一のリソースとして扱うことを SSI (Single System Image) と呼んでいます。管理者だけでなく、一般ユーザにとっても SSI が提供されることは、喜ばしいことだと思います。そこで今回構築する Diskless Cluster が少しでも SSI を実現できるように、ここでの作業を進めていきたいと思います。しかし、実際にはこの作業を徹底的に行おうとすると、Linux のファイル構成を大きく変更する必要が出てきてしまいます。構築の手間を考えると、ある程度のファイルは共有し、いくつかのファイルは重複させざるを得ないかもしれません。

主なディレクトリの作成

今、Debian GNU/Linux で構築されたシステムを見ると次のようになっています。

```
$ ls -l /

drwxr-xr-x  2 root  root      2048 Jul 25 21:41 bin
drwxr-xr-x  2 root  root      1024 Aug 11 15:55 boot
drwxr-xr-x  2 root  root      1024 May 26 21:11 cdrom
drwxr-xr-x  3 root  root     18432 Aug 25 14:57 dev
drwxr-xr-x 51 root  root      3072 Aug 25 14:57 etc
drwxr-xr-x  2 root  root      1024 May 26 21:11 floppy
drwxrwsr-x 11 root  staff     1024 Jun 29 15:45 home
drwxr-xr-x  2 root  root      1024 May 26 21:11 initrd
drwxr-xr-x  4 root  root      4096 Jul 25 21:41 lib
drwxr-xr-x  2 root  root     12288 May 27 05:06 lost+found
drwxr-xr-x  2 root  root      1024 Jun 29 16:01 mnt
dr-xr-xr-x 38 root  root         0 Aug 25 2000 proc
drwxr--r--  5 root  root      1024 Aug 23 11:40 root
drwxr-xr-x  2 root  root      2048 Jul 25 21:43/sbin
drwxrwxrwt  3 root  root      1024 Aug 25 15:27 tmp
drwxr-xr-x 14 root  root      1024 May 26 21:11 usr
drwxr-xr-x 14 root  root      1024 May 29 09:56 var
lrwxrwxrwx  1 root  root           20 May 26 21:53 vmlinuz -> /boot/vmlinuz-2.2.15
```

これらの中で、Diskless Cluster の計算マシンに必要なもの、またマシン間で共有すべきもの、そうでないものと分類を行います。分類を行った結果は以下に示す通りです。

計算マシンで必要ない	: cdrom floppy initrd lost+found mnt vmlinuz
Cluster 全体で共有	: bin lib sbin usr root home
個々に用意	: boot dev etc var
個々にディレクトリだけ用意	: proc tmp

Diskless Cluster ではサーバマシン上に、各計算マシン用のディレクトリを作成しますが、ここでは /tftpboot 以下にディレクトリを作成することにします。前節で示した NFS の設定ファイルからも分かるように、計算マシンは /tftpboot/IP_Address をルートファイルシステムとしてマウントします。Cluster 全体で共有するファイルも NFS を利用して共有するため、マウント・ポイントとなるディレクトリをあらかじめ準備しておく必要があります。

```
# mkdir -p /tftpboot/192.168.1.11
# cd /tftpboot/192.168.1.11
# mkdir bin lib sbin usr root home
```

このように非常に多くのファイルを、サーバマシンから参照します。ただし、各計算マシンを起動するために、最小限コピーしておかなければならないファイルがあります。これらは Linux の起動の順序に依存しますが、/sbin/init は最初に実行されるプログラムですので必ず必要になります。以下に、私が用意したバイナリの一覧を示します。これらを /tftpboot/IP_Address/sbin, /tftpboot/IP_Address/bin にコピーしておきます。

```
/sbin/init
/sbin/update
/sbin/getty

/bin/sh
/bin/mount
/bin/umount
/bin/uname
/bin/grep
/bin/rm
```

また、上記バイナリファイルの多くは、デフォルトでダイナミックリンクされています。/lib のいくつかのファイルを、同じように /tftpboot/IP_Address/lib にコピーしておきます。この時、ldd コマンドを使用することで、必要なライブラリを調べることができます。例えば、下記のようにコマンドを実行します。これより mount には libc.so.6 と ld-linux.so.2 が必要であることが分かります。

```
$ ldd mount
      libc.so.6 => /lib/libc.so.6 (0x40016000)
      /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

私の場合には、以下に示すライブラリが必要になりました。

```
ld-2.1.3.so
ld-linux.so.2 -> ld-2.1.3.so
libc-2.1.3.so
libc.so.6 -> libc-2.1.3.so
libdl-2.1.3.so
libdl.so.1 -> libdl.so.1.9.11
libdl.so.1.9.11
libdl.so.2 -> libdl-2.1.3.so
libncurses.so.5 -> libncurses.so.5.0
libncurses.so.5.0
```

次に個々に用意するディレクトリを作成します。dev は各マシンで別々に使われるので、サーバマシンの dev を丸ごとコピーします。ただし計算マシンの Root device を dev/nfsroot にするため、そのファイルがあるかどうかを確認します。なければ作成し、その後にコピーするのが良いでしょう。etc や var については個々に用意すべきファイルはあるものの、その多くを共有することができます。そこで、いったん/tftpboot/template 以下にコピーしておきます。boot ディレクトリはカーネルの System.map を置くだけです。System.map ファイルは、計算マシン用のカーネルを構築する時に作られます。ここでは、とりあえずディレクトリだけ作成しておきます。tmp や proc はディレクトリさえ提供しておけば問題ありません。

```
# cd /dev
# mknod nfsroot b 0 255
# cd /tftpboot/192.168.1.11
# tar cCf / - dev | tar xCfp /tftpboot/192.168.1.11 -
# mksplit template
# tar cCf / - etc var | tar xCfp /tftpboot/template -
# mkdir boot tmp proc
```

etc ディレクトリの作成と設定

現時点で etc ディレクトリは、/tftpboot/template/etc にあります。各計算マシン用の etc ディレクトリを/tftpboot/IP_ Address 以下に作成し、共有するファイルは/tftpboot/template/etc からのシンボリックリンクにします。注意すべきことは、/ディレクトリが/tftpboot/IP_ Address であると想定しなければならないことです。すなわち、以下のようにコマンドを実行します。

```
# cd /tftpboot/192.168.1.11/etc
# ln -s ../template/etc/hosts.allow .

あるいは

# cd /tftpboot/192.168.1.11/etc
# ln -s /template/etc/hosts.allow .
```

私の場合は、以下に示すファイル、ディレクトリをシンボリックリンクとして用意しました。

```
alternatives cron.d cron.daily cron.weekly group hostname hosts.allow
hosts.deny hosts.equiv inetd.conf ld.so.conf login.defs modules
networks nsswitch.conf pam.conf pam.d passwd protocols rc0.d rc1.d
rc2.d rc3.d rc4.d rc5.d rc6.d securetty serial.conf services
syslog.conf terminfo
```

また次のファイル、ディレクトリを各計算マシンごとに用意します。これらはカーネルが読み込まれる時に必要なファイルであったり、マシンごとに設定が異なるファイルであったりします。

```
default fstab hosts init.d inittab network rcS.d resolv.conf
```

計算マシンの設定に関して、最も注意すべき部分は起動と終了のスクリプトの設定です。これは、NFSによるマウントのタイミングを考えて設定を行って下さい。設定方法は各システムに依存しますので、以降で説明する事項を適宜読みかえて下さい。

起動の設定

起動では、etc/rcS.dディレクトリ以下のスクリプトのうち、ファイル名の先頭の番号が小さいものから順に実行されていきます。今はサーバマシンのものをコピーしただけの状態ですから、計算マシンに必要でないスクリプトを実行しないように修正を加えます。ただし、etc/rcS.d以下のスクリプトは全てetc/init.d以下にあるスクリプトのシンボリックリンクとなっていることに注意して下さい。

S30checkfs.shはfsckを実行するスクリプトです。すなわち、ディスクレスのマシンには必要ありません。ファイル自体を消去しても構いませんが、とりあえずファイルの先頭にexitを記述して、すぐにスクリプトを抜けるように設定しておきます。

S30checkfs.sh の記述

```
#
# checkfs.sh      Check all filesystems.
#
# Version:        @(#)checkfs  2.78  13-Nov-1999  miquels@cistron.nl
#

exit 0    # この1行を追加

. /etc/default/rcS
```

S35mountall.shは、元々ローカルに存在するファイルシステムをマウントするためのスクリプトです。しかし、ディスクレスマシンの場合には早い段階でファイルシステムをマウントしなければ、以下の起動スクリプトがうまく動作してくれません。そこで、ファイル中の以下の部分を修正します。mountコマンドのaオプションは、etc/fstabに指定されているファイルシステムを全てマウントするという意味です。

S35mountall.sh の記述

```
- mount -avt nonfs,noproc,nosmbfs

+ # mount -avt nonfs,noproc,nosmbfs
+ mount -av
```

この時に参照する各計算マシンの etc/fstab は次のように記述します。これは IP アドレスが 192.168.1.11 となる計算マシンの fstab です。

fstab の記述

```
# <file system> <mount point> <type> <options> <dump> <pass>
192.168.1.1:/tftpboot/192.168.1.11 /      nfs  rw  0  0
192.168.1.1:/usr                /usr   nfs  ro  0  0
192.168.1.1:/lib                /lib   nfs  ro  0  0
192.168.1.1:/bin                /bin   nfs  ro  0  0
192.168.1.1:/sbin              /sbin  nfs  ro  0  0
192.168.1.1:/root              /root  nfs  rw  0  0
192.168.1.1:/home              /home  nfs  rw  0  0
192.168.1.1:/tftpboot/template /template nfs  ro  0  0
none                            /proc  proc defaults
```

S40hostname.sh はホスト名を設定するスクリプトです。しかし、ここではホスト名を BOOTP から与えたいので、以下のように hostname コマンドをコメントアウトしておきます。また、/tftpboot/IP_Address/etc/hostname はテンプレートへのシンボリックリンクですが、中身を空にしておきます。

S40hostname.sh の記述

```
# hostname --file /etc/hostname
```

S40network はネットワークを設定するスクリプトです。今回、これら起動スクリプトは各マシンごとに用意するため、ここに各計算マシンの IP アドレスなどを記述できます。しかし、せっかく BOOTP サーバの設定をしているのですから、BOOTP を使って IP アドレスを取得しましょう。もちろん、こちらの方が多少管理の手間を省けます。S40network を以下のように記述します。bootpc コマンドは bootpc パッケージに入っています。

/etc/init.d/network の記述

```
#!/bin/sh

/sbin/bootpc --dev eth0 --server 192.168.1.1 --timeoutwait 20
ifconfig lo 127.0.0.1
route add -net 127.0.0.0 netmask 255.0.0.0 lo

exit 0
```

以上を設定すれば、なんとか計算マシンを起動することができるでしょう。もし何らかのエラーが出る場合は、スクリプトの実行順序を変えてみたり、いくつかのコマンドをコメントアウトしたりして解決を図って下さい。

終了の設定

既に何度も繰り返してきたように、各計算マシンはHDDをもっていない。マシンを終了させるためにいきなり電源を落したり、リセットをかけたりしてもそれほど問題がないかもしれません。しかし、リモートからClusterを管理するという観点では、手順を踏んだ終了を行いたいものです。一般にマシンを終了させるコマンドには、haltとrebootの2種類があります。etc/inittabの記述を見ると、Debianではhaltのランレベルは0、rebootのランレベルは6であることが分かります。つまりhaltに関してはetc/rc0.d、rebootに関してはetc/rc6.d以下のスクリプトが重要であることが分かります。

/etc/inittabの記述

```
# /etc/init.d executes the S and K scripts upon change
# of runlevel.
#
# Runlevel 0 is halt.
# Runlevel 1 is single-user.
# Runlevels 2-5 are multi-user.
# Runlevel 6 is reboot.
```

終了に関して最も注意すべき点は、先にファイルシステムをアンマウントしてしまうことにより、haltやrebootなどのコマンドが見つからず、途中でシステムがハングしてしまうことです。これを回避するために、ファイルシステムのアンマウントに関するスクリプトを最後尾にもってきます。S90haltスクリプトの後ろにもってきたスクリプトは、実際に実行されることはありません。

/etc/rc0.d以下のスクリプトの変更

```
[ 変更前 ]
S31umountnfs.sh
S40umountfs
S90halt

[ 変更後 ]
S90halt
S93umountnfs.sh
S95umountfs
```

/etc/rc6.d以下のスクリプトの変更

```
[ 変更前 ]
S31umountnfs.sh
S40umountfs
S90reboot

[ 変更後 ]
S90reboot
S93umountnfs.sh
S95umountfs
```

上記のように変更を加えると、ルートファイルシステムがアンマウントされたままシステムが終了してしまいます。さらに、終了時においてNFSの同期がしっかり取れているかどうかといった問題も

あります。これらに対処してもっとうまくやるには、上記のスクリプトの中身を修正する必要があると思います。私の方では試していませんが、余裕のある方は試してみてください。

var ディレクトリの作成

var ディレクトリも etc ディレクトリと同様に、共有するファイルは/tftpboot/template/var へのシンボリックリンクとして作成します。私の場合には、以下に示すディレクトリを各計算マシンごとに用意し、その他を共有することにしました。

```
var/run var/lock var/spool var/log
```

以上で、計算マシン用のディレクトリの作成はおおよそ終了です。2 台目以降の計算マシンの構築は、次のようにサーバマシン上で1 代目のマシンのディレクトリを丸ごとコピーし、各マシンに依存するファイルだけを修正すれば良いわけです。

```
# cp -a /tftpboot/192.168.11 /tftpboot/192.168.1.12
```

du コマンドを使って、各計算マシン用のディレクトリの合計サイズ (KB 単位) を表示させてみます。多くのファイルを共有したことにより、非常に小さなサイズとなっているのが分かります。もちろん、工夫を加えることでさらにサイズを小さくできると思います。

```
# du -sk /tftpboot/*
4058    /tftpboot/192.168.1.11
3267    /tftpboot/192.168.1.12
533     /tftpboot/bootImage
187048  /tftpboot/template
```

ユーザの共有

通常の PC Cluster では、NIS (Network Information Service) を利用してユーザの一元管理を行います。しかし、Diskless Cluster では計算マシンのファイルがサーバ上に存在するため、わざわざ NIS を使う必要がありません。今、/tftpboot/template/etc/passwd にパスワードファイルがコピーされています。そこで/etc/passwd をそれに対するシンボリックリンクとします。/etc/group も同様にシンボリックリンクとします。これより Cluster 全体でユーザ情報を共有できます。

```
# cd /etc
# rm passwd group
# ln -s /tftpboot/template/etc/passwd .
# ln -s /tftpboot/template/etc/group .
```

□ 計算マシン用のカーネルの作成

サーバマシン上で、計算マシン用のカーネルを作成します。カーネルのコンフィギュレーションでは最低限、以下の項目をチェックする必要があります。またこの際にモジュールではなく、カーネルに組み込まれるようにして下さい。

```
CONFIG_IP_PNP=y          # IP kernel level autoconfiguration をサポートする
CONFIG_IP_PNP_BOOTP=y    # BOOTP をサポートする
CONFIG_NFS_FS=y          # NFS をサポートする
CONFIG_ROOT_NFS=y        # Root on NFS を有効にする
CONFIG_EEEXPRESS_PRO100=y # 使用する NIC ドライバをサポートする
```

以下のようにコマンドを実行し、カーネルを構築します。

```
# cd /usr/src/linux
# make mrproper
# make menuconfig
# make dep clean
# make
# make bzImage
# make modules
# make modules_install
```

構築後、Netboot に付属している `mknbi-linux` コマンドを使ってカーネルイメージからブートファイルを作成します。作成したブートファイルは、`/etc/bootptab` ファイル中で指定したディレクトリにコピーしておきます。また、前節で作成した計算マシン用の `boot` ディレクトリに、作成された `System.map` をコピーするのを忘れないで下さい。

```
# cd /usr/src/linux/boot/
# mknbi-linux -d rom --i rom -k bzImage -o bootImage
# cp bootImage /tftpboot/
# cp /usr/src/linux/System.map /tftpboot/192.168.1.1/boot/
```

□ ブート

前章で述べたように、計算マシンのブートには Linux のカーネルがロードされる前にネットワーク設定を行うプログラム（起動 ROM バイナリ）を用います。起動 ROM バイナリを提供するソフトウェアとして、Netboot や Etherboot, NILO が代表的です。Netboot は主に Linux, MS-DOS をサポートし、Crynwr Packet Driver Collection によって提供されている NIC のドライバを多数利用しています。Netboot はサポートする NIC 数は多いのですが、ROM イメージのサイズが大きくなってしまふこと、autoprobe の機能がないことなどの欠点があります。Etherboot はそれらの欠点を解消したものといえますが、サポートする NIC 数は Netboot ほどではありません。Etherboot の欠点は NIC の各チップセットに対して、1 つ 1 つドライバを書かなければいけないこと、さらにこれらのドライバは Linux のドライバを流用できないことです。NILO (Network Interface LOader) は Netboot, Etherboot を引き継ぎ、これらの欠点を解消した新しいネットワーク・ブータであるといえます。

今回は、これらの中で Netboot を利用して起動 ROM バイナリを作成する方法を紹介します。Netboot を選択した理由は、現在の Debian のリリースにおいて Netboot だけが .deb ファイルとして提供されており、インストールの手間が省けるからです。しかし、もし余裕があるのならば NILO も試してみてください。

インストール手順について説明します。まず netboot パッケージをインストールします。次に `makerom` コマンドで NIC のドライバを組み込んだ起動 ROM バイナリを作成します。`makerom` コマンドを実行するといくつかの質問がなされるので、どの NIC のドライバを組み込むかはそこで指定してやり

ます。するとコマンドを実行したディレクトリに、image.flo と image.rom の2つのファイルができます。image.flo はFD用のバイナリ、image.rom はブートROM用のバイナリです。後は、あらかじめフォーマットしたFDにimage.floを書き込んで起動ディスクを作成します。

```
# apt-get install netboot
# makerom
# superformat /dev/fd0
# dd if=image.flo of=/dev/fd0
```

□ 動作確認

それでは実際にFDDに起動ディスクを挿し込み、計算マシンが正常に起動するか確認してみましょう。正常に起動した時、サーバマシン上の/var/log/syslogに記録されたログを以下に示しておきます。

サーバマシンに出力されたログ

```
server bootpd[749]: connect from 0.0.0.0
server bootpd[749]: version 2.4.3
server in.tftpd[750]: connect from 192.168.1.11
server tftpd[751]: tftpd: trying to get file: /tftpboot/192.168.1.11/bootImage
server mountd[184]: NFS mount of /tftpboot/192.168.1.11 attempted from 192.168.1.11
server mountd[184]: /tftpboot/192.168.1.11 has been mounted by 192.168.1.11
server mountd[184]: NFS mount of /usr attempted from 192.168.1.11
server mountd[184]: /usr has been mounted by 192.168.1.11
server mountd[184]: NFS mount of /lib attempted from 192.168.1.11
server mountd[184]: /lib has been mounted by 192.168.1.11
server mountd[184]: NFS mount of /bin attempted from 192.168.1.11
server mountd[184]: /bin has been mounted by 192.168.1.11
server mountd[184]: NFS mount of /sbin attempted from 192.168.1.11
server mountd[184]: /sbin has been mounted by 192.168.1.11
server mountd[184]: NFS mount of /root attempted from 192.168.1.11
server mountd[184]: /root has been mounted by 192.168.1.11
server mountd[184]: NFS mount of /home attempted from 192.168.1.11
server mountd[184]: /home has been mounted by 192.168.1.11
server mountd[184]: NFS mount of /tftpboot/template attempted from 192.168.1.11
server mountd[184]: /tftpboot/template has been mounted by 192.168.1.11
```

もし計算マシンがうまく起動しない場合には、次のことを順に確認してください。

(1) BOOTP サーバが応答しない

- /usr/sbin/bootptest コマンドを利用して、サーバの応答性を確認する。
- bootpd デーモンを-d4などのデバッグオプションを付けて起動し、ログを確認する。

(2) TFTP を利用してカーネルイメージがダウンロードできない

- /var/log/syslog に出力されるログを見る。多くは/etc/bootptab に記述したブートファイルの置き場所を間違えて指定しているか、TFTP アクセスを可能にするディレクトリを間違えているか指定しているかが原因である。

(3) 途中でカーネルパニックを引き起こす

- カーネル、そしてブートファイルが適切に作られているか確認する。
- 計算マシン用のディレクトリが正しく作成されているか確認する。

(4) ルートファイルシステムのマウントに失敗する

- 適切にカーネルが作られているか確認する。
- /etc/exports の記述が適切であるか確認する。
- /tftpboot/IP_Address/etc/fstab の記述が適切であるか確認する。
- /var/log/syslog に出力されるログを見る。

(5) (4) 以降でエラーが起こる

- 起動スクリプトが適切であるか確認する。
- 起動に必要な最小限のファイルが計算マシン上に準備してあるか確認する。

本章では Diskless Cluster の構築について説明してきました。繰り返しますが、設定ファイルに関する部分はシステムによって異なる上、今後も大きく変更される可能性があります。その OS についてある程度の知識や情報がなければ、なかなか難しいかもしれません。しかし、後述する参考文献に示しているように、ディスクレスマシンや Linux についてのドキュメントは Web 上に多数あります。さらに次章では、より優れた Cluster を構築するためのヒントを示します。これらを利用して、私が紹介したものよりも「管理がし易く、安定して動き、性能の良い Cluster」を皆様が構築できることを願っています。

■ より高機能・高性能に

本章では、構築した Diskless Cluster に対して「管理の容易さ」「性能向上」を図るためのヒントを紹介します。構築される Cluster の数が増加するにしたがって、こうした情報が数多く公開されるようになっていきます。

□ リモートの電源管理

リモートの電源管理は、特に Diskless Cluster に限った話ではありません。しかしこれまで説明してきた Diskless Cluster では、一元管理を 1 つの目標にしてきました。そこで、計算マシンの電源管理もサーバマシンから行いたいと思います。電源管理には ON/OFF がありますが、それぞれについて以下で説明を行います。

電源 ON

リモートから電源を ON するには Wake On LAN という仕組みを用います。Wake On LAN とはネットワーク経由でマシンを起動する仕組みであり、これを用いることで Cluster 内の計算マシンを一斉に起動することが可能になります。Wake On LAN を行うには、マザーボードと NIC、電源が Wake On LAN に対応している必要があります。そして BIOS で Wake On LAN を有効にしておく必要があります。

Wake On LAN では、リモートから MagicPacket と呼ばれる起動用パケットを NIC に送りつけます。NIC が MagicPacket を受け取ると、電源が ON になりマシンが起動されるわけです。詳しくは MagicPacket のドキュメントなどを参照して下さい。MagicPacket は、FF を 6 バイト繰り返した後に、MAC アドレスを 16 回繰り返す計 102 バイトが含まれるパケットです。以下に神戸大学の児玉さんが作成された MagicPacket を送信するスクリプトを示します。児玉さんのページ (<http://www.math.kobe-u.ac.jp/~kodama/index.html>) には、この他に ping プログラムに修正を加えて MagicPacket を送信する方法などが紹介されています。

ruby を利用したスクリプト

```
#!/bin/bash
# Wake On LAN. K.Kodama 1999-11-29 2000-05-13
# line format of hw-addresses: ip hostname hw-address
# e.g. 192.168.1.10 host10 0A:0A:0A:0A:0A:0A

PROG='basename $0'

function HELP(){ cat<< EOF
Wake up on LAN
Usage1: $PROG [-b broadcast] -a          wake up all
Usage2: $PROG [-b broadcast] host        wake up the host
Usage3: $PROG [-b broadcast] -s          read MAC list from stdin
EOF
}

function SendMagic(){
/usr/local/bin/ruby -e 'require "socket"
if ARGV.size>=1; adr=ARGV[0]; ARGV.clear; else adr=""; end
s=UDPSocket.open(); s.setsockopt(Socket::SOL_SOCKET,Socket::SO_BROADCAST,1)
while gets
  l=$_sub(/\s+/, "").split(/\s+/)
  ip=l[0]; name=l[1]; hw=l[2]; hs=hw.split(/:/).pack("H*H*H*H*H*H*")
  if adr==""; # Set adr as broadcast. A:1--127, B:128--191, C:192--223
    ips=ip.split(/\./);c=ips[0].to_i
    if c<=127; ips[1]="255";end
    if c<=191; ips[2]="255";end
    if c<=223; ips[3]="255";adr=ips.join(".");else adr="";end
  end
  2.times{ s.send((0xff.chr)*6+hs*16,0,adr,"discard") }
end' $@
}

if [ "$PROG" = "wakeup1" ];then macfile="/mac-list1";broadcast=""
elif [ "$PROG" = "wakeup2" ];then macfile="/mac-list2";broadcast=""
else macfile="/arp-sample";broadcast=""
fi
if [ "$1" = "-b" ];then broadcast=$2; shift 2;fi
if [ "$1" = "-h" -o "$1" = "" ];then HELP
elif [ "$1" = "-a" ];then SendMagic $broadcast < $macfile
elif [ "$1" = "-s" ];then SendMagic $broadcast
elif grep $1 $macfile>/dev/null;then grep $1 $macfile|SendMagic $broadcast
else HELP
fi
```

上記のスクリプトは、次に示す MAC アドレスが記述されているファイル
(~/arp-sample)を読み込み、MagicPacket をブロードキャストします。このスクリプトを用いるこ
とにより、計算マシンの一斉起動を実現することができます。

~/arp-sample の記述

```
192.168.1.11 compute1 00:A0:A0:A0:A0:A1
192.168.1.12 compute2 00:A0:A0:A0:A0:A2
```

電源 OFF

リモートから電源 OFF を行うには、自動電源 OFF の機能とリモートから電源 OFF の命令を実行できる機能が必要です。自動電源 OFF を行うには、まずマザーボードと電源が **APM (Advanced Power Management)** に対応している必要があります。そして BIOS で APM を有効にします。Linux のカーネルは APM BIOS 対応を有効にし、該当する項目をチェックして構築する必要があります。カーネル 2.2.15 においては、以下の設定で終了時に電源を OFF にできたという報告があります。

```
CONFIG_APM=y
# CONFIG_APM_DISABLE_BY_DEFAULT is not set
# CONFIG_APM_IGNORE_USER_SUSPEND is not set
# CONFIG_APM_DO_ENABLE is not set
# CONFIG_APM_CPU_IDLE is not set
# CONFIG_APM_DISPLAY_BLANK is not set
# CONFIG_APM_IGNORE_MULTIPLE_SUSPEND is not set
# CONFIG_APM_IGNORE_SUSPEND_BOUNCE is not set
# CONFIG_APM_RTC_IS_GMT is not set
# CONFIG_APM_ALLOW_INTS is not set
CONFIG_APM_REAL_MODE_POWER_OFF=y
```

しかしながら APM の自動電源 OFF については、これ以上正確なことを述べることはできません。上記の設定ではなく、lilo.conf でカーネルに apm のパラメータを与えたり、CONFIG_APM_REAL_MODE_POWER_OFF をセットせずにカーネルを作成した場合に、自動電源 OFF が動作したという例も報告されています。その他のドキュメントを参照しながら、各自の環境に合わせた設定を行うようにして下さい。APM ではなく、**ACPI (Advanced Configuration and Power Interface)** を利用するという方法もありますが、その時には開発バージョンのカーネルを使う必要があります。

電源 OFF の命令は、shutdown コマンドを rsh を利用して発行します。一般的な Cluster では rsh (あるいは ssh) が必須であるため、rsh の設定は既になされているはずで、これを利用するのが良いと思います。ただし、Debian では root 権限での rsh を許可していません。root 権限での rsh を有効にするには、計算マシンの rshd を以下のオプションを付けて起動するようにします。rshd も通常は inetd 経由で起動されるので、etc/inetd.conf を次のように修正します。

/etc/inetd.conf の中身

```
shell  stream  tcp  nowait  root  /usr/sbin/tcpd  /usr/sbin/in.rshd -h
```

rsh を利用したリモートからのシャットダウンの例

```
# rsh compute1 shutdown -h now
```

(注意)

上記は古い rsh-server の設定例です。少なくとも Debian の rsh-server のバージョンが 0.10-7 以降で、上記の設定を試すと次のようなエラーが syslog に出力されます。すなわち、rshd には h オプションがなくなり、代わりに PAM で設定を行わなければならないということです。

```
server in.rshd[138]: connect from 192.168.1.11
server rshd[138]: -l and -h functionality has been moved to pam_rhosts_auth
                  in /etc/pam.conf
```

pam.conf および pam.d は共有されるので、/tftpboot/template/etc/pam.d/rsh ファイルに次の記述を加えてやります。これで root での rsh が使えるようになるはずです。PAM についての詳細は、<http://riksun.riken.go.jp/archives/Linux/kernel.org/libs/pam/> などのドキュメントを参照して下さい。

```
#%PAM-1.0
auth    sufficient      pam_rootok.so          # この1行を追加
auth    required        pam_rhosts_auth.so
auth    required        pam_nologin.so
auth    required        pam_env.so
account required       pam_unix_acct.so
session required       pam_unix_session.so
```

またこれらを利用するにあたり、root 権限での rsh がセキュリティ上、非常に危険であることは理解しておいて下さい。

□ Cluster NFS

Cluster NFS は Universal NFS Daemon サーバにパッチを当てて、同じルートファイルシステムを複数のディスクレス・クライアントで共有可能にするソフトウェアです。Cluster NFS では同じファイル名をもつファイルに対して、共有されるべきファイル、特定マシンのファイルを次のように区別することができます。

```
共有されるファイル：sample.txt
特定マシンのファイル：sample.txt$$IP=192.168.1.11$$
```

Cluster NFS は、通常の NFS とほとんど設定が同じであるため、容易に導入することができます。Cluster NFS をうまく使うことで、重複ファイルを減らし、スムーズにファイルの共有ができるのではないかと思います。

□ NFS Performance Benchmarks

Diskless Cluster では NFS が非常に重要な役割を果たしています。並列プログラムの性能に直接関わることはありませんが、実行モジュールのロードや各計算マシンの起動時など、ユーザが体感しやすい部分に NFS が関わってきます。小規模の Cluster の場合には気にならなくても、大規模になると NFS の性能の影響を大きく受けるようになるかもしれません。これを解決するために、Cluster 内の NFS の性能を測定し、NFS のパラメータを適切に設定する必要があるかもしれません。計算マシンの etc/fstab の記述において

```
192.168.1.1:/tftpboot/192.168.1.11 / nfs rsize=8192,wsiz=8192 0 0
```

のように rsize と wsiz を指定してみましょう。値は 1024 の倍数で 16384 を越えない範囲で設定します。値を変更すると、Linux のカーネルと NIC の組み合わせによって NFS の性能が変わってきます。

す。以下に述べる簡単なベンチマークを行うことで、この値を適切に設定します。

NFS のベンチマークプログラムとしては、SPEC が開発した LADDIS ベンチマーク (Keith93) があります。これはシミュレートされたクライアントの負荷に対する NFS サーバの性能を測定します。しかし LADDIS は商用なので、実際にはなかなか利用できないでしょう。Debian では、nhfsstone という NFS のベンチマーク・ソフトウェアが用意されています。nhfsstone は nhfsstone パッケージに含まれています。nhfsstone プログラムを計算マシン上で実行すると、次のような結果が得られます。

```
# nhfsstone
nhfsstone: INVALID RUN, mix generated is off by 43.64%
op      want      got      calls      secs  msec/call  time %
null      0%      0.00%        0      0.00      0.00      0.00%
getattr  13%     14.27%      715      0.04      0.06      1.43%
setattr   1%     10.68%      535      0.00      0.00      0.09%
root       0%      0.00%        0      0.00      0.00      0.00%
lookup   34%     42.65%     2136      0.32      0.15     10.26%
readlink  8%      8.98%      450      0.40      0.90     13.03%
read     22%      2.57%      129      1.30     10.13     41.69%
wrcache   0%      0.00%        0      0.00      0.00      0.00%
write    15%     16.13%      808      0.42      0.52     13.42%
create    2%      1.05%        53      0.37      7.11     12.01%
remove    1%      1.05%        53      0.03      0.73      1.24%
rename    0%      0.00%        0      0.00      0.00      0.00%
link       0%      0.00%        0      0.00      0.00      0.00%
symlink   0%      0.00%        0      0.00      0.00      0.00%
mkdir     0%      0.00%        0      0.00      0.00      0.00%
rmdir     0%      0.00%        0      0.00      0.00      0.00%
readdir   3%      1.53%        77      0.20      2.59      6.37%
fsstat    1%      1.03%        52      0.01      0.25      0.41%
169 sec 5008 calls 29.63 calls/sec 0.62 msec/call
```

特別なベンチマーク・プログラムを用いなくても、簡単なコマンドを実行しておおまかな性能を計測することも可能です。次に示すのは、中身が 0 のバイト列で埋めつくされた 64MB のファイルの書き込みの性能のテストです。

```
$ time dd if=/dev/zero of=/tmp/testfile bs=16k count=4096
```

次に、このファイルを再度読み込みすることで、読み込み性能のテストを行います。

```
$ time dd if=/tmp/testfile of=/dev/null bs=16k
```

何度か平均をとって測定し、最も性能の良かった rsize, wsize を設定すれば良いわけです。

□ ローカルディスクの使用

これまで説明してきたように、Diskless Cluster ではファイルアクセスの多いジョブを実行する場合に性能が低下してしまいます。この問題を改善するためには、各計算マシンにローカルディスクを取り付けるという対処が考えられます。Diskless Cluster とは言えなくなりますが、tmp や scratch などのワーキングディレクトリ、そして swap 領域を個々のローカルディスク（一般に HDD）上に作成します。ジョブの実行前にデータファイルをローカルディスク上のディレクトリにコピーしておき、実行後に回収することで高速にジョブを実行することが可能になります。また、ローカルディ

スクは作業用としてのみ使用するため、計算マシンのシステム管理はサーバマシン上で行うことが可能です。

■ おわりに

実際に Diskless Cluster を構築してみると、計算マシン内で作業することはほとんどなく、サーバマシン内で多くの作業を行うのが分かるかと思います。そうすると Cluster での SSI が、思いのほか容易なのではないかと思えるかもしれません。今後、Cluster をより有効に利用していくには構築・管理の容易さが大切になってきます。私はこうした Diskless Cluster の一元管理の仕組みが、より簡単に利用できるようになれば良いと思います。現在利用されている Linux のファイルシステムや起動の仕組み、NFS といったソフトウェアなどは汎用的に作られています。そのために、Cluster で利用するとどうしても不便な場合があります。Cluster 用の Linux ディストリビューションや Cluster NFS などのプロジェクトに期待したいと思います。また、次章では主な Cluster の構築・管理ソフトウェアを紹介します。これらはディスクレスであるなしに関わらず、今回紹介したような Cluster を構築する一連の作業を自動的に行ってくれます。こうしたソフトウェアは非常に有効であり、Cluster を構築するための手間を減らすだけでなく、性能の良い Cluster を構築することができます。今後、これらの開発がさらに進み、今以上に簡単に Cluster を構築できるようになれば良いと思います。

■ Cluster の構築・運用に関するリソース

- FAI (Fully Automatic Installation)

FAI は Linux Cluster を構築するために、各計算マシンに Debian GNU/Linux を自動インストールするためのツールです。まず Diskless Cluster のサーバマシンに似た FAI のサーバマシンを用意します。次に、計算マシンのシステムを構築するための configure ファイルと起動ディスクを作成します。そして作成した起動ディスク (FD) を用いて計算マシンを起動します。ここまでは Diskless Cluster とほぼ同じですが、FAI においてこれはインストールの 1 ステップにすぎません。FAI では、この後に計算マシンのローカルディスクに対して、configure ファイルで指定したパッケージをインストールします。インストール後、計算マシンはローカルディスクより起動されるようになります。FAI ではユーザが行う主な作業は、最初の configure ファイルの作成です。それ以降は、ほぼ自動で個々の計算マシンのインストール作業が行われます。

<http://www.informatik.uni-koeln.de/fai/>

- ALINKA LCM (Linux Cluster Manager)

ALINKA LCM は Linux Cluster を構築・管理するためのツールです。ALINKA LCM は ALINKA RAISIN と呼ばれる商用ソフトウェアの GUI 部分を除いたものであり、オープンソースで提供されています。Cluster を構築するには、まずサーバマシンに ALINKA LCM をインストールし、設定を行います。計算マシンへの OS のインストール、および管理は、サーバマシン上の ALINKA LCM を通して行います。ALINKA LCM で構築される計算マシンは、サーバマシンのクローンとなります。また現在、ALINKA LCM と Linux のインストーラを組み合わせたオリジナルのディストリビューション (Debian GNU/Linux VT-PC Cluster) を VT の中田さんが作成されています。

<http://www.alinka.com/>

- SCore Cluster System Software

SCore は Workstation, および PC Cluster において, 高度な並列計算を行うための環境を提供します. SCore による Cluster の構築はほぼ自動化されています. インポートホストとなる通常の Linux BOX に SCore をインストールし, GUI 上で Cluster の設定を行います. そして, サーバマシンと計算マシン用の起動ディスクを作成します. インストール対象となるマシンを, 作成した起動ディスクを用いて起動すれば, 後はネットワーク経由で必要なパッケージがインストールされる仕組みです. SCore を用いた Cluster の構築は非常に便利ですが, SCore により提供されている Cluster の運用ツールはそれ以上に充実しています. Myrinet などの高性能ネットワークのサポート, Cluster 内のジョブスケジューリング, フォールトトレランス, 容易に並列プログラミングを行うことのできる環境などが提供されています. SCore を利用することで, 本格的な PC Cluster を容易に構築することができます.

<http://pdswww.rwcp.or.jp/>

■ 謝辞, およびフィードバック

本ドキュメントの作成, および講習会での講演にあたり, 多大なご協力を頂いた三木・廣安先生に心より感謝いたします. そして神戸大学の児玉様, Linux の開発・ドキュメントの作成に関わっていらっしゃる多くの方々に深く感謝いたします. 最後に, 様々な作業を手伝って頂いた三木研究室の学生の皆様に感謝いたします.

本ドキュメントの作成については細心の注意をはらいましたが, 私の力不足により間違いや分かりにくいところなどがあるかもしれません. このドキュメントに関して訂正・指摘・質問などがございましたら, 谷村 (tanisuke@mikilab.doshisha.ac.jp) 宛にメールを頂ければ幸いです. (2000 年 8 月 30 日版)

■ 参考文献

- 1) Netboot. <http://www.han.de/~gero/netboot/>.
- 2) Etherboot. <http://etherboot.sourceforge.net/>.
- 3) NILO. <http://nilo.sourceforge.net/>.
- 4) Linux Start. <http://www.linuxstart.com/applications/boot.html>.
- 5) Cluster NFS. <http://clusternfs.sourceforge.net/>.
- 6) Diskless Linux Mini Howto. <http://www.linux.or.jp/JF/JFdocs/Diskless.txt>.
- 7) NFS Howto. <http://www.linux.or.jp/JF/JFdocs/NFS-HOWTO.txt>.
- 8) NFS Root Client Mini Howto.
<http://www.linux.or.jp/JF/JFdocs/NFS-Root-Client.html>.
- 9) NFS Root Mini Howto. <http://www.linux.or.jp/JF/JFdocs/NFS-Root.txt>
- 10) 児玉宏児（神戸大学理学部）. リモートで電源 On/Off.
<http://www.math.kobe-u.ac.jp/~kodama/tips-WakeOnLAN.html>.
- 11) Aoyama Plus Project. <http://www.phys.aoyama.ac.jp/~naoya/>.
- 12) 清水尚彦. Linux/Alpha 活用講座 第 21 回 完全ディスクレスマシンの構築.
Linux Japan 2000 年 8 月号.
- 13) 野澤恵（茨城大学理学部）.
reno project. <http://www.env.sci.ibaraki.ac.jp/~snozawa/reno/>.
- 14) Linux Biscuit のおいしい食べ方. <http://www.research.co.jp/biscuit/index.shtml>.
- 15) 増田景一, 坂上仁志, 高橋豊（姫路工業大学情報工学科）.
ローエンド PC を用いた PC クラスタの構築. 第 18 回超並列計算研究会.
- 16) 小坂隆浩（大阪産業大学）. 実際の応用. PC クラスタ超入門, 1999.
- 17) Standard Performance Evaluation Corporation. <http://www.spec.org/>.

PC クラスタにおけるベンチマークと ジョブ管理

同志社大学 大学院 工学研究科

吉田 純一

junichi@mikilab.doshisha.ac.jp

■ はじめに

PC クラスタシステムはコストパフォーマンスの高い並列計算環境といえますが、その性能を十分に発揮するためにはいろいろと工夫が必要になります。まず第 1 に、構築したクラスタシステムの特徴をよく知らなくてはなりません。さらに、効率よくクラスタを運用するためにはジョブの管理も重要になります。そこで、本章ではクラスタシステムを評価するためのベンチマークプログラムを紹介するとともに、実際の測定結果をもとに PC クラスタのコストパフォーマンスについて考えてみます。また、本章の後半ではクラスタシステムを有効に利用するためのジョブ管理の一般的な概念と DQS を利用したジョブ管理について解説します。

■ ベンチマークとは

ベンチマーク (benchmark) という語には、一般的に「基準となるもの」という意味があります。しかしながら、コンピュータ用語としてのベンチマークとは、コンピュータのハードウェアやソフトウェアの処理速度を計測する試験問題、およびそれによる性能評価という意味を持ちます。つまり、**計算機システムの性能を計測すること**自体を指す言葉として用いられているのです。

それでは、なぜベンチマークによる性能の評価が重要なのでしょうか。PC クラスタの特徴を一言で表すならば「誰もが比較的簡単に構築可能なコストパフォーマンスの高い並列計算環境」ということができるでしょう。けれども、実際にその性能を測定しなければ、コストパフォーマンスの善し悪しを判断することはできません。これまでの専用並列計算機の場合にはベンダーと特定の計算機環境が一意に対応していたため、各ベンダーが計算機の性能指標を呈示していました。しかし、汎用の PC を組み合わせて自分で構築する PC クラスタシステムにおいては、その性能を誰も呈示してはくれないのです。自分で構築したクラスタシステムの性能は自分で測定してやらなければなりません。本章で紹介するような汎用的なベンチマークを行うことで、スーパーコンピュータとの性能比較も可能になります。

また、さまざまなベンチマークを行うことによって、そのシステムの長所や短所が明確となるという点もベンチマークの重要な側面です。ベンチマークの結果をもとにして、効率の良いシステムの運

用方針や改良の計画を決定することができます。このような理由から、構築したクラスタシステムの性能評価は極めて重要なのです。

■ ベンチマークプログラムの分類¹⁾

コンピュータは、プログラムによってさまざまな挙動を示すうえ、その用途もさまざまあるため、比較の指標を一意に決めることは困難です。そこで、**各用途に応じた性能評価の指標を決定し、その尺度にもとづいて性能を評価する**ということが重要となります。たとえば、一般ユーザ向けの性能評価と高度に専門的な用途向けの性能評価では、評価の指標も自ずから変わってきます。このため、用途に応じた様々なレベルのベンチマークが必要とされており、提供されています。

ベンチマークにはさまざまなものがありますが、これらを大きく分類すると、マクロベンチマーク（総合型ベンチマーク）とマイクロベンチマーク（分野別ベンチマーク）の2種類に分類できます。

マクロベンチマークではシステム全体の性能を評価することができます。このため、異なる構成のシステム間の比較指標としての利用が可能で、クラスタのコストパフォーマンスを考える上では有益だといえます。このクラスタとあのクラスタではどちらが速いか？といった単純な要求に対してはマクロベンチマークが有効です。しかし、マクロベンチマークでは、全体的な性能は評価できても、なぜ性能が良いのか、または悪いのかという点については知ることができないという欠点があります。一方、マイクロベンチマークで評価されるのは、CPUの速度やネットワークの性能といったシステムの特徴の側面のみ限定されます。一般にマイクロベンチマークの対象となるのは次のようなものです。

- **演算性能ベンチマーク**

計算機の演算性能についての評価基準を提供する

- **I/O 性能ベンチマーク**

計算機の I/O 性能、特にディスク・サブシステムについての評価基準を提供する

- **グラフィックス関連性能ベンチマーク**

ウィンドウ・システムなどのディスプレイへのイメージ描画性能に関する評価基準を提供する

- **ネットワーク関連性能ベンチマーク**

システムのネットワーク性能に対する評価基準を提供する

- **データベース性能ベンチマーク**

データベース・アプリケーションに特有な動作であるトランザクション処理能力に対する評価基準を提供する

- **アプリケーション指向ベンチマーク**

システムにおけるアプリケーションの実行性能に対する評価基準を提供する

- **並列/大規模計算ベンチマーク**

並列計算の実行性能に対する評価基準を提供する

これらの中で、PC クラスタシステムの性能評価において重要なのは、演算性能、並列計算、ネットワーク性能、アプリケーション性能などです。これらについては、次節以降でいくつかのベンチマークプログラムを紹介していきます。

マイクロベンチマーク、マクロベンチマークともに多くのベンチマークプログラムが公開されており、また利用されています。その数は代表的なものだけでも数 100 以上といわれています。そのなかでもハイパフォーマンスコンピュータのベンチマークとして特に有名なものを表 4.1 に挙げてみました。

タイプ	プログラム名	評価対象
マイクロ ベンチマーク	LINPACK	演算能力
	LMBENCH	システムコールと データの移動
	STREAM	メモリのバンド幅
マクロ ベンチマーク	NPB	並列演算性能
	PARKBENCH	並列演算性能
	SPEC	さまざまなベンチマーク
	Splash	並列演算性能

表 4.1: 代表的なベンチマークとその分類¹⁾

□ 演算性能ベンチマーク

演算性能ベンチマークは、計算機の演算性能についての評価基準を提供するもので、一般のユーザーの関心が最も高いところと言えるでしょう。一般に、計算機の演算性能は、(1) 整数演算性能と (2) 浮動小数点演算性能に大別ですが、これを決定するのは主に (A) CPU の性能 (CPU アーキテクチャ) と (B) 計算機システムのアーキテクチャです。これを測定するためには、CPU のクロック周波数だけでなく、メモリの速度やキャッシュの量、FSB(Front Side Bus) の周波数などさまざまな要素について検討が必要になります。そこで、システム全体の演算性能を評価するという目的においては、特定のアプリケーションを実行したときの速度を測定し、それを比較するのが一般的な評価方法になっています。

LINPACK / ScaLAPACK⁴⁾

LINPACK ベンチマークは、テネシー大学の Jack Dongarra によって開発された、線形代数問題の LU 分解を行うベンチマークです。ここでは LINPACK をベンチマークとして取り上げていますが、実際には線形代数問題の LU 分解を行うためのサブルーチンの集合であり、ベンチマークにも利用されていると表現の方が正確です。LINPACK 自体は非常に単純なものですが、システムの数値演算 (単精度および倍精度の浮動小数点演算、特にベクトル演算) 能力を測定するのに非常に有効であるとされています。

このため、LINPACK は世界のハイパフォーマンスコンピュータのランキング「TOP500 Supercomputer」のベンチマークにも採用されています。最新の TOP5 の性能を表 4.2 に示します。表にはいくつかの項目がありますが、 R_{max} は演算速度、 N_{max} は R_{max} を記録したときの問題サイズです。これに対して $N_{1/2}$ は、 R_{max} の半分の速度に達した時の問題サイズです。また、 R_{peak} は、最高速度です。

LINPACK はもともと単一 CPU 用のライブラリであったが、現在では並列バージョンである ScaLAPACK が広く用いられています。ScaLAPACK では、行列を分割し、それぞれを 1 つのプロセスが担当し、プロセス間通信には MPI もしくは PVM を用いることで並列化を実現しています。このため PC クラスタのような並列計算環境でも実行・計測が可能になっています。

姫野ベンチ (Himeno Benchmark)⁵⁾

姫野ベンチマークは理化学研究所の姫野龍太郎氏が非圧縮流体解析コードの性能評価のために考案したベンチマークです。このベンチマークはポアソン方程式をヤコビの反復法で解く場合に主要なループの処理速度を計るものです。したがって、流体計算の性能評価に最適なベンチマークであるといえます。

このベンチマークの特徴はコードが非常に短く、簡単にコンパイル・実行でき、即座に実測速度を

表 4.2: 2000 年 7 月における TOP 5 の演算性能

Rank	Computer	# Proc	R_{max} (Gflop/s)	R_{peak} (Gflop/s)	N_{max} (order)	$N_{1/2}$ (order)
1	ASCI Red	9632	2379.6	3207	362880	75400
2	ASCI Blue-Pacific SST	5808	2144	3868	431344	.
3	ASCI Blue Mountain	6144	1608	3072	374400	138000
4	SP Power3 375 MHz	1336	1417	2004	374000	.
5	SR8000-F1/112	112	1035	1344	120000	15160

求められることです。本章で取り上げるベンチマークの中でも最も手軽に実行できるものといえます。また、Windows や Machintosh 用のバイナリも配布されており、身近なマシンとの性能比較も容易です。姫野ベンチマークの eb サイトにはさまざまなシステムでの測定結果が掲載されていますので、それらとの性能比較を行うこともできます。

□ 並列計算ベンチマーク

並列計算環境全体の性能評価では、各ノードの計算性能に加えて通信の頻度や通信量などの要素も考慮する必要があります。それらを総合的に評価することのできるベンチマークとして NPB があります。

Nas Parallel Benchmark Suite ^{?)}

NAS Parallel Benchmarks (NPB) は、NASA Ames Research Center の Numerical Aerodynamic Simulation (NAS) プログラムで開発された、並列コンピュータの性能評価のためのマクロベンチマークです。NPB では、主に大規模な CFD Application における計算とデータの送受信性能を評価することができます。

また、ベンチマークの結果は NASA Ames Research Center より定期的に WWW などで行なわれているレポート (NAS Parallel Benchmark Results) に反映されます¹⁾。このレポートにはプログラム実行に要した経過時間 (elapsed time) と、Y-MP/1 または C90/1 との性能比などについて報告がなされており、大規模計算機と PC クラスタの性能の比較を行うことができます。NPB は、並列コンピュータの実行性能を知るうえで、権威あるベンチマークの 1 つであるといえます。

NPB は、5 つの Parallel Kernel Benchmark (EP, MG, CG, FT, IS) と 3 つのアプリケーション (LU, SP, BT) から構成されています。このうち、IS は整数演算のベンチマークですが、そのほかのものはすべて浮動小数点演算のベンチマークです。それぞれの問題に対し、問題サイズや反復回数が異なる 5 つのクラス²⁾が定義されており、それぞれの実行環境に応じたベンチマークを行うことができます。各ベンチマークの特徴は以下のようになっています。

The Embarrassingly Parallel (EP) Benchmark : EP ベンチマークは、並列計算に適しており、指定されたスキームに従い生成された、多数のガウス分布に従う擬似乱数を用いて、2 次元の統計情報を蓄積するものです。この問題は、モンテカルロ法を用いた応用プログラムによく見られます。この問題を解く際に転送をほとんど必要としないことから、ある意味において本ベンチマークは、システムがもつ浮動小数点演算の最大実効性能を示すものと言えます。

Multigrid (MG) Benchmark : MG ベンチマークは、3 次元ポアソン方程式を簡略化したマルチ

¹⁾残念ながら最近はあまり更新されていないようです

²⁾S(sample), W(orkstation), A, B, C の 5 つで C が一番難易度が高い

グリッド法で解くものです。Class B では、Class A と同じ大きさの格子を使用していますが、内部のループの繰り返し数が Class A より大きくなっています。

Conjugate Gradient (CG) Benchmark：CG ベンチマークは、大規模で正値対称な疎行列の最小固有値の近似値を共役勾配法を用いて解くものです。このカーネルは非構造格子を用いたアプリケーションでよく見られます。

3-D FFT PDE (FT) Benchmark：FT ベンチマークは、3次元偏微分方程式を FFT を用いて解くものです。このカーネルはスペクトル法を使用したアプリケーションの典型的なところを評価しているといえます。このベンチマークは、ライブラリルーチンの使用が認められている点において、ほかのものと性質が少し異なります。

Integer Sort (IS) Benchmark：IS ベンチマークは、パーティクル法を使用したアプリケーションにおいて重要なソートを評価するものです。このタイプのアプリケーションは、物理におけるパーティクルをあるセルに割り当てて、パーティクルがセルから流れ出るかどうかを見る、particle-in-cell 法のアプリケーションに似ているといえます。ソートは、パーティクルを再び適切なセルに割り当てる際に使用されます。

Simulated CFD Application Benchmarks：LU(Block lower triangular, block upper triangular), SP(Scalar Penta-diagonal) ベンチマークはいずれも Navier-Stokes 方程式の解法です。

□ ネットワーク性能ベンチマーク

ネットワークベンチマークは、システムのネットワーク性能に対する評価基準を提供するものです。ネットワークベンチマークは主に（１）実ネットワークの性能評価、（２）ネットワーク・スタックの性能評価等に利用されます。一般にネットワークの性能は次の２点で決定されます。

- スループット：単位時間における処理量を示す指標
- 遅延：レイテンシ（待ち時間）と RTT

スループットとは単位時間における処理量を示す指標です。Ethernet のようなパケット転送型ネットワークにおけるスループットは、通信ノード間の回線のバンド幅および中間ノード内のネットワークスタックの性能を示しているといえます。また、遅延には接続時のレイテンシと、ネットワークを介した場合の RRT（Round Trip Time:往復時間）が含まれています。

バルクデータを送受信する場合には、性能はスループットだけで決定され、遅延性能はほぼ無視することができます。これは、遅延要素が大量のデータ転送にかかる時間に隠蔽されてしまうためです。一方で、telnet などを用いたインタラクティブなジョブでは遅延性能が主要な要素となります。

ネットワークスタックの性能を評価する場合には、さらに細かな分類が必要です。このため、一般に、ネットワークベンチマークプログラムでは、性能測定を行う環境や仮定される事項が多く、様々なバリエーションが存在します。

例えば、計測方法についても、

- 受動型：ネットワーク内の流量モニタによる計測
- 能動型：検査パケットを送出してネットワーク内の状況を予測

といった視点による分類や、

- 単体型：1 台だけで計測
- ペア型：測定対象のネットワークの対向にいるノードとのやりとりをもとに計測

といった分類が可能です。

TCP/UDP の性能評価 netperf ⁷⁾

netperfはネットワークに接続された2ノード間におけるTCPおよびUDPを利用した通信のスループットおよび、ネットワーク遅延を計測するベンチマークです。システムティックかつ自動的に計測するために、任意のトラフィックパターンの指定、スクリプトによる計測シナリオの記述などができます。

NFS の性能評価

クラスタシステムでは、ノード間で情報を共有するために、NFS (Network File System) を利用する機会が多いのではないのでしょうか。ファイルへの入出力が必要となるプログラムではNFSの性能が、プログラムの実行にも大きく影響してきます。そこで、ネットワークベンチマークの中にはNFSの性能評価を行うためのものも存在します。これらは主にNFSサーバのスループットとレスポンス時間性能を計測するものです。NFSのベンチマークについては、本テキストのディスクレスクラスタの章にも記述があるので、そちらを参考にしてください。

□ アプリケーション指向ベンチマーク

アプリケーション指向ベンチマークは、システムにおけるアプリケーションの実行性能に対する評価基準を提供するものです。通常は、ユーザが必要とする性能の性質を分析し、それぞれを個別に評価する必要がありますが、アプリケーション指向ベンチマークでは、実際に使用するアプリケーションを直接使用するために、システムの使用目的に即した評価を行うことができます。本節では、遺伝的アルゴリズムの汎用プログラムを利用したベンチマークを紹介します。

GA Benchmark(GAPPA bench) ⁶⁾

知的システムデザイン研究室（著者の所属する研究室）では、クラスタシステムを主に遺伝的アルゴリズム (Genetic Algorithms: GA) やシミュレーテッドアニーリング (Simulated Annealing: SA) といった進化的計算手法の数値実験のために利用しています。そこで、昨年度に開発した並列分散遺伝的アルゴリズムの汎用プログラム GAPPA を応用し、クラスタの性能評価にも利用しています。

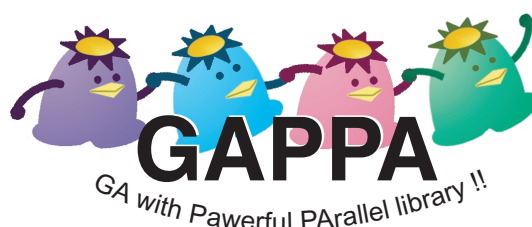


図 4.1: GAPPA (<http://is.doshisha.ac.jp/gappa/>)

遺伝的アルゴリズム (GA) は生物の進化を応用した、確率的最適化手法です。対称問題の候補解を個体として表現し、個体間の交叉や突然変異、自然淘汰などを通してよりよい解を発見するというのが GA の基本アルゴリズムです。GA に代表されるような進化的計算手法は、一般に計算負荷が高く多くの並列分散モデルが提案されています。

なかでも、我々が研究を行っている分散遺伝的アルゴリズムは、個体を複数のサブ母集団に分割して GA を実行し一定間隔おきに個体情報を交換するという並列モデルです。このアルゴリズムは通

信量、通信頻度ともに少なく、粒度の粗いモデルです。このため専用並列計算機と比較して通信性能の低い PC クラスタ上でも通信のオーバーヘッドが小さいという利点があります。GAPPA ベンチマークでは分散遺伝的アルゴリズムを用いた関数の最適化に要する時間を計測し、並列計算性能を評価します。

このように、実際に使用するアプリケーションの実行時間をもとにして計算機の性能を評価するのも有効なベンチマークの一つです。

■ ベンチマークプログラムのインストール

本節では、以下のベンチマークプログラムのインストール方法を簡単に説明します。姫野ベンチや NPB は通信ライブラリに MPI を用いていますが、MPI 自体の設定についてはすでにできているものとして、ここでは述べません³。

- 姫野ベンチマーク
- NAS Parallel Benchmark
- Netperf

□ 姫野ベンチマークのインストール

姫野ベンチマークのインストールは、本節で解説するベンチマーク中で最も簡単です。特別な設定は特に必要ありません。インストールの手順は以下の通りです。

1. ソースのダウンロード

姫野ベンチマークの Web サイト (<http://w3cic.riken.go.jp/HPC/HimenoBMT/>) から、次項環境に応じたソースをダウンロードします。通常は Fortran + MPI または C + MPI のいずれかを用いるとよいでしょう（ここでは C+MPI を使用します）。ダウンロードしたファイルをクラスタシステムの適当なディレクトリ（実行するユーザのホームディレクトリでよい）に展開します。

2. ソースのコンパイル

C + MPI バージョンのソースには Makefile とソースファイル (himenoBMT.m.c) が含まれています。必要に応じて Makefile を修正し、make を行うと bmt という実行ファイルが生成されます。

```
# make
```

3. ベンチマークの実行

実行ファイル”bmt”を MPI を使って実行します。このとき実行する CPU の数は 2^n 個 ($n=0,1,2,\dots,k$) に設定しなければならないので注意してください（姫野ベンチマの仕様）。

```
# mpirun -np 8 bmt 8CPU で実行する場合
```

³MPI については本テキストの MPI の章を参照してください

□ NAS Parallel Benchmark (NPB) のインストール

インストールに関するドキュメントは NPB を展開したディレクトリの /Doc/README.install にありますが、その概要をまとめました。

1. ソースのダウンロード

NPB のサイトから、プログラムのソースをダウンロードし⁴、展開します。すると、NPB2.3 というディレクトリが生成されます。

```
# tar xvzf NPB-2.3.tar.gz
```

2. 設定ファイルの変更

make する前に、各環境に応じて設定ファイルを変更しなくてはなりません。NPB2.3/config/ に make.def.template というファイルがあるので、これをコピーして make.def とし、MPICC や MPIF77, MPI のライブラリのパスなどを正しく設定します。make.def の一部を図 4.2 に示します。これを参考に mpicc の方も設定してください。

```
#-----
# This is the fortran compiler used for MPI programs
#-----
MPIF77 = mpif77
# This links MPI fortran programs; usually the same as ${MPIF77}
FLINK = ${MPIF77}

#-----
# These macros are passed to the linker to help link with MPI correctly
#-----
FMPI_LIB = -L/usr/local/mpich/lib -lmpich

#-----
# These macros are passed to the compiler to help find 'mpif.h'
#-----
FMPI_INC = -I/usr/local/mpich/include

#-----
# Global *compile time* flags for Fortran programs
#-----
FFLAGS = -O3
# FFLAGS = -g

#-----
# Global *link time* flags. Flags for increasing maximum executable
# size usually go here.
#-----
FLINKFLAGS =
```

図 4.2: make.def の設定 (一部)

3. make make.def を正しく変更した後、NPB2.3 ディレクトリで次のように make します。NPB では、さまざまな種類のベンチマークが可能で、それぞれについて実行するプロセス数、問題のクラスを指定することができます。それらはすべて別々の実行ファイルになるため、make の際に対象問題・プロセス数・クラスを指定して make しなくてはなりません。例えば S クラスの LU ベンチマークを 2 プロセスで実行したい場合には、以下のように make します。

```
# make LU NPROCS=2 CLASS=S
```

⁴ダウンロードの際にはユーザー登録が必要になります。

make に成功すると、NPB2.3/bin/ に lu.S.2 という実行ファイルが生成されます。うまく make できない場合には make.def の設定が誤っている可能性があるので再度確認してみてください。

4. ベンチマークの実行

lu.s.2 を実行するためには、NPB2.3/bin/ で、次のようにタイプします。

```
# mpirun -mp2 lu.S.2
```

make の際の注意

NPB の make に際しては以下の点に注意する必要があります。

1. 対象問題とプロセス数の関係

NPB には上記の 8 種類の対象問題が定義されていますが、問題の性質により実行可能なプロセス数に制限があります。

ct, ft, mg, lu, is のプロセス数は 2 の乗数 (2^* : 1, 2, 4, 8 ...)

sp, bt のプロセス数は n の 2 乗 (n^2 : 1, 4, 9, ...)

でなくてはなりません。

2. 複数の実行ファイルを一度に make する

NPB では設定ごとに実行ファイルが生成されるため、通常は何度も make の作業を繰り返さなければならず、面倒です。そこで、一度に複数のファイルを make するための方法が用意されています。まず、config/suite.def.template をコピーし、config/suite.def とします。このファイルに make したい設定の一覧を図 4.3 のように記述します。

```
# This is a sample suite file to build several executables
sp      A      16
sp      A      25
sp      A      36
ft      B       1
ft      B       2
ft      B       4
```

図 4.3: suite.def の設定 (一部)

このような、suite.def ファイルを用意して、NPB2.3 ディレクトリで

```
# make suite
```

とすると、suite.def で指定した設定の実行ファイルを一度に make することができます。

□ Netperf のインストール⁸⁾

Netperf は通信の性能評価するために測定する各ノードの通信を監視する必要があります。このため、監視用のサーバをサービスとして組み込みます。この作業には、root の権限が必要になるため管理者以外はインストールできないので注意が必要です。

1. ソースのダウンロード

まず、NetperfのソースをWebサイト（<http://www.netperf.org>）からダウンロードし、適当なディレクトリに展開します。

2. makefile をカスタマイズ

インストールする環境に応じて、makefileを変更します。ダウンロードしたままの状態では、makefileはnetperfを/opt/netperfというディレクトリにインストールしようとします。このままでよい場合には変更の必要はありませんが、変更する場合には、makefile内の以下の部分を変更します。（ここでは、/usr/local/netperfにインストールする）

```
NETPERF_HOME = /usr/local/netperf
```

次に、ccのパスを指定します。

```
CC = /usr/bin/cc
```

makefileに対する最低限の設定は以上です。このほかにも変更可能な点がありますが、ここでは触れません。

3. make する

makefileに変更を加えた後、netperfのソースがあるディレクトリで

```
# make install 
```

とタイプします。これによって、netperfのソースがコンパイルされ、実行ファイルがmakefileで指定したディレクトリ（この例では/usr/local/netperf）にインストールされます。

4. /etc/services と /etc/inetd.conf の変更

/etc/services と /etc/inetd.conf という2つのファイルにそれぞれ以下の行を追加します。この変更にはroot権限が必要になります。

- /etc/services に以下の1行を追加
netperf 12865/tcp
- /etc/inetd.conf に以下の1行を追加
netperf stream tcp nowait root /usr/local/netperf/netserver netserver

動作確認

正しくインストールされたかどうかを確認するために、netperfを実行してみましょう。まず、netserverプログラムを起動します。

```
# /usr/local/netperf/netserver
```

続いて、netperfを実行します。

```
# /usr/local/netperf/netperf
```

すると、ローカルホストに対してnetperfが実行され次のような結果が表示されるはずです。

TCP STREAM TEST to localhost				
Recv	Send	Send		
Socket	Socket	Message	Elapsed	
Size	Size	Size	Time	Throughput
bytes	bytes	bytes	secs.	10^6bits/sec
65535	65535	65535	10.00	530.89

複数ノードでの実行

複数のノードで実行するためには各ノードに、上記の手順で netperf をインストールした上で、それぞれに netserver が起動されている必要があります。他のノードに対するネットワーク性能を測定するためには次のようにします。

```
# /usr/local/netperf/netperf -H node01
```

すでに述べたように netperf では、ベンチマークのためにさまざまなオプション設定を行うことができますが、ここでは詳しく述べません。実行オプションや測定方法の詳細については netperf に付属のドキュメントを参照してください。

■ ベンチマークの測定結果と分析

本節では、これまでに紹介したベンチマークのうちいくつかを実際に PC クラスタシステム上で実行し、その結果を紹介します。ここで対象としたのは知的システムデザイン研究室の3つのクラスタシステムです。それぞれの構成は表 4.3 のようになっています。

ここに挙げた3つのクラスタはいずれも昨年度に構築されたものです。このなかで最も標準的なクラスタは ruins です。各ノードは 1CPU で、ノード間は 100BASE-T の Ethernet とスイッチングハブで接続されています。fraulein は、マスタ 1 ノードとスレーブ 8 ノードの 9 台構成で、各ノードは 2CPU が搭載された SMP (Symmetric Multiple Processor) マシンです。各ノード間は ruins と同様に、100BASE-T の Ethernet とスイッチングハブで接続されています。jupiter は、ノード間の接続に 100BASE-T でなく Myricom 社の Myrinet という GbitEthernet を使用しています。このため通信によるオーバーヘッドは 100BASE-T と比べて小さくなっています。

表 4.3: ベンチマークを行った PC クラスタシステム

	fraulein	jupiter	ruins
CPU	Pentium III 500MHz	Pentium III 600MHz	Pentium II 400MHz
SMP	○ (2 procs)	×	×
# Node	8 + 1	8 + 1	16
# PROC	18	9	16
Memory/node	128MB	128MB	128MB
OS	Linux 2.2.15	Linux 2.2.14	Linux 2.2.12
Distribution	Debian/GNU Linux	Debian/GNU Linux	Debian/GNU Linux
Compiler	gcc 2.95.2	gcc 2.95.2	gcc 2.95.2
Network	100BASE-T + Switching HUB	Myrinet	100BASE-T + Switching HUB

□ 3つのクラスタの比較

まず、姫野ベンチマークの測定結果をもとに3つのクラスタの性能を評価してみます。測定結果を図 4.4 に示します。グラフは横軸がプロセッサ数、縦軸が計算速度 (Mflops/sec) です。なお、jupiter については通常の TCP/IP と GM の2種類の通信方法で測定を行いました。ここでは詳しく述べませんが、GM を用いることで Myrinet の性能を最大限に発揮することができます。

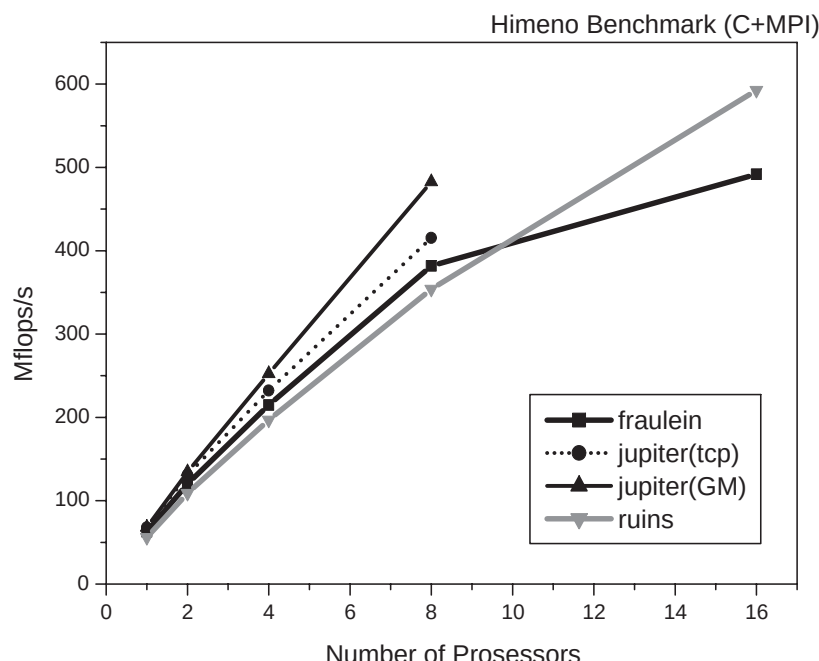


図 4.4: 姫野ベンチマークの結果 (C+MPI)

8CPU までの性能を比較すると、どのクラスタでもほぼ線形の上昇が見られます。しかし、ノード数が増えればそれだけ通信負荷も高くなるためグラフの傾きは次第に小さくなります。GM+Myrinet を用いた jupiter では通信による遅延が少ないために、より線形に近い速度向上が見られます。16CPU で fraulein の性能が落ちている点については SMP 用の最適化が成功していない可能性があります。このようにベンチマークを行うことによって、各クラスタの特徴や問題点が明らかになるのです。

□ 汎用並列計算機との速度比較

NPB の LU ベンチマーク (CLASS=A) の結果をもとに専用並列計算機との速度比較を行います。専用並列計算機における実行結果は NPB のサイトに掲載されているものを参照しました。残念ながら、NPB のサイトは最近あまり更新されておらず、最新の並列計算機での性能を知ることはできませんが、数年前の計算機との比較は可能です。ベンチマークの結果を図 4.5 に示します。グラフ中に太線で示したのが、知的システムデザイン研究室で構築したクラスタです。

汎用の並列計算機の場合は CPU の数が多いため、16CPU 程度では比較は難しいですが、それでも Intel の Paragon や、SUN の UltraEnterprise などと比較すると、我々のクラスタの方が性能が高いことが分かります。すでに述べましたが、これらのクラスタは昨年度制作したものですので、最新の CPU を使用すればさらに高い性能が期待できるのは言うまでもありません。

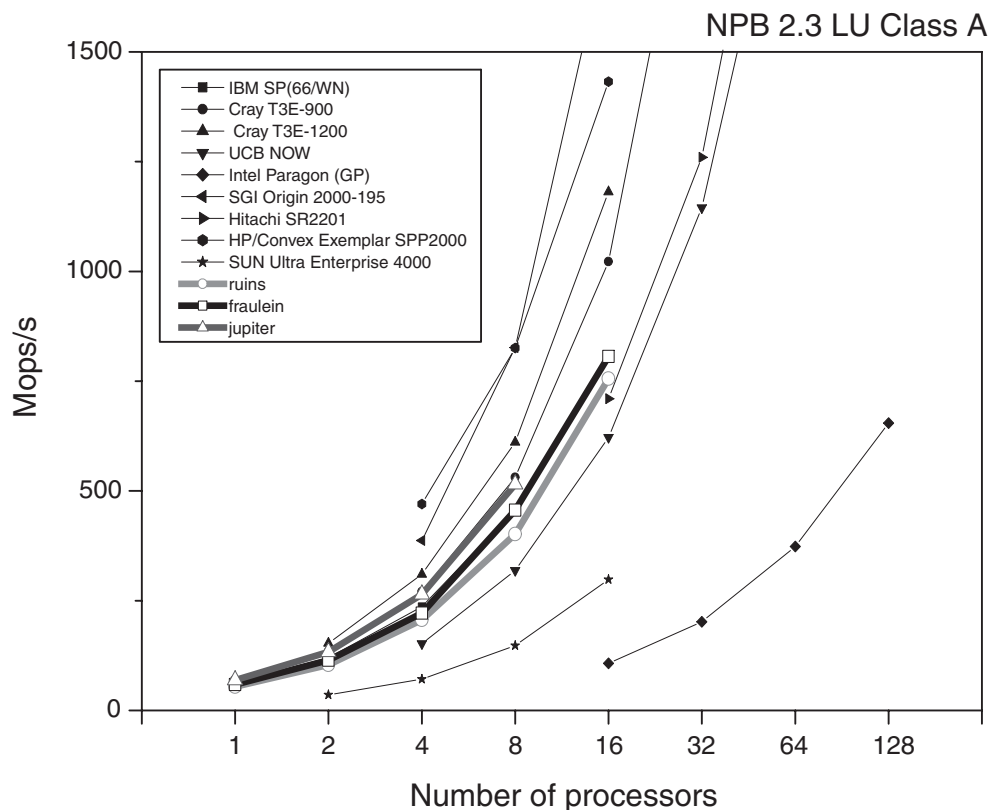


図 4.5: NPB2.3 LU ベンチマーク (CLASS=A) の結果

□ コストパフォーマンスを考えてみる

図 4.5 を見ると、計算性能ではまだまだ専用並列計算機に及ばないという感がありますが、PC クラスタの魅力は、計算性能よりもむしろそのコストパフォーマンスにあるといえます。ここでは、ベンチマークの結果からコストパフォーマンスについて考えてみます。

姫野ベンチマークの Web サイトでは、姫野ベンチマークの測定結果の一覧を参照することができます。代表的な専用並列計算機によるベンチマークの結果を表 4.4 に示します。

これらのスーパーコンピュータの価格体系は PC のそれとは異なるため、価格を知ることは難しいのですが、参考までに VPP5000(Fujitsu) のレンタル料とクラスタの価格を比較してみます (表 4.5)。

スーパーコンピュータと PC クラスタでは、アーキテクチャが全く異なるためベンチマークの結果だけで性能を比較することはできませんが、単純に計算すると、VPP500 を 3ヶ月レンタルするレンタル料で、同程度の計算性能を持つ PC クラスタを構築することができることになります。これはあくまでも例え話にすぎませんが、PC クラスタのコストパフォーマンスの高さを端的に表しているといえるのではないのでしょうか。また、PC クラスタには汎用の PC パーツを使用しているため、高性能、低価格化が一層進んでいます。ここで使用したクラスタはいずれも昨年度に制作してものであるため CPU の性能だけ見ても最新の CPU と比較するとかなり劣っています⁵。したがって、同じコストでも最新のパーツで構成した場合にはコストパフォーマンスはさらに高くなると考えられます。

⁵本稿執筆時点で入手可能な CPU のうち最速ものは、AMD の Athlon 1.1GHz または Intel PentiumIII 1GHz です。価格は Athlon 1.1GHz で 8 万円強である

表 4.4: 姫野ベンチマークの測定結果 (姫野ベンチマークのサイト より)

メーカー	機種	CPU	クロック	CPU 数	実測速度 (MFLOPS)
HITACHI	SR8000	8GFLOPS	-	16	47661.36
Compaq	AlphaServer SC	Alpha21264	667MHz	64	12383.8
FUJITSU	VPP700E	2.4GFLOPS	-	16	7986.84
HITACHI	SR8000/E1	9.6GFLOPS	-	2	7626.13
NEC	SX-5	10GFLOPS	312.5MH	1	6717.41
COMPAQ	AlphaServer SC	Alpha21264	667MHz	32	6593.6
NEC	SX-5	10GFLOPS	312.5MH	1	6481.33
FUJITSU	VPP5000	9.6GFLOPS	300MHz	1	5013.2

表 4.5: VPP5000 と PC クラスタのコストパフォーマンスの比較

	計算性能	コスト
Fujitsu VPP5000	5013.2 MFLOPS	¥4,400,000 (月額レンタル料)
PC Cluster (ruins)	592.6 MFLOPS	約¥1,000,000 (1999 年現在)

■ ベンチマークを行う上での注意

実際にベンチマークを行う上で気をつけなくてはならない注意点をいくつか挙げておきます。

● 無駄なプロセスを排除する

ベンチマークは、対象となる計算機の性能を測定するためのものであるため、その性能を正しく評価するためにはベンチマークプログラム以外のプロセスを排除する必要があります。ベンチマークを行う前には、あらかじめその旨をユーザに伝え、よけいなジョブを停止してから行うのが基本です。

● 計測環境を整える

マクロベンチマークを用いてシステム間のおおまかな比較を行う場合に、計測環境を整えることは困難ですが、マイクロベンチマークを行う場合には対象となる要素以外の要素はあらかじめ揃えておく必要があります。

例えば、

- ハードウェア間の差異を比較する際は、利用するソフトウェア環境を同一にする
- ソフトウェア間の差異を比較する際は、利用するハードウェア環境を同一にする

という具合です。

● 測定対象やベンチマークの性質を理解する

ベンチマークを行うには、まず、(1) 測定する対象自体の性質、(2) 測定に用いるベンチマークの特性を理解しなければなりません。たとえば、ソフトウェア間の差異を比較するために同じハードウェアを使っていたとしても、ソフトウェアの設定が異なっていれば、正しく性能の評価を行うことはできません。

- **安易な比較をしない**

すでに述べたように、ベンチマークの結果は対象となる計算機のある側面を評価した結果であり、それらにはそれぞれ意味があります。また、ベンチマークの結果は、その計測が行われた環境とあわせて初めて意味を持つ値となります。呈示された数値のみに惑わされて、あっちがよい、こっちがよいというような安易な比較をすべきではありません。

■ クラスタシステムにおけるジョブ管理

通常、クラスタシステムは、複数のユーザによって利用されるケースが多いと思われます。このとき、クラスタシステムを効率よく運用するためには、各ユーザが好き勝手にノードを使うのではなく、クラスタ全体でジョブのスケジューリングやリソースの管理を行う必要があります。しかし、これを手作業で行う（各ノードの利用状況を調べて、空いているノードを探してジョブを走らせる・・・）のは困難です。そこで、ここからはクラスタをうまく運用するためのジョブ管理システム（Job Management System: JMS）を紹介します。まず、はじめにクラスタにおけるジョブ管理の一般的な手法について解説し、次に代表的な JMS の一つである DQS(Distributed Queueing System) を例に JMS によるジョブの管理方法を紹介します。

□ クラスタの混み具合を覗いてみる

ジョブ管理システムの話に入る前に、便利なツールを一つ紹介します。単一 CPU のマシンの利用状況は PS コマンドなどですぐに把握することができますが、クラスタの多数のノードの状況を把握するのは大変です。

```
# rsh ps ax node01
```

リモートシェル（rsh）で、各ノードに PS コマンドを送ることも可能ですが、あまり現実的な対応策とはいえません。こんな時、すべてのノードの状況を一覧表示してくれるようなプログラムがあれば便利です。

便利な Beowulf Watch

Beowulf Watch（bWatch）はクラスタの各ノードのメモリやプロセスの状態を表示するための簡単な Tcl/tk スクリプトです。このプログラムは、rsh を利用しており、すべてのユーザーが簡単にクラスタの利用状況を把握することができます。インストールや設定は簡単なものに対し、非常に有効なソフトウェアです。

Host Name	Num Users	Time	1 min Load	5 min Load	15 min Load	Total Mem	Free Mem	Shared Mem	Buffers	Cache	Total Swap	Free Swap
node1	4 users	4:17	0.00	0.07	0.15	377 Mb	6476 Kb	53 Mb	7 Mb	337 Mb	762 Mb	762 Mb
node3	0 users	4:17	4.15	3.85	2.47	61 Mb	38288 Kb	11 Mb	0 Mb	19 Mb	0 Mb	0 Mb
node4	0 users	4:17	1.08	0.94	0.56	61 Mb	40376 Kb	8 Mb	0 Mb	18 Mb	0 Mb	0 Mb
node5	0 users	4:17	1.02	0.98	0.67	61 Mb	32880 Kb	9 Mb	0 Mb	20 Mb	0 Mb	0 Mb
node6	0 users	4:17	1.08	0.92	0.52	61 Mb	40544 Kb	7 Mb	0 Mb	18 Mb	0 Mb	0 Mb
node7	0 users	4:17	0.99	0.89	0.51	61 Mb	40552 Kb	7 Mb	0 Mb	18 Mb	0 Mb	0 Mb
node8	0 users	4:18	0.99	0.89	0.51	61 Mb	41532 Kb	7 Mb	0 Mb	17 Mb	0 Mb	0 Mb
node9	0 users	4:18	3.15	2.75	1.60	61 Mb	41188 Kb	7 Mb	0 Mb	17 Mb	0 Mb	0 Mb
node10	0 users	4:18	2.07	1.82	1.06	61 Mb	41552 Kb	7 Mb	0 Mb	17 Mb	0 Mb	0 Mb
node11	0 users	4:18	0.99	0.90	0.51	62 Mb	27040 Kb	8 Mb	0 Mb	32 Mb	0 Mb	0 Mb
node12	0 users	4:18	3.07	2.74	1.63	62 Mb	44032 Kb	6 Mb	0 Mb	16 Mb	0 Mb	0 Mb
node13	0 users	4:19	14.27	11.64	6.22	62 Mb	1636 Kb	3 Mb	48 Mb	1 Mb	0 Mb	0 Mb

図 4.6: Beowulf Watch (<http://www.sci.usq.edu.au/staff/jacek/bWatch/>)

インストール手順は以下の通りです.. BWatch の実行には、Tcl/tk⁶の実行環境が必要です。

1. ソースのダウンロード

BWatch の Web サイト (<http://www.sci.usq.edu.au/staff/jacek/bWatch/>) からソースをダウンロードし、適当なディレクトリに展開する

⁶tcl/tk は John Ousterhout が設計、実装した言語とツールキット。Tcl / Tk の Tcl とは『スクリプト言語』で、Tk というのは『ツールキット』、つまり、GUI を構成するための部品の集まりです。

2. make

以下のようにタイプして、make する

```
# make bwatch
```

3. 動作確認

bWatch を起動するためには次のようにタイプします。

```
# ./bWatch.tcl
```

4. /usr/local/bin へのインストール

```
# make install
```

とすれば、/usr/local/bin に bWatch をインストールすることもできます。インストールディレクトリを変更したい場合には Makefile に変更を加えます。

□ ジョブ管理システムとは

bWatch のような便利なアプリケーションは存在するものの、自分がジョブを実行するたびに、クラスタの利用状況を調べて、他人のジョブが終わるのを見計らって実行するのは難しい。そこで、ユーザの代わりにジョブのスケジューリングやリソースの管理を行ってくれるのが、ジョブ管理システムです。

一般にジョブ管理システムにはどのような機能が求められるのでしょうか？Saphir らは NASA の The Numerical Aerodynamic Simulation (NAS) parallel system やクラスタについて分析し、ジョブ管理システムの役割には次の 3 つの側面があると思いました。

● ユーザーサーバ (userserver)

ユーザはキューにジョブを追加したり、ジョブに対するリソースを要求したり、キューからジョブを削除したり、ジョブやキューの状態を尋ねたりします。ジョブ管理システムはこのようなユーザの要求の窓口にならなくてはなりません。

● ジョブスケジューラ (job scheduler)

ジョブ管理システムは、ジョブの種類、リソースの要求、リソースの利用可能性、スケジューリングのポリシーなどを考慮した上で、ジョブのスケジューリングを行わなくてはなりません。

● リソース管理 (resource manager)

ジョブ管理システムは、リソースを監視し、スケジュール通りにリソースを割り振らなくてはなりません。

JMS を利用することで、ユーザはクラスタの利用状況を把握する必要がなくなります。ユーザは JMS にジョブを渡すだけでよいのです。ユーザの渡したジョブは JMS が適切なノードに割り振り、キューに蓄積されます。そして、必要なリソースが解放されるのを待って実行されます。JMS によるジョブの管理の概念図を図 4.7 に示します。

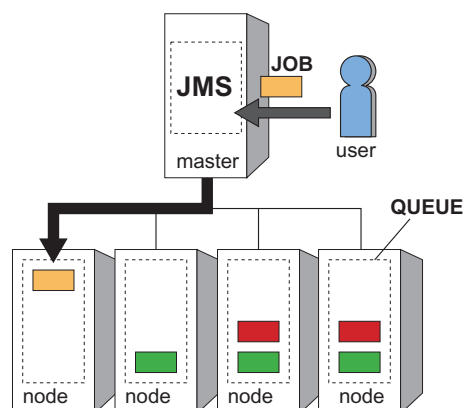


図 4.7: JMS によるジョブの管理と実行

■ JMS によるジョブの管理と実行

JMS は、クラスタの全てのノードのジョブを管理することから、分散システムとして認識されることがありますが、図 4.7 にも示したように実際にはマスターノードで一元的に管理しており、きわめて中央集権的です。したがって、全ての設定やログファイルなどは 1 か所で管理されます。JMS を使うと、PVM や MPI を用いた並列プログラミングを行わなくても、ジョブを適当に分散してくれます。

JMS は、実行中のジョブへの影響を最小限に押さえて、ジョブの設定を更新する必要があります。そのため、ジョブを実行する前にスクリプトが起動し、セキュリティのチェックや、クリーンアップを行います。また、ユーザは、いつでも自分のジョブを停止または中止することができます。なお、プロセスを停止させたり、中止した際にはクラスタのリソースを解放することが重要となります。

クラスタにおけるジョブ管理には様々な側面があります。主なものを表 4.6 にまとめました。次節からは、これらについて順に解説します。

表 4.6: ジョブ管理における問題点とその解決策

問題点	対応策	解決すべき問題
ジョブのプライオリティ	静的な処理 動的な処理	優先度の高いジョブの実行の遅れ 通信のオーバーヘッド。実装が困難
リソース要求	静的な処理 動的な処理	ロードバランスが悪い 通信のオーバーヘッド。実装が困難
リソースの共有	Dedicated Space Sharing Time Sharing	単純だが、効率が悪い タイリング問題、 large job問題 プロセスベースのジョブコントロールが必要
スケジューリング	Independent Gang	単純だが、効率が悪い 実装が困難
ローカルジョブとの競合	stay migration	ローカルジョブの実行の遅れ 移住先の選択、移住時の通信オーバーヘッド

□ ジョブの種類

クラスタ上で、実行されるジョブはノードの利用方法やジョブの性質から、いくつかに分類することができます。まず、必要とするノード数で2つに分類が可能です。1つのノードで実行されるシングルジョブと、複数のノードで実行されるパラレルジョブです。次に、ジョブの性質からインタラクティブジョブとバッチジョブの2つに分類することができます。

● インタラクティブジョブ (interactive job)

ターミナルへの出力や、ユーザによる入力などが必要となるジョブ。素早いレスポンスが要求されますが、一般に計算機のリソース要求は小さい。

● バッチジョブ (batch job)

バッチジョブは大規模なメモリ空間や、長いCPU タイムなど巨大なリソースを要求します。しかし、素早いレスポンスはあまり必要ではなく、ジョブはキューに投入され、リソースが空くの待ってから実行されます。

インタラクティブジョブとバッチジョブはJMSによって同時に管理することが可能です。

また、JMSの管理外で実行されるジョブも存在します。これはforeign jobと呼ばれます。これらは主に、クラスタの各ノードで独自に実行されるようなジョブです。これらはクラスタとしてのジョブ(クラスタジョブ)に対してローカルジョブと呼ばれます。

□ ジョブスケジューリング

ジョブのスケジューリング問題はプロセッサのスケジューリングに非常によく似た問題で、大変難しい問題です。スケジューリングには、時間をもとにスケジューリングを行うカレンダースケジューリング (calendar scheduling) とイベントの発生によってスケジューリングを行うイベントスケジューリング (event scheduling) があります。スケジューリングに関する問題には様々なフェイズが存在します。ジョブのスケジューリングは一般に、なんらかのプライオリティ (順番) に応じて行われます。プライオリティは、キューへ投入された時間や、リソースの要求、ジョブの種類、ユーザの権限などを考慮して決定されます。

プライオリティの決定は、静的または動的に行われます。それぞれ、つぎのような特徴を持ちます。

● 静的プライオリティ (static priority)

静的プライオリティを採用した場合は、あらかじめ定めたとおりにジョブを実行します。キューに登録された順に実行していく非常にシンプルな方法です。

● 動的プライオリティ (dynamic priority)

動的にプライオリティを決定する場合、ジョブの実行は、他のジョブの進行や、新しいジョブの登録状況などに応じて変化します。また、昼間はインタラクティブジョブを優先し、夜間はバッチジョブを優先するといったポリシーを適用することもできます。動的プライオリティでは後述のタイリング (tiling) 技術が重要であり、これによってシステムの性能は大きく左右されます。

□ リソース管理

リソースの要求は、動的にも静的にも処理することができます。リソースの管理は、クラスタ全体のパフォーマンスを考える上できわめて重要です。並列のジョブを実行する場合にも、全てのノードが常に利用されているわけではありません。このようなリソースについても有効に活用できることが望ましいといえます。

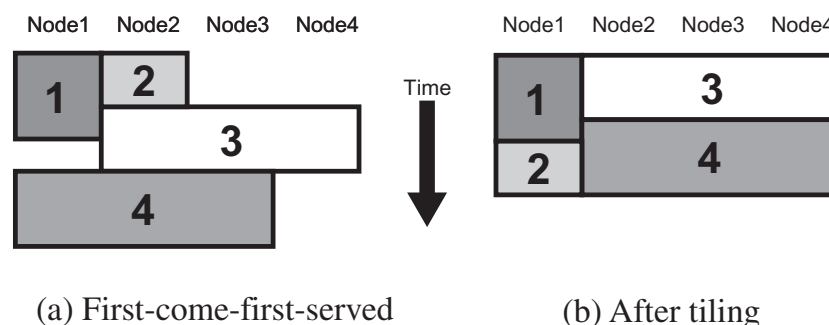


図 4.8: タイリングによる処理効率の向上

静的リソース管理は、現在のクラスタの多くで利用されています。あらかじめ決定されたリソースをジョブが終了するまで固定し、ジョブを実行します。しかしながら、この方法では、リソースをうまく使うことは困難であり、ユーザ自身がロードバランスを考慮する必要があります。また、この方法ではノードのトラブルなどに対応できないという問題もあります。

動的リソース管理では、ジョブの実行中に使用しないノードを解放することができます。ただし、これを実現するためにはジョブと JMS が密接に連携して動作する必要があり、実装は困難です。ジョブは JMS に対して非同期にリソースの解放や追加の要求を渡すため、JMS は常に全ノードのリソースを把握した上で、最適化を行わなくてはなりません。

JMS とジョブとの高度な連携のためには、プログラミング言語やライブラリを改良しなければなりません。並列化ライブラリである PVM や MPI には、これに必要となる機能が含まれています。

□ クラスタのノード 共有

クラスタで、複数のジョブを効率的に事項するためには、ノードのシェアリングが重要となります。ノードのシェアリングのためには複数の方法があります。

Dedicated mode

Dedicated mode では、クラスタで同時に実行できるジョブは 1 つだけに限定します。また、各ノードには 1 つのプロセスが割り当てられます。最も単純な手法でありますが、ジョブが完了し全てのリソースを完了するまでは次のジョブを実行できません。

Space sharing mode

Space sharing mode では、複数のジョブを複数のグループに分けて同時に実行することができます。一般に 1 ノードには 1 つのプロセスが割り当てられます。このときタイリング技術が重要になります。タイリングによる処理速度の向上のモデルを図 4.8 に示します。

クラスタジョブのような大きなジョブを、ノードでの小さなジョブが妨げて実行できないという large-job 問題といわれる問題があります。この問題は前述の 2 つのノードシェアリング法では解決できません。そこで、これを解決するためには次に述べる time sharing が必要です。

Time sharing

前述の 2 つの方法では、基本的に 1 ノードには 1 プロセスしか割り当てられません。一方、time sharing mode では複数ユーザのプロセスを 1 つのノードに割り当てることが可能になります。Time

sharing については複数のポリシーが存在します。

- **Independent scheduling**

もっとも単純な Time sharing 法としては各ノードの OS の機能を利用してスケジューリングを行う方法があります。これは、ローカルスケジューリングまたは独立スケジューリングと呼ばれます。ただし、この方法では並列のジョブに対するパフォーマンスがきわめて悪くなります。なぜなら、パラレルジョブの実行では、ノード間の協調が非常に重要となるためです。

- **Gang Scheduling**

Gang Scheduling ではパラレルジョブの実行のために、全てのノードのプロセスに対して同時にスケジューリングを行います。つまり、パラレルジョブの一つのプロセスがアクティブであれば、すべてのプロセスはアクティブです。この方法は、ホモなクラスタにおいてはパフォーマンスを向上させるといわれています。この方法の問題は、「同時に」という点にあります。クラスタにおいては全てのノードで完全に同期がとれているわけではなく、システムクロックにずれがあることもあります。Gang Scheduling を行った場合このシステムクロックのずれがクラスタ全体のパフォーマンスを低下させることになります。したがって、非同期なクラスタや、ワークステーションをつないだだけのヘテロなクラスタには向かないといえます。

ローカルジョブとの競合

クラスタのノードでローカルジョブが実行されている場合、スケジューリングはより複雑になります。このときローカルジョブは、クラスタジョブよりも優先度が高いとすると、このときにとるべき方法は以下の2つです。

- **stay scheme**：クラスタジョブが、ローカルジョブが終了するまで待つ

この方法では移住のコストはありませんが、ローカルジョブ、クラスタジョブともにパフォーマンスが低下するという問題があります。

- **migrate scheme**：クラスタジョブを他のノードに移す

同期待ちなどで、システム全体のパフォーマンスが低下します。また、移住先の状況を把握しておく必要があります。

■ ジョブマネージメントシステムの紹介

現在代表的な JMS として市販のもの、フリーのものを併せて 20 種類以上が存在します。Baker らは、代表的な JMS を調査し、その性能を分析しました。

JMS を導入する際の注意点としては、MPI のサポートを考慮する必要があります。またジョブのスケジューリング、単一ジョブ実行時間の上限、登録ジョブ数の上限、Aterm (busy) によってキューを一時的に利用負荷にするロードの程度などをどの程度まで設定できるのかなどについても考慮しなくてはなりません。以下に、代表的な JMS とその簡単な特徴を示します。

- **DQS (Distributed Queuing System)**

Florida State University によって開発されたフリーの JMS。MPI はサポートされていますが、HPF には対応していません。商用バージョンは Codine と呼ばれ、ドイツの GENIAS GmbH が提供しています。ちなみに、ここで紹介するものの中では唯一、Debian GNU/Linux 用のパッケージが公式に用意されています。

<http://www.scri.fsu.edu/pasko/dqs.html> (DQS)

<http://www.scri.fsu.edu/pasko/dqs.html> (Codine)

- **LSF (Load Sharing Facility)**

カナダの Platform Computing Corporation によって提供されている商用パッケージであり、世界中で 20000 ライセンス以上が購入されています。"Scalable Parallel Computing¹⁾"では主にこの JMS が紹介されています。

<http://www.platform.com/>

- **Condor**

University of Wisconsin によって開発されたフリーのパッケージです。信頼性のあるチェックポイントシステムやプロセス移住をサポートした最初のシステムです。リモートからの I/O にも対応しています。ただしオーバーヘッドが大きいために、動作は遅いといわれています。

<http://www.cs.wisc.edu/condor/>

- **NQS / Generic NQS / The Connect:Queue**

NQS は、JMS のデファクトスタンダードとも言うべき存在です。このため、多くの JMS が NQS との互換性 (NQS のコマンドのサポート) を謳っています。Stirling Software によって 1980 年代初期に開発された NQS はさらに The Connect:Queue と呼ばれる商用パッケージに発展しました。これをフリーなパッケージとして実装されたのが Generic NQS です。これはイギリスの University of Sheffield によって運営されています。

- **PBS (Portable Batch System)**

NASA によって開発、配布されています。BSD licence のもとで自由に利用できます。

<http://pbs.mrj.com>

■ Distributed Queuing System (DQS)

前節で紹介した JMS のうち、ここでは Florida State University によって開発されたフリーの JMS である DQS を取り上げて、一般的なジョブ管理システムの構築と運用について簡単に説明します。ここで DQS を取り上げるのは以下のような理由からです。

1. フリーウェアである
2. 設定項目が少なくとっつきやすい

DQS は、他の JMS と比べて設定すべき項目が少なく、とりあえず JMS を試してみるという点では非常に有効だと思われます。

3. Debian 用のバイナリファイルが提供されている

ここで紹介した JMS のなかでは唯一 Debian/GNU Linux 用のバイナリファイル（deb ファイル）が提供されています。Debian ベースのクラスタでは、非常に簡単にインストールできます。次節以降では Debian でのインストール方法を解説します。

□ DQS の設定

JMS によるジョブの管理は、マスターノードにおいて集中的に行います。そのため、マスターノードとスレーブノードではインストール時の設定が異なります。本節ではそれぞれについて順に説明します。

マスターノードへのインストール

マスターノードには、管理するスレーブノードを `qconf -ah` コマンドで登録する必要があります。`qconf` はキュー（queue）の設定を行うコマンドであり、オプションの `-ah` は Add Hosts の意です。このときマスターノードは自動的にリストに加えられているので、追加する必要はありません。登録されているノードは `qconf -sh` コマンド（-Show Hosts）で確認することができます。

スレーブノードへのインストール

`/etc/dqs/` にある 2 つの設定ファイル `conf_file` と `resolve_file` を変更します。これらのファイルで、このノードが所属するグループ（DQS ではセルと呼ぶ）と、マスターノードを設定します。この 2 つのファイルは、NFS などでも共有するのが理想ですが、不可能である場合も、全てのノードで同一にしなければなりません。

また、各ノードにジョブを実行するためのキューを追加する必要があります。`qconf -aq` コマンドで、キューの名称とホスト名を登録します。追加したキューは、`qmod -e` コマンドでモードを変更し利用可能な状態にする必要があります。

設定の確認

```
# qstat -f
```

キューの状態を表示する `qstat` コマンドに、`-f` オプションをつけて実行すると、全てのキューの状態を確認することができます。また、`-ext` オプションをつけると全てのキューの設定を確認することができます。

```
# qstat -f
```

Queue Name	Queue Type	Quan	Load	er	State
node01	batch	0/1	0.00	er	UP
node02	batch	0/1	0.00	er	UP
node03	batch	0/1	0.00	er	UP
node04	batch	0/1	0.00	er	UP

図 4.9: qstat -f コマンドでキューの状態を確認

qstat -f コマンドを実行すると図 4.9 のように、各キューの状態が表示されます。Queue Name は各ノードに設定したキューの名前に対応してる。Load とは各マシンの負荷を表す。State は状態を表す。UP がジョブを投げれる状態、Alarm は負荷が高いために出る警告でジョブが投げれない状態、DISABLE は管理者の設定によりジョブが投げれない状態です。

□ DQS によるジョブの実行

DQS を利用するためには、直接プログラムを実行するのではなく、実行のためのスクリプトが必要となります。このスクリプトには、実行プログラム名だけでなく、実行時のリソース要求などもコメントとして記入しておきます。このスクリプトを DQS のマスタに渡すことで、適切なノードにジョブを分配してくれます。

□ 実行スクリプトの作成

ここでは、動作確認のため以下のようなスクリプトを作成し、DQS から実行します。

```
#!/bin/sh
#$ -l qty.eq.1  ←リソースの要求
#$ -N DQSTEST  ←ジョブの名前
#$ -cwd
hostname        ←ここから実行するジョブの内容
sleep10
```

図 4.10: DQS 用実行スクリプト (dqs.sh)

DQS の実行スクリプト内には qsub のオプションを記入します。dqs.sh のなかで #\$ 以下に記入したものがそれにあたる。オプションはそれぞれ次のような意味を持ちます。

l オプション (リソースリスト)

ジョブの実行に必要なリソースを要求します。通常は使用するノード数を指定するのに用います。

N オプション (ジョブの名前)

ジョブに名前を付けることができます。

cwd オプション (Current Working Directory)

スクリプトの実行ディレクトリをカレントワーキングディレクトリとします。このオプションをつけることで、スクリプトファイルからの相対パスでの記述が可能になります。

なお、MPI で並列のジョブを実行するには次のようなオプションを与える必要があります。

```
# qstat -f
```

Queue Name	Queue Type	Quan	Load	State
node01	batch	0/1	0.00 er	UP
node02	batch	0/1	0.00 er	UP
node03	batch	0/1	0.00 er	UP
node04	batch	1/1	0.01 er	UP
username DQSTEST	99	0:5	r	RUNNING 05/07/00 21:08:12

図 4.12: ジョブの実行状況の確認

```
#!/bin/sh
#$ -l qty.eq.8
#$ -N luA8
#$ -A dummy_account
#$ -cwd
mpirun.ch_gm -np 8 lu.A.8
```

図 4.11: DQS 用実行スクリプト (MPI 版)

□ ジョブの登録

```
# qsub dqs.sh
```

```
your job 99 has been submitted
```

DQS にジョブを登録するためには、`qsub` コマンドを利用します。上の例は先ほど作成した `dqs.sh` を追加するためのものです。上の例の 99 は DQS が定めたジョブの ID です。

□ ジョブの実行状況の確認

ジョブが登録されると、DQS は負荷の低いマシンでそのジョブを実行します。`qstat -f` コマンドでキューの状態を確認すると先ほど登録したジョブが実行されているの確認することができます。ジョブが実行されると図 4.12 のように表示されます。

全てのキューが使用中の場合

全てのキューが使用中の場合、または必要なリソースが確保できない場合には `QUEUED` と表示され、キューに追加されます。他のジョブが終了しリソースが解放されると、DQS がそこにジョブを流し、実行します。

□ キューからのジョブの削除

```
# qdel ** (**はジョブの ID)
```

```
username has deleted the job "***"
```

実行中のジョブを途中で削除する場合には、`qdel` コマンドを使用します。削除したいジョブの ID を `qstat -f` で確認し、`qdel` コマンドの引数として指定します。

□ Qmonitor(qmon)

DQS には、X-Windows 用の GUI インターフェイスである `qmon` も用意されています。`qmon` を起動すると、図 4.13 のようなウィンドウが表示されます。ここから、さまざまなコマンドを実行することができます。また、各キューの状態やジョブの実行状況を視覚的に把握することができます。`qmonitor` は `qmon` コマンドで起動します。

```
# qmon
```

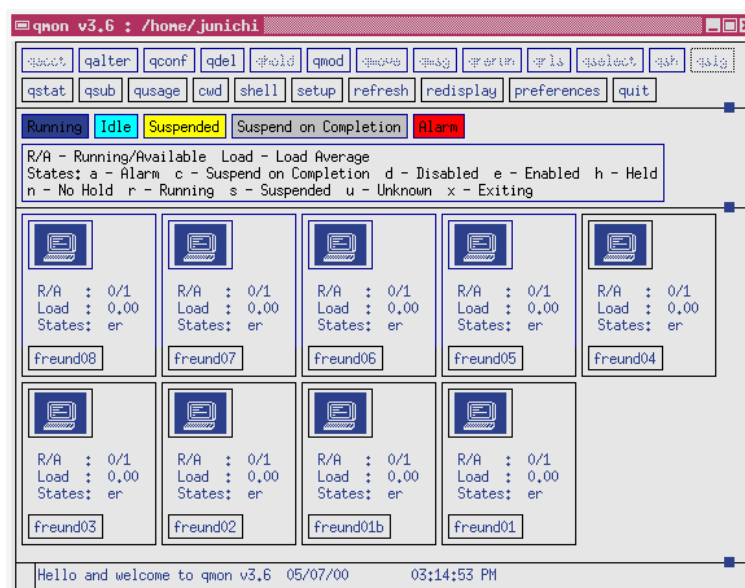


図 4.13: Qmon の GUI

■ おわりに

本章では、PC クラスタにおけるベンチマークとジョブ管理について解説を行いました。本テキストを手にとっておられる方は、自分でクラスタを構築してみたいと考えている方ばかりだと思います。実際にクラスタシステムを構築されましたら、まず最初にベンチマークを行ってみてください。複数のノードによる計算性能の向上を数値で体感できると思います。また、万一設定に問題があれば、ベンチマークの結果がそれを教えてくれると思います。

実際にクラスタが完成して、運用をはじめるとジョブ管理の問題も出てくると思います。そのときにはぜひ本章を参考にして、JMS の導入も検討してみてください。クラスタをさらに有効に活用できるのではないかと思います。

しかしながら、ベンチマークについても JMS についても、ここでは簡単な解説にとどめています。ここでは述べなかった Tips やインストールなどのより詳細なドキュメントを随時 Web サイト上で公開していく予定です。原稿の執筆に際しては誤りのないように注意を払いましたが、誤りなどがあればぜひ指摘していただきたいと思います。それらについても、随時 Web サイト上で公開していきたいと考えています。

なお、本稿に関するドキュメントは下記の URL で配布予定です。

PC クラスタ超入門：<http://mikilab.doshisha.ac.jp/dia/SMPP/>

(2000 年 8 月 30 日版)

■ 参考文献

- 1) Kai Hwang and Zhiwei Xu, 『Scalable Parallel Computing』, WCB/McGraw-Hill (1998)
- 2) TOP500 Super Computing Sites, <http://www.top500.org/>
- 3) NAS Parallel Benchmarks Home Page, <http://www.nas.nasa.gov/Software/NPB/>
- 4) ScaLAPACK Project, <http://www.netlib.org/scalapack/>
- 5) 姫野ベンチマーク, <http://w3cic.riken.go.jp/HPC/HimenoBMT/>
- 6) GAPP: parallel distributed genetic algorithm library,
<http://is.doshisha.ac.jp/gappa/index.html>
- 7) The Public Netperf Homepage, <http://www.netperf.org>
- 8) Information Networks Division, Hewlet-Packard Company, 『Netperf: A Network Performance Benchmark』, <http://www.netperf.org> ,(1995)
- 9) Florida State Univ. Supercomputer computation research institute, 『DQS3.2.X USER GUIDE』,(1998)
- 10) Florida State Univ. Supercomputer computation research institute, 『DQS3.2.X Insstration and maintenance manual』,(1998)
- 11) Florida State Univ. Supercomputer computation research institute, 『DQS3.2.X REFERENCE MANUAL』,(1998)

ALINKA LCMを用いたPCクラスタ システムの構築

ビジュアルテクノロジー (株)/青山学院大学理工学部総合研究所

中田 寿穂

nakata@v-t.co.jp

■ はじめに

一昨年あたりから、大学の研究室レベルで PC クラスタの導入事例が多く見られるようになった。かくゆう筆者も青山学院大学で、Dual Pentium II 69 台の PC クラスタを 1999 年に設計・製作したのだが、いざ構築しようとなるとなかなか有用な資料が存在しなかった。Linux に関しては SLS¹ のころから使っていたのでそれなりの知識はもっていたが、元来実験物理屋である筆者は² 並列計算に関してはズブの素人であった。Web サーフィンしながらやっとなおもいで必要な資料を捜し当て PC クラスタを構築したというのが実状であった。現状、クラスター専用のディストリビューションが存在しないため、一般的には各ノードにそれぞれ OS をインストールし、rsh を使用できる環境を構築することにより、とりあえずは MPI ベースの並列プログラムの実行環境は構築できる。しかしながら、より便利な環境を構築しようとなると、/home の NFS による共有、NIS によるユーザ管理、PBS などのバッチ処理システムと結構いろいろと設定しなければならない。数台ならまだしも、これが数十台もの規模となるとインストール、設定だけでも一仕事である。多くの人は、クラスタ環境を構築することが目的ではないのだから、これらの仕事に関してはなるべく簡便に構築でき、日々の運用・管理に関してもなるべく手を抜きたいのが本性であろう。そんな要望に答えるために作成したのが **Debian GNU/Linux VT-PC Cluster Selection** です。本テキストでは、筆者が現在参加している Aoyama Plus Project に関しての紹介と **Debian GNU/Linux VT-PC Cluster Selection** の中核をなすクラスタ管理システム **ALINKA LCM** に関して解説します。

■ Aoyama Plus Project

Aoyama Plus とは **Aoyama Gakuin University PC Cluster System**³ の略称で 1998 年度よりはじめたプロジェクトです。開設当初のプロジェクトメンバーは青山学院大学理工学部物理学科の久保 健教授、羽田野 直道 助教授、古川 信夫 助教授、中田 寿穂 助手の 4 名です。主に物性物理学、数理物理学、統計力学の理論計算に並列計算機を使用しています。プロジェクトの経緯に関して <http://www.phys.aoyama.ac.jp/~furukawa/Aplus/history.html> を参照してください。

¹Linux ディストリビューションの 1 つ

²実は、高温超伝導の新物質開発、結晶構造解析が本業であったりする。

³<http://www.phys.aoyama.ac.jp/aoyama+/>

現在所有する並列計算機システムは

- ARK (Dual Pentium II 350MHz, 69nodes)
- Bern (VT-Alpha5, 10nodes)
- xian (Celeron 300A, 4nodes)
- clump (Dual Celeron 533MHz, 11nodes)
- d???? (Dual Pentium III 600MHz, 6nodes)
- eden (Celeron 500MHz, 9nodes)

の 6 機種です.

□ ARK (Dual Pentium II 350MHz, 69nodes)

ARK は Dual Pentium II (350MHz) 69nodes の PC クラスタシステムであり, 1999 年 7 月に稼働しはじめました. この並列計算機システムは 1 台のゲートウェイ/ファイルサーバと 4 台のブートサーバ, 64 台のディスクレス計算ノードで構成されており, 総メモリ量は約 18GB となっています. 各 PC は 100base-TX のネットワークで接続されています. 図 5.2~5.7 に NAS の Parallel ベンチマークの結果を示します.



図 5.1: ARK の概観

□ bern (VT-Alpha 5 533MHz/600MHz, 10nodes)

VT-Alpha 5 533MHz(Redhat Linux) 8 台, 600MHz(Degial UNIX) 2 台のヘテロタイプのクラスタシステムで 1998 年に導入. 購入当時は, Alpha 版の gcc, g77 のコンパイラ性能が良くなく, Degital UNIX 上で Dec FORTRAN, C を用いてコンパイルしたものを Linux 上で実行していた. 今年度中に Kondra(or Redhat, SuSE) + SCore のシステムにバージョンアップする予定です.

□ clump (Dual Celeon 533MHz, 11nodes)

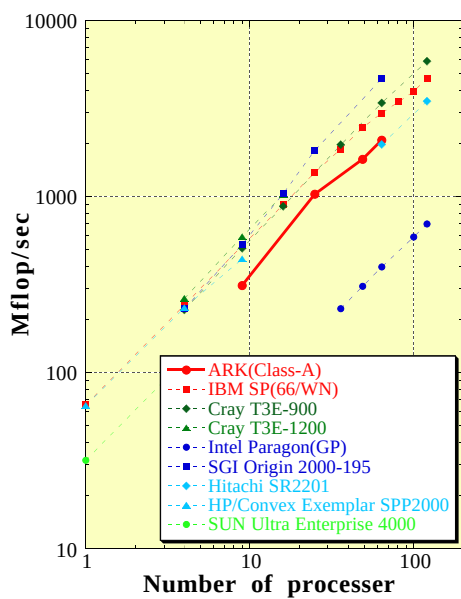


図 5.2: Total Machine Performance of BT Class A

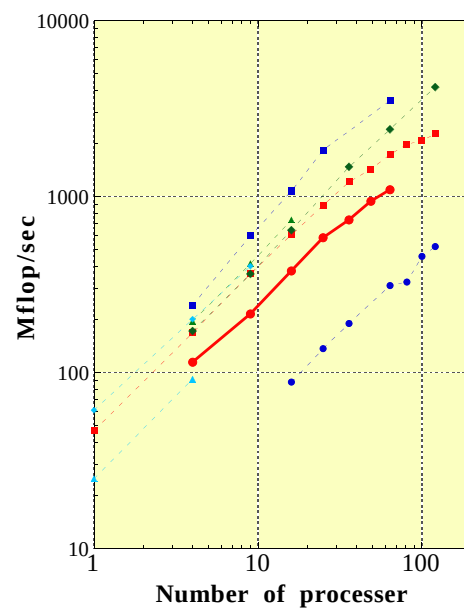


図 5.3: Total Machine Performance of SP Class A

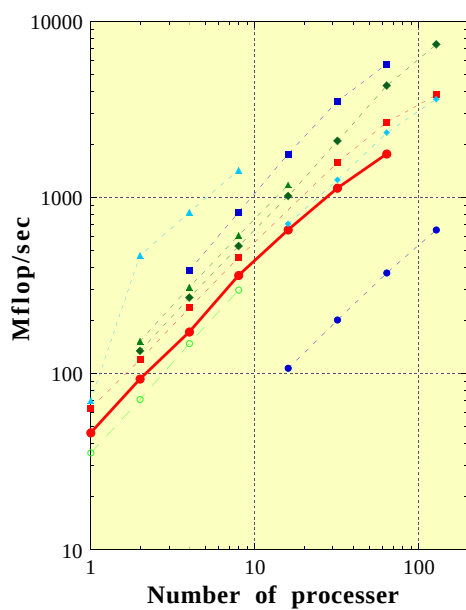


図 5.4: Total Machine Performance of LU Class A

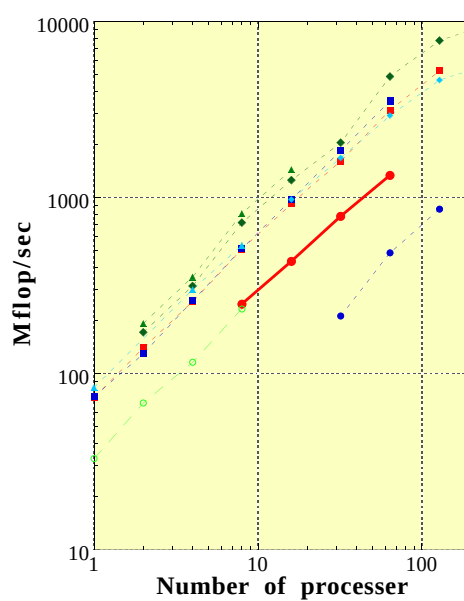


図 5.5: Total Machine Performance of MG Class A

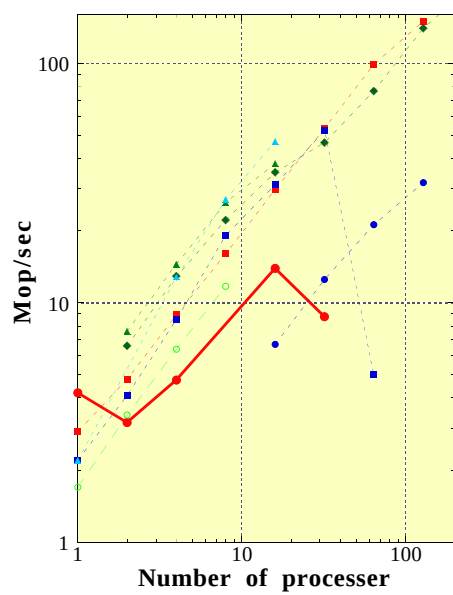


図 5.6: Total Machine Performance of IS Class A

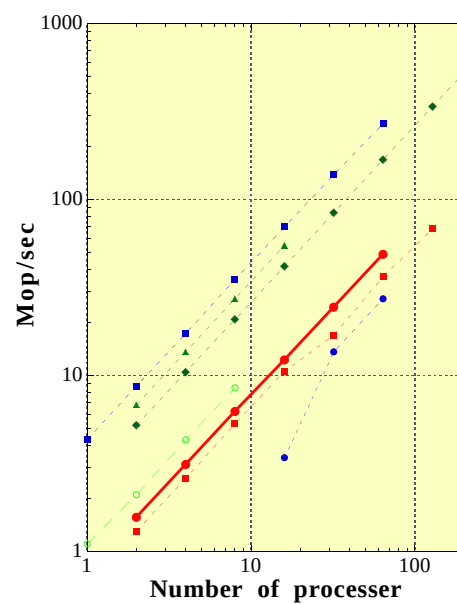


図 5.7: Total Machine Performance of EP Class A



図 5.8: bern の概観

ARK を構築してわかったことだが, PC も部材さえちゃんと選べば長期安定稼働する. ARK も 1999 年 7 月に起動しはじめ, 大きなトラブルもなく現在も順調に働いている.

ただ 1 つ困ったことがある. 大学というところは埃が多く, 1 年もすると PC のケースの中が埃だらけになってしまう. ショートの防止のためにも, 1 年に 1, 2 回はケースの中を掃除したいのだが, 1 台 2 台ならともかく, 69 台ともなると大仕事である. また, CPU ファンなどの故障もケースを開けてチェックしなければならない. 市販の ATX ケースは非常にやすく購入できるが, やはりスペース的な問題がでてくる. これらの問題を留意して作成したのが **clump** である. clump はドア付きの 19 インチラックに, 引出し式の棚板を付け, その棚板の上に M/B, FDD 等を装着しおり, 非常にコンパクトなシステムとなっている. また棚板自体, ATX の M/B が載せられるように設計しているため, 今後のでてくるであろう, IA-64 のシステムでも ATX 規格の M/B であれば, 置き換えることが可能である. この棚板のシステムに関しては, 同等のものがビジュアルテクノロジー (株) から販売されています.



図 5.9: clump 概観

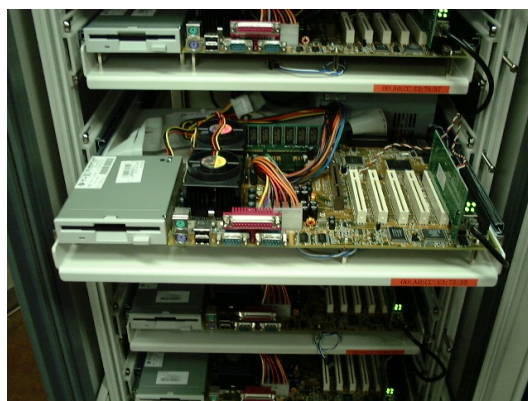


図 5.10: オープン棚板システム

☐ d?????? (Dual Pentium III 600MHz, 6nodes)

Turbo Linux + SCore のシステムです. 次期 SCore で採用される, マルチ NIC システムの評価用クラスタシステムです.

☐ eden (Celeron 500MHz, 8nodes)

今年度, 研究室に入ってきた 4 年生に, 練習問題で作成させた PC クラスタシステムです. NLX 規格のベアボーンキットを使用しており, 総予算 36 万円の超格安クラスタシステムです. ちなみに eden の e は “economical” の e でもあります.

■ ALINKA LCM と Debian VT-PC Cluster Selection

□ ALINKA LCM

我々がおもに行っている数値シミュレーションでは、各マシンの計算時間に対する通信時間はほとんど無視できるほどの通信 (通信時間は計算時間に対して 1%以下) 量でしかない。またテンポラリーにローカルディレクトリに書き出すファイルもないため、ARK のような、100base-TX のネットワークで結ばれたディスクレスクラスタシステムが最も適していた。HDD などの駆動部のないディスクレスクラスタシステムは一度構築してしまえば、運用管理は非常に楽であるが、その一方で構築手順は、各マシンに搭載されるネットワークカードの MA アドレスを調べたり、各 NFS ROOT マウントするためのディレクトリを作成しなければならなかったりと、結構めんどろであった。数台ならまだコツコツと設定するのだが、69 台度となると大仕事である。

「世の中広いのだから、このような作業を自動的に行ってくれるものがあるのでは」といろいろと探しまわってみたらところ、なんと **ALINKA LCM** というクラスタ構築ソフトがあるではないか !!

ALINKA LCM はフランスの ALINKA 社 (<http://www.alinka.com/>) が開発した PC クラスター管理ソフトで、もともと商用であったが、2000 年の 2 月に GPL2(GNU Public Licence Ver.2 <http://www.gnu.org/copyleft/lgpl.txt>) で公開され、ALINKA 社のサイトよりダウンロードできます。現在対応しているディストリビューションは RedHat 6.0 および Debian 2.1 です。

clump に Debian 2.1 をインストールし、alinka-lcm-1.1-3.deb をインストールし使用していみたところ、もともと商用のソフトだけあって、機能的には十分なものであった。また、ALINKA LCM のグラフィカルユーザインターフェースとして ALINKA RAISIN が商用ソフトとして販売されているが、通常使用するにあたっては ALINKA LCM だけで十分である。また ALINKA LCM はシェルスクリプト、awk, php で記述されているため、比較のカスタマイズがしやすく、今回 Debian 2.2(potato) 用に改良するのも比較的容易であった⁴。

□ Debian GNU/Linux VT-PC Cluster Selection

Debian GNU/Linux は非常に強力なパッケージ管理システムを持つディストリビューションですが、パッケージ数の多さ (Ver.2.1 で約 4000) とパッケージ管理ユーティリティ dselect の扱いづらさから初心者には敬遠されがちでした。しかしながら、それを上回る利点も多々存在するのも事実です。

もともと筆者が Debian ユーザであったこともあり、今回 Debian GNU/Linux 2.2 をベースにクラスター管理システム ALINKA LCM と並列計算に必要なパッケージ (gcc, g77, mpich, pvm, blacs, lapack, scalapack 等) を厳選収録したセレクション集 Debian GNU/Linux VT-PC Cluster Selection 1.0(以下 VTPCS-1.0) を作成しました。VTPCS-1.0 は、Debian GNU/Linux と 100%互換性があり、Debian GNU/Linux の豊富なパッケージを追加インストールすることにより、容易にシステムを拡張することが可能です。

⁴FHS(Filesystem Hierarchy Standard) の採用により、Debian 2.1 とはネットワーク関係の設定ファイルの位置が変更されているのと、PAM ユーザ認証が採用されたことにより、alinka-lcm-1.1-3.deb では正常に動作しない。

■ Debian GNU/Linux VT-PC Cluster Selectionを用いた PC クラスタの構築

□ ハードウェア構成

クラスタ環境を構築するには少なくとも 3 台以上の Linux が動作する PC(サーバが 1 台, スレーブが 2 台以上) が必要となります。またクラスタを管理するサーバは, ネットワークカードを 2 枚以上搭載し, スレーブ (計算) ノードをプライベートアドレスで運用できるような環境をお勧めします。スレーブノードはハードディスク搭載, 非搭載どちらの構成でもクラスタを構築することが可能です。マスターノードはクラスタネットワークに接続するための NIC を少なくとも 1 枚搭載することが望ましく, VTPCS-1.0 をインストールできるだけのハードディスク容量 (システムインストールに 800MB 以上, スレーブノード用に 1 ノードあたり 20MB/node) が必要となります。

スレーブノードは Linux が動作する標準的な PC で, シングル CPU またはマルチ CPU で, フロッピードライブを搭載し, Linux でサポートしている NIC を搭載している必要があります。IDE, SCSI ハードディスクはオプション搭載となります。

クラスタ専用のネットワークはイーサネット, ファーストイーサネットが使用できます。Beowulf では Myrinet などより高度なネットワークをサポートしていますが, ALINKA LCM 1.1 ではサポートしておりません。

□ CD-ROM のディレクトリ構成

```

/+-+
+-boot          ..... CD-ROM ブート用のファイル
+-debian        ..... Debian GNU/Linux Potato のパッケージ
|               (main, contrib,
|               non-free(OGI として配布可能なもののみ))
+-debian-jp      ..... Debian JP Project で配布しているパッケージ
+-debian-non-US  ..... Debian GNU/Linux Potato Non-US パッケージ
+-dists          ..... OGI 独自パッケージ
+-doc            ..... ドキュメント
+ +-ALINKA_LCM   ..... ALINKA LCM ドキュメント
+-inst           ..... インストーラ
+-BootImage      ..... スレーブノード用起動ディスクイメージ

```

□ インストールの準備

ここでは, VTPCS-1.0 をインストールするために必要な事前準備について簡単に説明します。

VTPCS-1.0 のインストーラを起動する方法として, CD-ROM から起動インストールする方法がサポートされています。

インストーラを CD-ROM から起動する場合, お使いのコンピュータが CD-ROM ブートをサポートしている必要があります。コンピュータの説明書あるいは BIOS の設定を調べるなどをして, CD-ROM ブートが可能かどうか確認してください。

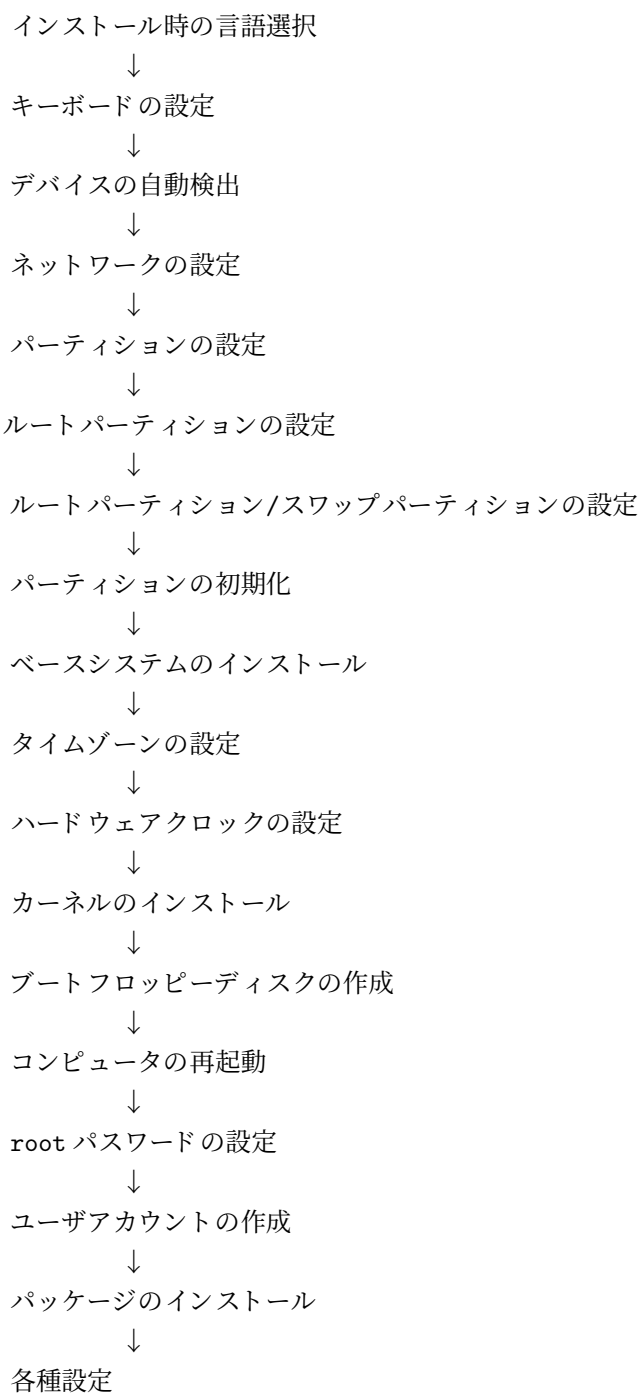
□ モニター情報

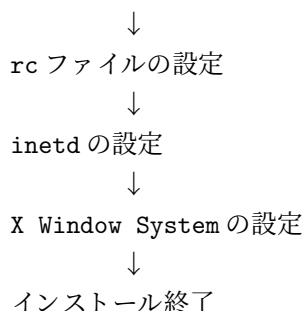
インストールの最後で, X Window System の設定を行います. このとき, お使いのモニター性能 (水平同期周波数, 垂直同期周波数) を選択しなければなりません. モニターに添付されていたマニュアルなどを参照し, モニター性能を調べておいてください.

□ インストール

インストーラの起動

VTPCS-1.0 の CD-ROM をセットした後, コンピュータを再起動してください.
インストールの流れは, 次のようになっています.





インストール時の言語選択

インストーラを起動すると、「VT-PC Cluster」のタイトル画面が表示され、画面下の方に boot プロンプトが表示されます。プロンプトに対して、何も入力せずしばらく放置しておくか、[Enter] だけを入力した場合、インストール時のメッセージが日本語で表示されます。プロンプトに対して、「einstall」と入力すると、インストール時にメッセージが英語で表示されます。どちらか好きな方を選んでください。

キーボードの設定

インストーラの言語タイプを選択し終わると、インストーラが起動し、キーボードタイプの選択画面が表示されます。VTPCS-1.0 でサポートしているキーボードタイプは、JP-106/109 または US の 2 種類ですので、お使いのキーボードに合った方を選んでください。

デバイスの自動検出

キーボードの設定が終わると、インストーラはデバイスの自動検出を行います。自動検出の後、検出されたデバイスに必要なモジュールがインストールされます。起動ディスクを使って起動した場合、モジュールのインストール時に追加モジュールディスクを要求することがありますので、画面に表示されたメッセージに従って、フロッピーディスクを交換してください。

モジュールインストールの終了後、インストールされたモジュールの一覧が表示されるので、確認したら "OK" を押して次に進みます。

ネットワークの設定

複数のネットワークカードに対してネットワークの設定を行うことができます。ネットワークの設定では、「DHCP」、「固定 IP」、「ネットワークなし」のいずれかを選択します。

DHCP を選択する場合は、皆さんのコンピュータが接続されているネットワーク上に DHCP サーバが存在していなければなりません。DHCP サーバが存在しない場合は、「固定 IP」を選んでください。DHCP を選んだ場合、ホスト名の入力画面に移るので、適切なホスト名を入力してください。

「固定 IP」を選択した場合、事前に調べておいた IP アドレス、ネットマスクなどを間違えないように入力してください。あなたのコンピュータが 2 枚ネットワークカードを搭載している場合、1 枚目のネットワーク (eth0) を既存のネットワークに接続し、2 枚目のネットワーク (eth1) をクラスターネットワークに接続し、設定は

```
IP: 192.168.1.1,  
Netmask: 255.255.255.0  
Broadcast: 192.168.1.255
```

としてください。

ネットワークなしの場合も、ホスト名だけ設定することが可能ですので、ホスト名入力画面において、適当なホスト名を入力するようにしてください。ホスト名入力画面でホスト名を入力しない場合、ホスト名が "noname" となりますので気をつけてください。

パーティションの設定

VTPCS 1.0 では、パーティションの設定方法として次の 4 つが用意されています。

1. 自動パーティション
2. メニュー形式
3. fdisk 形式
4. 現状のパーティションを維持

”自動パーティション” は、メニューで選択した領域を使ってインストーラが適切にパーティションを分割します。利用する領域を選択する画面でマークの付け方を間違わなければ、既存のパーティションを壊すようなことはありません。

”メニュー形式”は、メニューを使ったユーザフレンドリーなインターフェイスでパーティションの設定を行うことができます。

”fdisk 形式”では、fdisk コマンドを使ってパーティションの設定を行います。

”現状のパーティションを維持”では、現在のパーティション構成のままで、何も変更しません。

スレーブノードのためのディレクトリを /var/tftpboot に作成します。スレーブノード 1 台あたり約 20MB 必要となりますので注意して、パーティションを作成してください。

ルートパーティションとスワップパーティションの設定

ルートパーティションとして利用するパーティション (Linux, ext2) を指定します。ルートパーティションを指定した後、スワップパーティションも指定します。

マウントポイントの設定

ルートパーティション、スワップパーティション以外に、/usr や/home に別のパーティションを割り当てる場合、ここでパーティションとマウントポイントの設定を行います。

パーティションの初期化

ここでは、ルート、スワップ、そしてそのほかのパーティションの初期化を行います。使用するパーティションの一覧が表示され、すべてに初期化のチェックマークが付いているので、初期化したくないパーティションが存在する場合は、チェックマークを外してください。

チェックマークの指定が終わったら、”OK”を選んでパーティションの初期化に進みます。

ベースシステムのインストール

パーティションの設定が全て終了したら、つぎはベースシステムのインストールです。VTPCS 1.0 では、ベースシステムの取得先として次の 7 種類に対応しています。

- (1) CD-ROM
- (2) FTP

- (3) HTTP
- (4) NFS
- (5) Floppy Disk
- (6) HardDisk(ext2)
- (7) HardDisk(vfat)

それぞれサーバやディレクトリなど、設定が必要な場合は値を入力する画面が表示されるので、適切な値を入力してください。

タイムゾーンの設定

画面に表示されたタイムゾーンの一覧から適当なものを選んで下さい。日本時間にする場合、"Japan"を選択します。

ハードウェアクロックの設定

ハードウェアの時刻を世界標準時として使用するか、ローカル時間 (例えば日本時間) に設定するかを選びます。ハードウェアの時刻をローカル時間とする場合は"はい"を、ハードウェアの時刻を世界標準時に合わせる場合は、"いいえ"を選択して下さい。(ハードウェアの時刻そのものは BIOS で変更することができます)

カーネルのインストール

カーネルのインストールを行います。このとき、カーネルの入手先はベースシステムの入手先と同じものが使用されます。このため、カーネルのインストールは自動的に処理が進みます。

MBR の設定

MBR にブートローダーを書き込みます。既に他のブートローダーあるいはブートセクタがインストールされている場合は、ここで MBR の書き込みを行ってはいけません。間違えて書き込んでしまうと、既存のブートローダーが上書きされてしまいます。

ブートフロッピーディスクの作成

ここでは、ハードディスクからブートできなくなった場合などに必要となるブートフロッピーディスクの作成を行います。ブートフロッピーディスクを作成する場合は、物理フォーマット済みのフロッピーディスクを FDD に挿入し、"はい"を選択してください。ブートフロッピーを作成しない場合は、"いいえ"を選びます。

コンピュータの再起動

ブートフロッピーの作成が終了したら、インストールは後半へと進みます。後半へ進む前に、一度コンピュータを再起動します。CD-ROM ドライブ、FD ドライブから媒体を取り出した後、"はい"を選びます。

root パスワードの設定

スーパーユーザ (root) のパスワードを設定します。パスワードの入力は確認のために 2 回行う必要があります。2 回とも同じものを入力するようにしてください。

パスワードには、英数字と記号が混じったものを使用するようにしましょう。単純なパスワードでは、セキュリティが弱くなります。

ユーザアカウントの作成

インストール時に、ユーザアカウントを作成することができます。ユーザアカウントを作成する場合は、”はい”を選んでください。一つのユーザアカウントを作成した後、再度ユーザアカウントの作成を選ぶことによって、複数のユーザアカウントを作成することができます。このとき、クラスタ管理ユーザとして”lcm”というユーザを作成してください。

パッケージのインストール

クラスタ構築、並列計算に必要なパッケージが自動的にインストールされます。

rc ファイルの設定 (rcconf)

サーバプログラムのうち、使用するプログラムを選ぶことができます。プログラム一覧でチェックが付いているものが使用するプログラムなので、もし使用したくないものがあれば、チェックを外してください。ALINKA LCM はユーザ管理に PostgreSQL を使用しています。PostgreSQL が起動するよにチェックしておいてください。

inetdconf

各種ネットワークサービスのうちどのサービスを提供するかを設定することができます。提供したいサービスにのみチェックをつけてください。

X Window System の設定

X Window System の設定では、まずビデオカード、マウスデバイスの自動検出が行われます (キーボードタイプはインストール時に選択したものがデフォルト値として利用されます)。

自動検出が終了した後、マウスオプションの設定画面が表示されます。現在対応しているオプションは、「3 ボタンマウスのエミュレーション機能」のみです。2 ボタンマウスを使用している場合は、この機能を「使用する」ようにしてください。

次に、お使いのモニターの水平、垂直同期周波数の選択画面に進みます。モニターのマニュアルなどに載っている数値内に収まるものを選びます。選択する値は、できるだけ上限に近いものを選ぶようにします。

次に、使用する解像度を選びます。VTPCS-1.0 では次の 7 種類が選択候補として用意されており、メニューに表示される各解像度の横には、その解像度で利用できる色深度が表示されています。

640 x 320
800 x 600
1024 x 480
1024 x 768
1280 x 1024
1600 x 1200

次に、色深度を選びます。ビデオカードに搭載されているメモリサイズと選択した解像度を考慮して、利用可能な色深度だけがリストアップされます。

色深度を選んだ後、これまでの設定を元に最適なビデオモードを選択します。最適なビデオモードを選択するために、画面がブラックアウトしますが、これは正しい動作です。しばらくすると、X サーバが起動し、正しく表示されているかどうかを問うウィンドウが表示されています。画面が正しく表示されている場合は、“終了”ボタンを押して、表示テストを終了させてください。

表示テスト終了後、新しく作成した XF86Config ファイルを /etc/X11 配下にインストールするかどうか聞いてくるので、インストールする場合は、“はい” をインストールしない場合は“いいえ”を選んでください。“いいえ”を選んだとき、新しい XF86Config は /etc/X11/XF86Config.oxsetup-new という名前で保存されます。再度設定をやり直す場合は、“再設定”を選んでください。マウスオプションの設定へ戻ります。

インストール終了

X Window System の設定が終了したら、VTPCS のインストールは完了です。

□ ALINKA LCM の設定

ALINKA LCM の環境の設定

スーパーユーザで

```
# /usr/local/alinka-lcm/alinka-lcm_install
```

を実行し、ALINKA LCM のシステムを使用するための準備を行います。このスクリプトは、/etc/lcm.conf に必要な設定事項を設定するためのもので画面に表示されている質問事項に答えていくことで、/etc/lcm.conf の設定が行えます。画面に表示される質問事項に順次答えていってください。eth0 を外部ネットワークに接続、eth1 をクラスタネットワークに接続し、eth0 の IP アドレスを 192.168.1.1、ネットマスクを 255.255.255.0、ブロードキャストアドレスを 192.168.1.255 とした場合、全ての設定をデフォルト値に設定して問題なく動作します。/etc/lcm.conf の設定に関しては ALINKA LCM のマニュアルを参照してください。

```
# alinka-lcm_install
```

で/etc/inetd.conf ファイルに bootpd の設定が追加されるのでこれを有効にするため inetd を再起動する。

```
# /etc/init.d/inetd restart
```

スレーブノード用ブートフロッピーディスクの作成

スレーブノード用のカーネルイメージが/cdrom/BootImage ディレクトリに納められています。対応するネットワークカード毎にブートイメージがありますので、ご使用のネットワークカードに対応するブートイメージをご使用ください。

```
# mount /cdrom
# superformat /dev/fd0
# dd if=/cdrom/BootImage/bzImage_2.2.14-xxxx of=/dev/fd0
# rdev /dev/fd0 /dev/nfsboot
```

ご使用のネットワークカード対応のブートイメージが存在しない場合はカーネルをコンパイルし、ブートイメージを作成してください。

```
# cd /usr/src
# tar Ixvf kernel-source-2.2.14.tar.bz2
# cd kernel-source-2.2.14
# make xconfig
```

以下の設定を有効にしてください。

```
=====
Network options --->
[*] IP: kernel level autoconfiguration
[*]      BOOTP support
[*]      RARP support
```

```
Filesystems --->
Network File Systems --->

<*> NFS filesystems support
[*]   Root file system on NFS
```

以下はマシン依存

```
Network device support --->
Ethernet (10 or 100Mbit) --->
```

=====

カーネルのコンパイルをし、ブートイメージをディスク書き込む。

```
# make-kpkg
# superformat /dev/fd0
# dd if=/usr/src/kernel-source-2.2.14/arch/i386/boot/bzImage of=/dev/fd0
# rdev /dev/fd0 /dev/nfsboot
```

スレーブノードの起動

マスターサーバでmk_node_daemon を起動し、その後先に作成した FD を用いて、各スレーブノードを起動します。

```
# /usr/local/alinka-lcm/sbin/mk_node_daemon start root &
```

全てのスレーブノードが起動しおえたら、mk_node_daemon は停止してください。

```
# /usr/local/alinka-lcm/sbin/mk_node_daemon stop &
```

ユーザの追加

ALINKA LCM システムを使用するユーザを追加します。

```
# adduser foo
```

ユーザ foo にクラスタの実行権限を与えます。

```
# su lcm -c "/usr/local/alinka-lcm/bin/mk_permission foo root user"
# su lcm -c "/usr/local/alinka-lcm/sbin/sync_perm -c root"
```

mk_permission コマンドはユーザfoo に (root) クラスタのユーザ使用権限を与えます。
sync_perm コマンドはスレーブノードの/etc/passwd ファイルの更新を行います。

□ MPIプログラムの実行

ユーザ foo でログインし, MPI のサンプルプログラムを実行してみます。

```
$ cp /usr/share/doc/mpich/examples/code/pi.c /home/foo
$ mpicc cpi.c -o cpi
$ mpirun -i cpi
```

■ ALINKA LCM ドキュメント

□ 免責事項

警告：この説明書は ALINKA LCM クラスタ マネージメント ツールに関するものです。ALINKA は ALINKA LCM を使用することに起因する如何なる損害 (直接的/間接的を問わず) に対しても責任はありません。

□ ALINKA LCM とは

原理

ALINKA LCM で作成された Linux クラスタは 1 つのマスターと複数のスレーブ・ノードで構成されます。

- “スレーブノード” とは計算を行うノードです。スレーブノードへの OS のインストールと管理は ALINKA LCM によって行なわれます。スレーブノードはマスターノードの”クローン”です。つまりもし libfoo がマスタにインストールされている場合には、ALINKA LCM は libfoo をスレーブにインストールします。
- “マスターノード” はクラスタの管理をするために使用します。またスレーブにソフトウェアをインストールする際のテンプレートとして使用されます。デフォルトではマスターノードはスレーブノードに対する NFS ファイルサーバ (主として /home を共有) です。マスターノードはスレーブノードとは異なり計算は行いません。マスターノードは外界とスレーブノードとのリンクを行うだけです。

物理的なクラスタの連結

通常全てのノード (マスターとスレーブ) はネットワークインターフェースカード (以降 NIC と略記。例えばファーストイーサネット) を使って同一のスイッチまたはハブに接続されます。またマスターノードは 2 枚目の NIC で他のネットワークに接続する構成をすることも可能です。

IP ネットワーキング

全てのノード (マスターとスレーブ) は TCP/IP を使用します。通常 192.168.1.XXX のようなプライベート IP アドレスを使用しますが、config ファイルである `/etc/lcm.conf` で任意の IP アドレスを設定することもできます。またクラスターマスターに対する 1 つの IP アドレスが IP プールから必要になります。

ノード生成

全てのスレーブノードは bootp リクエスト経由でスタートアップ時に IP アドレス取得のリクエストを投げます。マスターはこのリクエストを検知し、自分の tftpboot ディレクトリにスレーブノードのためのルートディレクトリを作成します (その際、ディスクの使用量を減らすためにハードリンクが使用されます)。そしてスレーブノードはマスターノードから自分のルートディレクトリを NFS でマウントします。

- もしスレーブノードにハードディスクがない場合、クラスタは全てのファイルアクセスを NFS 経由で行います。
- スレーブノードがハードディスクを持っている場合、それを使用することもできます(/etc/lcm.conf で設定します)。スレーブの/etc と/var は、その場合でも NFS サーバ上にあるため、管理が容易です(ファイル数は多くなく、また容量も大きくありません)。他の全てのファイル(/bin /dev /lib /sbin と /usr)はサーバからローカル ハードディスクにコピーされます。/tmp と/scratch とスワップはスレーブノードのローカルハードディスク上にあります。

□ インストールに必要な環境

□ 必要なハードウェア構成

マスターサーバ

マスターサーバは下記の仕様を持つ標準 PC です。

- Linux がインストールでき、実行できるもの
- Linux ディストリビューション, ALINKA LCM, その他の必要なアプリケーションがインストールできるハードディスクを搭載していること。
- クラスタネットワークに接続するための少なくとも 1 枚の NIC が必要となります。マスターサーバを外部ネットワークと接続するためには 2 枚目の NIC を搭載することが望ましい。
- CPU: インテル x86 互換であること。スレーブノード数に相応する処理能力を持つこと。
- メモリ: スレーブノードの数に比例して搭載メモリ容量も必要となります。
- 使用する全てのハードウェア (特に NIC) は、Linux オペレーティングシステムをサポートしているものでなければなりません。
- ソフトウェアに関しては、マスターサーバのハードディスクには Linux ディストリビューションインストールされていることが必要です。使用するディストリビューションは ALINKA LCM でサポートされている必要があります (またはパッチを当てる必要があります)。

スレーブノード

スレーブノードも標準 PC 構成でなければなりません。シングル CPU またはマルチ CPU で、IDE または SCSI のハードディスクをオプション搭載することが可能です。またフロッピードライブと Linux がサポートしている NIC が必要となります。CPU の能力とメモリ容量は使用するアプリケーションによって異なります。再度指摘しますが、必要なハードウェアで重要なのは Linux で使用できる NIC です。ハードウェアと Linux オペレーティング システムの設定に関する問題のリスクを最小限にするために、各ノードに対して同一のハードウェアを使用することをお勧めします。

ネットワーク・ハードウェア

アプリケーションによっては、クラスタリング専用ネットワーク (ダム ハブ, またはファースト イーサネットスイッチ) を使用することができます。Beowulf はより高度なネットワークハードウェアを使

用することができますが, ALINKA LCM version 1.1 ではサポートしていません. ALINKA LCM のオペレーションは Linux の機能に従属しています. 大切なことは Linux オペレーティングシステムの稼動保証のあるハードウェアを使用することです.

□ ALINKA LCM のインストール

□ 必要なソフトウェア

ALINKA LCM を使用するには正常に動作する Linux(Debian 2.1, 2.2 または Redhat 6.0 は問題ありません)のほかに, 次のパッケージが必要になります. tar - nfs-server - sudo - php cgi-bin(pgsql をサポートしていること) - bootpd - tcpdump - postgresql - gawk 勿論mpichあるいはpvm3を必要とするソフトウェアを使用している時は, これらのライブラリも必要となります. 使用しているディストリビューションが Debian slink (2.1), potato (2.2) またはそれ以上であれば, これらのソフトウェアはディストリビューションに含まれています. 使用しているのが Redhat (6.0) の場合, 必要なパッケージの幾つかがディストリビューションに含まれていません. それらは RPM として供給され, ソースは ALINKA の WEB サイト <http://www.alinka.com> から入手できます. これらのパッケージをインストールする必要があります. Version 1.1 は NIS および Shadow passwords のサポートはしておりません.

□ インストール

Redhat 6.0 を使用している場合は, root アカウントで次のコマンドを実行してください.

```
# rpm -i alinka-lcm-1.1-0.i386.rpm
```

Debian を使用している場合は, root アカウントで実行してください.

```
# dpkg -i alinka-lcm_1.1-0_i386.deb
```

またクラスタに対する管理者アカウントを作成しなければなりません. この説明書では lcm を管理者アカウントとして使用します.

□ 設定

パッケージをインストールした後, root アカウントで下記のように alinka-lcm_install スクリプトを実行します.

```
# /usr/local/alinka-lcm/sbin/alinka-lcm_install
```

このスクリプトを実行すると /etc/lcm.conf ファイルに必要な設定に関する幾つかの質問が表示されます. 多くのフィールドがありますが, 下記に述べるフィールドに関してのみ, 答える必要があります. /etc/lcm.conf に関する詳細に関しては対応する章をご参照ください.

□ セットアップに必要な /etc/lcm.conf なファイルのフィールド

設定質問 1

What is the name of the LCM administrator? (default lcm)

LCM 管理者の名前は? (デフォルトは lcm です)

解説:

クラスタ管理者の名前を指定します。

Field:

LCM_ADMINISTRATOR="lcm"

設定質問 2

Where is your Tftpboot directory ? (default /var/tftpboot)

Tftpboot ディレクトリの場所は? (デフォルトは/var/tftpbootです)

解説

このディレクトリにはスレーブがブートするのに必要な最小限のファイルセットとスレーブの/etcと/varが格納されます。この設定に使用するディスク容量を減らすためにこれらのファイルは"Template"からのハードリンクを張っています。

Field

LCM_TFTPBOOT="/var/tftpboot"

設定質問 3

Please enter the complete path of your cgi-bin php

cgi-bin php の完全なパスを入力してください。

解説

インストーラは通常このプログラムを自動的に見つけますが、見つけられなかった時、上記の表示が行われます。(このプログラムは ALINKA LCM が必要とする外部プログラムです。)

Field

LCM_PHP3CGI="/xxx/xxx/php"

設定質問 4

Please specify your external network interface (not cluster)

外部ネットワークに接続するインターフェース (クラスタに接続するインターフェースではありません) を指定してください。

解説

外部ネットワークと通信するための NIC の指定します。マスターに NIC を 1 枚しか使用していない時は外部との通信は行われず、通常 eth0 であり、0 と答えてください。2 枚以上の NIC を使用している場合、外部ネットワークとの通信に使用する NIC のデバイス番号を指定してください (外部ネットワークに対する設定をしておく必要があります)。

Field

LCM_INTERNET_IF="eth0"

設定質問 5

Please specify your cluster network interface

クラスタに接続する NIC の番号を指定してください。

解説

クラスタとの通信に使用する NIC の番号を指定します。マスタで NIC を一枚しか使用していない場合は、通常それは eth0 であり、そうであれば eth0 と答えます。もし 2 枚以上の NIC を使用している場合には、クラスタ側に接続する NIC の番号を指定します (ALINKA LCM は NIC 設

定は行いません。ディストリビューションインストール時に設定してください)。通常この NIC はクラスタが使用するハブまたはスイッチに接続されています。

Field

LCM_LOCALNET_IF="eth0"

設定質問 6

Please specify your network address on your cluster interface (default 192.168.1.0)

クラスタインターフェースのネットワークアドレスを指定してください(デフォルト値は192.168.1.1です)。

解説

デフォルトはクラス C のプライベートアドレス192.168.1.0 となっています。

Field

LCM_NETWORK="192.168.1.0"

設定質問 7

Please specify your network mask on your cluster interface (default 255.255.255.0)

クラスタインターフェースのサブネット マスクを指定してください(デフォルト値は255.255.255.0です)。

解説デフォルトで使用する IP アドレスはクラス C になっていますので、デフォルトのサブネット マスクも 255.255.255.0 となっています。

Field

LCM_NETMASK="255.255.255.0"

設定質問 8

Please specify your broadcast address on your cluster interface (default 192.168.1.255)

クラスタインターフェースのブロードキャスト アドレスを指定してください(デフォルト値は192.168.1.255です)。

解説

ブロードキャストアドレスを指定します。デフォルトの値はデフォルトの設定では問題ありません。

Field

LCM_BROADCAST="192.168.1.255"

設定質問 9

Please specify the LCM master's address on your cluster network (default 192.168.1.1)

クラスタネットワーク上の LCM マスターノードの IP アドレスを指定してください(デフォルト値は192.168.1.1です)。

解説

マスターのクラスタ側の IP アドレスを指定します。

Field

LCM_MASTER="192.168.1.1"

設定質問 10

What is the LCM master's name? (default servername)

LCM マスタの名前は? (デフォルト値は servername です)

解説

マスターノードの名前を指定します。

Field

LCM_MASTERNAME="servername"

設定質問 11

Please choose the nodes' root name : (With node they will be called node-xxx)
(default node)

ノードのルート名を選択してください。例えば node とした場合、各ノードの名前は node-xxx となります (デフォルト値は node です)。

解説

スレーブノードはその IP アドレスによってコールされます (最初のノードは 101 から始まります)。スレーブノードの名前はここで指定するルート名に従って割り当てられます。例えばデフォルトである "node" をルート名とした場合、各スレーブノードの名前は node-101, node-102, node-103 となります。

Field

LCM_NODENAME="node"

設定質問 12

Please enter your cluster domain name: (default cluster)

クラスタ側のドメイン名を入力してください (デフォルト値は cluster です)。

解説

スレーブのドメイン名を指定します。デフォルトの場合、node-101 のフルネームは node-101.cluster となります。

Field

LCM_DOMAINNAME="cluster"

設定質問 13

Do you want to use the hard disks on the nodes? [yes/no]

(They will be wiped out) (default no)

ノードのハードディスクを使用しますか? [yes/no] yes の場合スレーブの HDD の内容は完全に削除されます (デフォルト値は no です)。

解説

スレーブノードがハードディスク (SCSI または IDE) を持っており、それを使用する場合には yes と答えてください。その場合ハードディスクの内容は完全に削除されますのでご注意ください。この機能を使用するには、スレーブはディスクをサポートする Linux kernel でブートする必要があります。

Field

USE_NODE_HARD_DISKS="no"

*この質問に yes と答えた場合には、ディスクを設定するための以下の質問に答える必要があります。

ます。この質問にno と答えた場合には設定質問 18 までスキップしてください。

設定質問 14

Please enter the nodes files image location (default /var/tftpboot) ノードファイルイメージを置く場所を入力してください (デフォルト値は/var/tftpboot です)。

解説

マスターはスレーブで必要とする全てのファイルを tar でまとめて保持します (500MB 以上になるかも知れません)。ファイル名はnodefiles.tgz で、このファイルが置かれる場所を入力します。

Field

FILES_IMAGE_LOCATION="/var/tftpboot"

設定質問 15

Please enter the node slash size (default +1000M) ノードのルートファイルシステムのサイズを入力してください (デフォルト値は+1000M です)。

解説

スレーブノード上のルートファイルシステムはマスターと同じファイルを扱うのに必要な十分な容量が必要となります。マスターが使用している容量をチェックするには、du -s /usr コマンドを使います。得られた値より少なくとも 100M オクテット多い値を指定してください。指定は+記号に続いてメガオクテットの数字、そして大文字の M を入力します (例 +800M, +1300M 等です)。+1000M は 1 ギガオクテットを意味します。

Field

NODE_SLASH="+1000M"

設定質問 16

Please enter the node swap size (default +800M) ノードのスワップサイズを入力してください (デフォルトは+800M です)。

解説

スレーブノードで使用するスワップサイズを指定します。通常はメモリ容量の 1 から 3 倍の値を指定します。入力形式は+記号に続いて、単位がメガオクテットの数字、最後が大文字の M となります (例。 +60M, +500M, 等)。

Field

NODE_SWAP="+800M"

設定質問 17

Now enter the node tmp size (default +1000M)

ノードのtmp ディレクトリのサイズを入力してください (デフォルト値は+1000M です)。

解説

スレーブのtmp ディレクトリの大きさを指定します。残された全ての容量はユーザが読み書きに使用できる/scratch ディレクトリとしてマウントされます。/tmp はスタートアップ時にクリアされることにご注意ください。指定は+記号に続いてメガオクテットの数字、そして大文字の M を入力します (例 +800M, +1300M 等です)。+1000M は 1 ギガオクテットを意味します。

Field

```
NODE_TEMP="+1000M"
```

設定質問 18

Please enter the PVM path (default /usr/share/pvm3)

PVM パスを入力してください (デフォルトは /usr/share/pvm3 です).

解説

PVMを使用する場合には、PVM_ROOT のパスを入力してください (Redhat では /usr/share/pvm3, Debian では /usr/lib/pvm3 となります). Redhat を使用して PVM を使用する場合には、環境変数 PVM_ROOT がシェルによってエクスポートされていることをご確認ください (例えば、zsh では .zshrc ファイルに PVM_ROOT=xxx/xxx/xxx の行を挿入してください). Debian の PVM ではデフォルトで /usr/lib/pvm3 にパッチが当てられていますから、この操作は不要です.

Field

```
PVMROOT="/usr/share/pvm3"
```

設定質問 19

Please enter the LCM nodes monitoring frequency (in minutes) (default 15)

LCM ノードのモニタリング頻度を分で指定してください (デフォルトは 15 分です).

解説

指定されたX分間隔でマスターはスレーブノードに対して負荷を確認する問い合わせをおこないます. 負荷データをより多く、あるいはより少なく入手するにはこの値を増減させます. デフォルトは 15 分間隔で平均負荷がモニタされます. /etc/lcm.conf 内の LCM_LOAD パラメータを 1 または 5 に変更することができます.

Field

```
LCM_LOAD_FREQ=15
```

□ インストールの終了

/etc/lcm.conf をチェックし、変更することも可能です. またインストールを続行することも可能です. インストールプロセスは ALINKA LCM が動作するよう設定をします. もし ALINKA LCM が動作しない場合には、/etc/lcm.conf を編集し、alinka-lcm_install を再度実行してみてください. またはスーパーユーザー権限で手動で cf_master, mk_db, mk_template を実行してください.

□ ALINKA LCM のアンインストール

新しいバージョンをインストールする等の理由で、ALINKA RAISIN をアンインストールするにはスーパーユーザーで次のプログラムを実行してください.

```
# /usr/local/alinka-lcm/sbin/alinka-lcm_uninstall
```

/tftpboot ディレクトリが消去され、また alinka-lcm のデータベースを削除されます. 最後に alinka-lcm パッケージを削除してください.

□ 使用方法

ALINKA LCM のインストールが終了した後、クラスタシステムを使用するために幾つかの作業が必要です.

□ ブートディスクの作成

クラスタにスレーブノードを追加する時、多くの場合スレーブノード起動用のブートフロッピーを作成する必要があります。Linux カーネルのブートディスクを作成する時と作業は同じですが、次のオプションを選択する必要があります (モジュールとして組み込むのではなく直接カーネルに組み込んでください)。

- IP: カーネルレベルで BOOTP と RARP をサポートすること。
- NFS ファイルシステムをサポートし、更に ROOT NFS ファイルシステムをサポートすること。
- スレーブノードが使用する NIC のドライバがインストールされていること。
- HDD を使用する場合は、そのドライバ (SCSI, IDE) が使用可能になっていること。

次にカーネルが nfs でブートできるようにするよために、次のコマンドを root で実行してください。

```
# mknod /dev/nfsboot c 0 255
# rdev MyKernelFile /dev/nfsboot
```

次にカーネルをフロッピーにコピーします。

```
# dd if=MyKernelFile of=/dev/fd0
```

もしカーネルに渡したい他のパラメータがある場合、lilo を使ってカーネルをフロッピーからロードすることもできます。

□ クラスタの構築

スレーブノードをハブまたはスイッチに接続します。次に ALINKA LCM デーモン `mk_node_deamon` をマスターノードで起動し (`/usr/local/alinka-lcm/sbin/mk_node_deamon start root`)、スレーブノードのフロピードライブに起動ディスクを入れ、電源スイッチを ON します。デーモンは `pid` と `bootp` をリクエストをしたスレーブノードの MAC アドレスを `/usr/local/alinka-lcm/tmp` ディレクトリ下のファイルにセーブします。マスターノードが `bootp` リクエストを検知した時、`LCM_TFTPBOOT` ディレクトリにそのスレーブノードのルート ディレクトリを作成します。その後、`/etc/bootptab` ファイルを更新し、`bootpd` がリクエストに応答します。約 2 分後にスレーブノードは稼動を始めます。全てのスレーブノードが立ち上がり、稼動を始めたら ALINKA LCM デーモン `mk_node_deamon` を停止することができます。これを行うには `/usr/local/alinka-lcm/sbin/mk_node_deamon stop` を実行してください。

クラスタは稼動しましたが、ユーザがクラスタにログインできるようにする必要があります (デフォルトではユーザ `lcm` のみがログインできるようになっています)。

□ 並列プログラムの稼動

まず `mk_permission` でクラスタの使用を認めるユーザの登録を行います。次にスレーブノードに、マスターノードでスーパーユーザ権限で `sync_perm` を実行し各スレーブノードの `/etc/passwd` ファイルを更新するように指示します。

```
# mk_permission user_name cluster_name user
# sync_perm -c root
```

以上でユーザはクラスタを使用することができるようになります。並列タスクの実行は `lcmpirun` コマンドと `lcmpvmrun` コマンドを使って行うことができます。必要の場合はクラスタ名として “`root`” (デフォルトクラスタ名) を指定します。これらのコマンドの詳細な説明は次章をご参照ください。

□ ディレクトリの説明

□ ディレクトリ構成

この章では、ALINKA LCM が使用する各ファイルが `/usr/local/alinka-lcm` ディレクトリのどこにあるかを説明します。

bin

このディレクトリには ALINKA LCM で使用する全ての標準ユーザコマンドが格納されています (詳細は 5.5.18 章をご参照ください)。

debian_potato - redhat_hedwig - mandrake_venus ...

各ディストリビューション固有のファイルが格納されているディレクトリです (mandrake 6.0 のファイルも含まれていますが、確認試験はされていません。mandrake 6.0 には何らかのパッチが必要になるかもしれません)。

log

このディレクトリは ALINKA LCM の管理者 (デフォルト `lcm`) が読み取り、書き込みができなければなりません。様々なログファイルが記録されます。

- `node_hostname.load`

ノード `hostname` に対する負荷を記録し、その更新頻度は `lcm.conf` ので設定できます。カラム 1 に負荷が登録された時間 (`date + %s`)、カラム 2 に負荷の平均値 (`lcm.conf` に依存) を記録しています。

- `clustername.cl_load`

各 `$LCM_FREQ` 分 (`lcm.conf` ファイルの `frequency` で設定) ごとのクラスタ `clustername` に対する負荷を記録します。カラム 1 は負荷が登録された時間 (`date + %s`)、カラム 2 は平均負荷 (mean of nodes/number of proc. of node) を記録します。停止またはダウンしているノードは 0 負荷として記録されます。

php3

ALINKA LCM と PostgreSQL データベースとの関係のために使用する `php3` スクリプトがインストールされています (詳細は `php3` の章をご参照ください)。

sbin

管理者用コマンドのためのシェル スクリプトがインストールされています。

tmp

ALINKA LCM のテンポラリファイルを書くために使用するディレクトリです。ALINKA LCM の管理者 (lcm) が書き込みと読取りができなければなりません。

□ bin ディレクトリ

cluster_command

- 名前

cluster_command: クラスタの各スレーブノードでコマンドを実行

- 書式

cluster_command [クラスタ名] [コマンド]

- 説明

クラスタ内の全てのスレーブノードでコマンドを実行します。指定されたクラスタ名に属する各スレーブノードで任意のコマンドを実行するためにrshを使用します。これはrootとして実行されないことにご注意ください。

- 戻り値

なし

- EXIT CODES

1: 使用エラー

gt_load

- 名前

gt_load: 負荷ログ ファイルの更新を行います。

- 書式

gt_load [-hv]

- 説明

gt_load コマンドは存在する全ての負荷ログファイルを更新します。このコマンドはモニタリングプロセスを稼働させるためにコマンドラインでコールすることができます。通常cronによって\$LCM.FREQUENCY 分ごと (/etc/lcm.conf で設定できます) に実行され、(test_node コマンドを呼び出すことで)、ノードのステータス (Running, Down, Halted, ...) を ALINKA LCM データベースに更新します。更に、ノーマルモードとテストモードのモード変換を行います。ノーマルおよびテストモードでのgt_load に対する頻度は/etc/lcm.conf ファイルで設定されます。通常テストモードは 1 分で、いずれにしてもノーマルモードよりずっと小さな値にすべきです。gt_load コマンドは ALINKA LCM データベースとクラスタが連動していることを確認する唯一の方法です。test_node コマンドを使うことは実際には余り良い方法ではありません。と言うのはrshがリクエストに応答しない時、自動的にステータスに変更されるからです。従ってもしコンピュータをリブートし、およそ 30 秒に 5 回test_node コマンドを実行するとすれば、実際にはノードステータスをBooting[...]ではなく、Boot_failedと認識し設定してしまいます。

gt_load コマンドは全てのノードでtest_node コマンドをバックグラウンド実行し、45 秒間休止します。それからクラスタおよびサブクラスタの負荷ファイルを更新します。また CPU 情報がある全ての稼動ノードに対するcpu_information をコールします。

このスクリプトはcronによって自動的に起動され、自動的にテストモードからノーマルモードに変更、またノーマルモードからテキストモードに変更されます。cpu_information コマンドは新しいノードがスタートした、あるいはノードがリスタートした時、1 回または 2 回だけ実行すべきです。モードをチェックするには/etc/crontab ファイルを見てください。45 秒以内にgt_load を 2 回コールするテストモードを実行することができます。

```
$ ....alinka-lcm/bin/gt_load -v &
$ ....alinka-lcm/bin/gt_load &
```

最初の行はgt_load コマンドをバックグラウンドで実行し、2 行目はgt_load が既に実行されていますので、テスト モードに変更されます。ALINKA RAISIN のユーザの場合、[show explorer] をクリックする、あるいは任意のモードをクリックし、ツリーのクラスタまたはサブクラスタをクリックすることで自動的に実行することができます。

- EXIT CODES

- 0: クラスタは安定モード (ノーマル モード)
- 1: 使用
- 2: クラスタは非安定モード (テスト モード)

lcmpirun

- 名前

lcmpirun: クラスタで MPI プログラムを実行するためのラッパープログラム

- 書式

```
lcmpirun [-a attime] [-i] [-f filename] [-c clustername] [-d directory] [-n nice]
        [-p processors] program [arguments]
```

- 説明

lcmpirun コマンドは既存のクラスタで MPI プログラムを実行するためのシェルコマンドツールです。この場合指定したクラスタで MPI プログラムを実行する権限が必要です。実行許可されている場合には任意の MPI プログラムを任意のクラスタで実行することができます。また何個の CPU が必要かを指定することができます。lcmpirun, rsh コマンドは最初のスレーブノードで実行され、mpirun コマンドを実行します。rsh ですのでlcmpirun の出力はターゲットプログラム自身の出力に比べ遅れを生じます。lcmpirun コマンドはクラスタで稼動している全てのノードを含む “machine file” を作成し、リモートの最初のスレーブノードにラッパースクリプトを作成します。このスクリプトがmpirun と共に指定されたプログラムを実行します。指定されたプログラムがインタラクティブなものである場合、スタート後SDTOUT とSTDERR がターミナルスクリーンに表示されます。インタラクティブでない場合、at コマンドで実行され、SDTOUT とSTDERR はマスターノードの\$LCM.LOG ディレクトリにあるログファイル\$ID.out と\$ID.err にリダイレクトされます。注意してほしいのはat コマンドはマスターノードで実行され、rsh を持つリモートラッパーを実行するということです。

-c オプションを使用しないと、クラスタ名としてroot が使用されます。

-a オプションは後で実行するためのスケジューリング、

- i オプションはログファイルを使用しないで結果は画面に表示,
- f オプションは入力ファイルを使用,
- d オプションは作業用ディレクトリの定義,
- p はプログラムが必要とする CPU の数を指定します.

あるプログラムが実在する CPU 以上の CPU を要求した場合, MPI はこのプログラムをあるノードで 2 回以上実行することになり, 勿論効率的ではありません. また `lcmpirun program </datafile>` のシンタックスは動きませんので使わないでください. その代わりに `-f` オプションを使用してください.

• オプション

-a atime

MPI プログラムがスタートする日付と時間を指定します. (例 12:12 Dec 12', '4pm + 3 days') 詳細に関してはオンラインマニュアルをご参照ください. もし時間を指定しなかった場合には即座に開始されます.

-i

インタラクティブの場合, バックグラウンドで開始せず, また結果を `$LCM_LOG` にログしないで, `STDOUT` と `STDIN` に表示します.

-f

`STDIN` をファイルからリダイレクトします.

-c cluster

プログラムを実行するクラスタの名前を指定します. サブクラスタがない場合は, デフォルトは `root` となります.

-d directory

作業用ディレクトリの指定します.

-n nice

プライオリティを 0-19 で指定します. 0 が高で 19 が低になります. デフォルトは 0 (バージョン 1.1 では使用できません)

-p processors

使用する CPU の数を指定します. デフォルトは `all` となっています.

-u username

MPI プログラムを実行できるユーザ名を指定します, クラスタ管理者のみ使用可です.

• EXIT CODES

- 1: 使用
- 2: このユーザにはクラスタでプログラムを実行することが認められていない.
- 3: ディレクトリまたはプログラムが存在しないか, `/home` ディレクトリにない.
- 11: マスタノードにテンポラリマシンファイルが作成できない. これは実際には指定されたクラスタで稼働しているスレーブノードがないことを意味します.
- 12: マスタノードにテンポラリプログラムファイルが作成できない.
- 13: マシンまたはプログラムをリモートへ移動できなかった.
- 21: データベースにタスクを挿入できなかった.
- 22: スケジュールができなかった

- **名前**

lcmpvrun: クラスタで PVM を実行するプログラム.

- **書式**

```
lcmpvrun [-a attime] -c clustername [-d directory] [-n nice]
        [-p processors] program [arguments]
```

- **説明**

lcmpvrun は既存のクラスタで PVM プログラムを実行するためのシェルコマンドです. PVM プログラムをクラスタで実行するユーザ権限が必要です. 権限がある場合には任意の PVM プログラムを実行することができます. lcmpvrun コマンドはpvmdを開始し, PVM プログラムを実行するクラスタの最初のノードにrshします. 使用しているのディストリビューションがRedhatであり, PVMを使用したい場合には, 環境変数PVM_ROOTがシェルによってエクスポートされていることを確認してください (例えばzshの場合, .zshrc ファイルに次の行を挿入します. PVM_ROOT=/xxx/xxx/xxx). 使用しているのが Debian の場合にはこの必要はありません. /usr/lib/pvm3 に既にパッチが当てられています.

- **オプション**

-a atime

MPI プログラムがスタートする日付と時間を指定します. (例 12:12 Dec 12', '4pm + 3 days') 詳細に関してはオンラインマニュアルをご参照ください. もし時間を指定しなかった場合には即座に開始されます.

-c cluster

プログラムを実行するクラスタの名前を指定します. サブクラスタがない場合は, デフォルトはrootとなります.

-d directory

作業用ディレクトリの指定します.

-n nice

プライオリティを0-19で指定します. 0が高で19が低になります. デフォルトは0(バージョン 1.1 では使用できません)

-u username

PVM プログラムを実行できるユーザ名を指定します, クラスタ管理者のみ使用可です.

- **EXIT CODES**

- 1: コンフィギュレーションファイル/etc/lcm.conf が読めません.
- 10: コマンドラインにクラスタ名がありません.
- 11: コマンド ラインにプログラム名がありません.
- 12: クラスタにノードがありません.
- 13: タスクをキューする時にエラーが生じました (プログラムが存在しないのかもしれませんが)

mk_db

- **名前**

mk_db: ALINKA LCM のための postgres データベースを作成します.

- **書式**

mk_db

- **説明**

`mk_db` は ALINKA LCM 用のデータベースを作成します。ALINKA LCM をインストールする時, 1 回だけ実行してください。実際は `alinka_lcm_install` によって自動的にコールされます。

`mk_cluster`

- **名前**

`mk_cluster`: 新しいクラスタを作成するプログラム。

- **書式**

```
mk_cluster father_name son_name
```

- **説明**

`mk_cluster` は (サブ) クラスタ内に新しいサブクラスタを作成します。作成する (サブ) クラスタに対する管理者権限が必要です。 `father_name` は新しいサブクラスタを作成するための親クラスタ名であり, これは ALINKA LCM データベースに既に存在している必要があります。 `son_name` は新しく作成するサブクラスタ名を意味します。

- **EXIT CODES**

3: 誤った使用

10: クラスタが作成できなかった

`mk_permission`

- **名前**

`mk_permission`: (サブ) クラスタの使用許可を設定するプログラム

- **書式**

```
mk_permission username clustername administrator|user
```

- **説明**

`mk_permssion` は (サブ) クラスタの使用許可を設定するシェルコマンドです。ユーザに (サブ) クラスタの使用または管理の許可を与えるために使用します。このコマンドを実行するには許可を変更する親クラスタの管理者権限が必要です。 `username` は許可を与えるユーザの名前, `clustername` はユーザに許可する (サブ) クラスタの名前, `administrator|user` はアドミニストレーションかユーザ許可を選択するキーワードです。

- **EXIT CODES**

3: 誤った使用

10: ユーザが存在しないか, 許可が設定できなかった。

`mv_node`

- **名前**

`mv_node`: ノードをクラスタに移動するためのプログラム

- 書式

```
mv_node hostname clustername
```

- 説明

`mv_node` はクラスタからノードを移動するためのシェルコマンドです。このコマンドを実行するには親クラスタの管理者権限が必要です。 `hostname` は移動するノードの名前, `clustername` は移動先のクラスタ名を意味します。

- EXIT CODES

3: 誤った使用

10: ユーザが存在しないか, 許可が設定できなかった。

`rm_cluster`

- 名前

`rm_cluster`: クラスタを削除するプログラム

- 書式

```
rm_cluster cluster_name
```

- 説明

`rm_cluster` はサブクラスタを (サブ) クラスタから削除します。 `rm_cluster` を実行するには削除したいサブクラスタの親クラスタの管理者権限が必要です。 `cluster_name` は削除するサブクラスタの名前を指します。このサブクラスタは ALINKA LCM データベースに存在していなければなりません。

- EXIT CODES

3: 誤った使用。

10: クラスタが存在しないか, またはクラスタが削除できなかった。

`rm_permission`

- 名前

`rm_permission`: (サブ) クラスタの許可を取り消すプログラム。

- 書式

```
rm_permission username clustername administrator|user
```

- 説明

`rm_permission` は (サブ) クラスタ上の許可を取り消すシェルコマンドです。使用または管理の許可を取り消します。許可を取り消す (サブ) クラスタの親クラスタの管理者権限が必要です。 `username` 許可を取り消すユーザの名前, `clustername` 許可を取り消すサブクラスタの名前を意味します。 `administrator|user` は取り消す許可を示すキーワードで, `user` は使用許可, `administrator` は管理許可を意味します。

- EXIT CODES

3: 誤った使用

10: ユーザが存在しないか, 許可が取り消せなかった。

rn_cluster

- 名前

rn_cluster: (サブ) クラスタをリネームするプログラム.

- 書式

```
rn_cluster old_cluster_name new_cluster_name
```

- 説明

rn_cluster は (サブ) クラスタの名前を変更するシェルコマンドです. このコマンドを実行するにはリネームするクラスタの管理者権限が必要です. old_cluster_name はリネームするクラスタの現在の名前, new_cluster_name は新しい名前を意味します.

- EXIT CODES 3: 誤った使用

10: ユーザが存在しないか, クラスタのリネームができなかった.

test_node

- 名前

test_node: ステータスをチェックするためにノードをrshし, ALINKA LCM データベースを更新します.

- 書式

```
test_node [-hv] hostname date status
```

- 説明

test_node コマンドはステータスを更新し, 負荷ログファイルを更新するためにスレーブノードをに textttrsh するシェルコマンドです. ステータブルノードとは人のアクションがない限り状態が変わらないノードを意味します. 従ってRunning, Halted, Down, Boot_failed はステータブルノードであり, 残りは全てアンステータブルと考えられます. test_node コマンドはgt_load コマンドによってコールされます.

- EXIT CODES

0: ステータブル

1: 使用

2: アンステータブル

□ PHP3

delete_cluster.php3

- 書式

```
delete_cluster dbname user clustername
```

- 説明

ユーザが親クラスタの管理者の場合, サブクラスタをデータベースから削除します.

delete_ipaddr.php3

- 書式

`delete_ipaddr dbname`

- 説明

フリー IP アドレステーブルから、利用できる IP アドレスを選択します。

delete_node.php3

- 書式

`delete_node dbname hostname`

- 説明

クラスタデータベースからノードを削除します。

delete_permission.php3

- 書式

`delete_permission dbname lcm_user username clustername permission`

- 説明

まず user が管理者権限を持っているかどうかをチェックし、権限があれば username で指定されたユーザの許可を削除します。

insert_cluster.php3

- 書式

`insert_cluster database lcm_user father son`

- 説明

father で指定されるクラスタに son で指定される新しいサブクラスタを作成します。

insert_ipaddr.php3

- 書式

`insert_ipaddr db_name ip_addr`

- 説明

ip_addr で指定する IP アドレスをフリー IP アドレステーブルに追加します。

insert_node.php3

- 書式

`insert_node dbname hostname ipaddress macaddress clustername hdstatus`

- 説明

ALINKA LCM データベースに新しいノードを追加します。

insert_permission.php3

- 書式

```
insert_permission dbname lcm_user username clustername permission
```

- 説明

`lcm_user` は `clustername` で指定されたクラスタに対する管理者権限が必要で, `username` で指定されたユーザに使用許可を与えます.

insert_task.php3

- 書式

```
insert_task dbname clustername lcm_user id file
```

- 説明

ALINKA LCM データベースに新しいタスクを追加します. `file` はライブラリ (MPI または PVI), プログラムディレクトリ (作業ディレクトリ), 優先度 (0 = 高い ~ 19 = 低い), スタートする日時の順に記述されたファイルです.

move_node.php3

- 書式

```
move_node dbname lcm_user hostname clustername
```

- 説明

ALINKA LCM データベースの `hostname` で指定されたノードを `clustername` で指定されたクラスタに移動します. `lcm_user` は現在のクラスタと移動先のクラスタの両方に対する管理者権限が必要です.

office.php3

- 書式

```
office
```

- 説明

パラレルタスクが開始されるノードをリターンします. 最後にリターンされた名前のノードでマスタが実行するパラレルコマンドを `rsh` します.

populate_tb_ipaddr.php3

- 書式

```
populate_tb_ipaddr db_name class network
```

- 説明

スレーブで使われる全ての IP アドレスをデータベースに挿入します.

populate_tb_node_type.php3

- **書式**

```
populate_tb_node_type
```

- **説明**

クラスタで使われている CPU にノードタイプを与えます。ノードタイプでのモニタが可能になります。

rename_cluster.php3

- **書式**

```
rename_cluster database lcm_user src dst
```

- **説明**

src で指定するクラスタをdst で指定する名前に変更します。lcm_user はsrc の親クラスタに対する管理者権限が必要です。

update_node_type.php3

- **書式**

```
update_node_type db hostname processors modelname
```

- **説明**

データベースのノードタイプを変更します。新しいノードがブートした後、使用します。

☐ **sbin****alinka-lcm_install**

- **名前**

alinka-lcm_install: ALINKA LCM を設定します。

- **書式**

```
alinka-lcm_install
```

- **説明**

ALINKA LCM を設定するインタラクティブなシェルコマンドです。

- **戻り値エラーメッセージ**

- **EXIT CODE**

1: アドミニストレータでない。

alinka-lcm_uninstall

- **名前**

alinka-lcm_uninstall: /etc 内の alinka-lcm_install でインストールされた lcm.conf を削除します。

- **書式**

alinka-lcm_uninstall

- **説明**

/etc 内の alinka-lcm_install で作られた全ての項目を削除します。例えばこのスクリプトで /etc/hosts 内の ALINKA LCM に関する項目は削除されます。削除するためにこれらのファイルで, #-- Lcm begin と #-- Lcm end を探し, この間にあるものを削除します。

cf_master

- **名前**

cf_master: マスターサーバで ALINKA LCM を設定するために使用します。

- **書式**

cf_master

- **説明**

cf_master はマスターサーバを設定します。/etc/host, /etc/host.equiv, /etc/sudoers に新しい項目を作成します。/tftpboot が存在しない場合には, /tftpboot から \$LCM_TFTPBOOT までのシンボリックリンクを作成します。/etc/crontab と /etc/inetd に対する項目を作成します。inetd metaserver コンフィギュレーションファイルをリロードします。LCM_LOG と LCM_TMP ディレクトリを作成します。そしてディストリビューション固有のコンフィギュレーションファイルを実行します。このプログラムはそれ自身を実行してはいけません。alinka-lcm_install から呼び出され実行します。

- **EXIT CODES**

1: エラー

3: エラー: /tftpboot はディレクトリで, \$LCM_TFTPBOOT ではありません。

chfreq

- **名前**

chfreq: getload と test の頻度を変更します。

- **書式**

chfreq [-hv] [-l] loadfreq [-t] testfreq [-g] getload

- **説明**

chfreq コマンドは lcm.conf の LCM_LOAD_FREQ, LCM_TEST_FREQ と LCM_LOAD の値を変更し, それに従って /etc/crontab を変更します。これは lcm.conf を編集し, gt_load を手動で実行することで同様のことを行うことができます。

- **戻り値**

blah

- **EXIT CODE**

- 1: 使用
- 2: アンステーブル

gt_cpuinfo

- **名前**

gt_cpuinfo: ノードの CPU に関する情報を得るプログラムです.

- **書式**

gt_cpuinfo hostname

- **説明**

gt_cpuinfo は新しいノードの作成時に自動的にコールされ、そのノードの CPU に関する情報を入手するシェルコマンドです. これで得られた情報は ALINKA LCM データベースからアクセスすることができます.

- **EXIT CODES**

- 2: 管理者でない.
- 10: ノードから応答がない.
- 11: サポートしていないモデル.
- 12: データベースの更新ができない

local_hd_setup

- **名前**

local_hd_setup: ハードディスクを設定するために各スレーブノードで実行されるスクリプトです.

- **書式**

local_hd_setup --erase_hard_drive|--second_stage

- **説明**

このスクリプトはsudo 権限 (lcm 管理者) を持っているユーザがスレーブ上で (frsh 経由) 実行します. このスクリプトはnode_hd_setup から呼び出し実行するもので、手動でこれを実行してはいけません. erase_hard_drive オプションでスレーブのハードディスクのパーティションを削除し、新しいパーティションを作成し、パーティションのフォーマットを行います. second_stage オプションは/tmp/nodefiles.tgz の内容を/temp にコピーし、/temp をアンマウントし、/mnt に/dev/alinkahda1(tempo だったもの) をマウントします.

- **EXIT CODE**

- 1: 使用
- 10: tar が第二段階のインストールに失敗しました.
- 11: プライマリパーティションがアンマウントません.

mk_db

- **名前**

mk_db: /etc/lcm.conf を参照し ALINKA LCM を実行するのに必要なデータベース構築するプログラム.

- **書式**

mk_db

- **説明**

mk_db は ALINKA LCM が必要とする PostgreSQL データベースを作成します. LCM_ADMINISTRATOR と LCM_DB パラメータを必要とします. ALINKA LCM が必要とする全てのテーブルを作成し, ノードなし, サブクラスタなしのルートクラスタの初期化をはじめます. このコマンドを実行するには postmaster デーモンが稼動している必要があります.

- **EXIT CODES**

- 10: ALINKA LCM 管理者が \$LCM_ADMINISTRATOR と合致しません.
- 11: LCM_ADMINISTRATOR が Linux のユーザではありません.
- 12: LCM_ADMINISTRATOR ユーザの作成ができませんでした.
- 13: データベースの作成ができませんでした.
- 20: sq_cluster が作成できませんでした.
- 21: tb_cluster が作成できませんでした.
- 22: クラスタルートのためのフリーな ID がありません.
- 23: クラスタルートの挿入ができませんでした.
- 30: sq_node が作成できませんでした.
- 31: tb_node が作成できませんでした.
- 40: tb_ipaddr が作成できませんでした.
- 41: IP アドレスの割り当てができませんでした.
- 50: sq_node_type が作成できませんでした.
- 51: tb_node_type が作成できなかったか, 不明なタイプの挿入ができませんでした.
- 52: ノードタイプの割り当てができません.
- 60: sq_user が作成できませんでした.
- 61: tb_user が作成できませんでした.
- 62: ALINKA LCM 管理者用のフリーな ID が見つかりません.
- 63: ALINKA LCM 管理者をユーザとして挿入することができませんでした.
- 70: tb_permission が作成できませんでした.
- 71: ALINKA LCM 管理者にアドミニストレーション許可を与えることができませんでした.
- 80: sq_task が作成できませんでした.
- 81: tb_task が作成できませんでした.

mk_node

- **名前**

mk_node: 新しいノードを作成するプログラム.

- **書式**

mk_node [-m macaddr] [clustername]

- **説明**

mk_node はノードの作成に関する作業をすべて行います. 1 つの新しいノードがブートするのを

待ち, `-m` なしで実行するとスレーブノードのセットアップを行います. クラスタ名が指定された時, ノードは直ちにその (サブ) クラスタの所属となりますが, 指定されない場合にはルートクラスタがデフォルトとして使用されます.

- **オプション**

`-m` は新しいノードの MAC アドレスを指定するためのもので, データベース内に存在するかどうかを事前にチェックすることができます. これは `mk_node_daemon` で使用されます.

- EXIT CODES 3: 誤った使用

mk_node_daemon

- **名前**

`mk_node_daemon`: クラスタに対するノード作成を開始または終了するデーモン.

- **書式**

`mk_node_daemon start|stop`

- **説明**

ALINKA LCM に対するノード作成を可能にするデーモンの起動または停止を行います. 起動している時は, `bootp` リクエストを出して起動した新しいコンピュータを自動的にクラスタに追加します. `/usr/local/alinka-lcm/tmp` のデーモンのPID と `bootp` への応答を要求したノードのMAC アドレスを監視することができます.

- EXIT CODES

1: 任意

mk_template

- **名前**

`mk_template`: 選択したディストリビューションに依存するテンプレートを作成します.

- **書式**

`mk_template`

- **説明**

Linux の基本的な実行環境をす作成します. このスクリプトはスレーブノードで使用するテンプレートディレクトリを作成しますが, マスターサーバのあるファイルにハードリンクをはることにより, 容量を削減します. このコマンドは `alinka-lcm_install` コマンドにより一回だけ実行され, テンプレートディレクトリを作成します.

- EXIT CODES

1: エラー

node_hd_setup

- **名前**

`node_hd_setup`: スレーブノードに対するハードディスクの設定

- 書式

`node_hd_setup`

- 説明

このスクリプトは全てのスレーブノードが追加された後、マスターで実行してください。もしスレーブノードが自分のハードディスクを使わなければならない場合 (`/etc/lcm.conf` で設定)、このプログラムがその設定を行います。このプログラムが扱うノードは `hd_status` が “Waiting” のものだけです。もしあるスレーブノードが既にコピー中、フォーマット中、或いは “Waiting” のスレーブノードがない場合は、このスクリプトとは終了します。まずこのスクリプトは存在しない場合 `$FILES_IMAGE_LOCATION/nodefiles.tg` を `/bin /lib /dev /sbin` と `/usr` から作成します。そして全てのスレーブノードのブートとクラスタがノーマルな状態 (テスト状態ではない) になるのを待ちます。次に全ての稼動しているノードを選択し、`hd_status` を “Formating” にセットします。次にスレーブノードに `rsh` し、`local_hd_setup` を立ち上げ、スレーブノードのハードディスクに対して `FDISK` を実行し、その後フォーマットを行います。これが終わるとノードの `hd_status` は “Copying” にかわります。ローカルファイルは `/dev/alinkahda1` (通常は `/dev/hda1`) に置かれます。このパーティションはフォーマットされ、`/temp` にマウントされます。全てのノードで “Formating” ステータス終了すれば (つまりコピーステータス)、`/bin /lib /dev /sbin` と `/usr` で必要なファイルをまとめた `tar.gz` ファイルをサーバから最初のスレーブノードにコピーし、次にこれら 2 つのノードから 3,4 番目のノードへ、更にこれら 4 つのノードから 5,6,7,8 番目のノードへとコピーされ、この動作が全てのノードの `/tmp` に `tar.gz` ファイルがコピーされるまで続けられます。以上の操作が完了すると、`tar.gz` ファイルは `/temp` ディレクトリで解凍され、`/dev/alinkahda1` が `/mnt` としてマウントされます。これは `local_hd_setup` のステップ 2 として行われ、`hd_status` は “Uptodate” に変わります。

- EXIT CODES

- 1: `lcm.conf` が読めないか、フォーマットまたはコピー状態のノードがある。
- 2: 既に実行中。
- 12: インストールの第 2 段階が未完了。
- 15: ディスクのパーティション作成またはフォーマットで問題が発生。

`rm_node`

- 名前

`rm_node`: クラスタからノードを削除する。

- 書式

`rm_node hostname`

- 説明

クラスタから `hostname` で指定されたノードを削除します。コンフィギュレーションファイル (ディストリビューションによって異なります) を更新し、ALINKA LCM データベースを更新します。

- EXIT CODES

- 3: 誤った使用

`crontab_mode`

- **名前**

`crontab_mode`: チェックモードからテストまたはノーマルモードに変更する。

- **書式**

`contab_mode [-hv] -t|-n`

- **説明**

ノーマルモードでは`gt_load` に対するコールを余り頻繁には行いません。 `gt_load` はスレーブノードからの情報を入手し、負荷ファイルを更新します。 `lcm.conf` の `LCM_LOAD_FREQ` のデフォルト値を手動または`alinka-lcm_install` ステージで変更していなければ 15 分毎に 1 回実行します。 テスト モードは何かおかしいこと (“strange”) が起こった時にセットされます。 おかしいこととはアンステーブルな状態を意味し、人間の介入なしに、自動的にこのモードに変わります (データベース情報をご参照ください)。 テストモードの状態にあるノードはアンステーブルな状態でもあります。 何故なら次のレベルのチェックで、実行状態、テスト状態、フェイル状態のいずれになるか解らないからです。 従って`Running`, `Halted`, `Down`, `Boot_failed` のいずれかの状態がステーブル状態に相当します。

“ALINKA RAISIN” ユーザの方は自動的に管理されますので、上記の内容は無視してください。

- **EXIT CODE**

1: `lcm.conf` is unreachable

`sync_perm`

- **名前**

`sync_perm`: ALINKA LCM データベースと `/etc/passwd` との間でのユーザに対する許可の同期を行います。

- **書式**

`sync_perm [-hv] -c clustername | -i cluster_id | -n node_hostname`

- **説明**

あるノードの `/etc/passwd` ファイルに問題がある場合に、このコマンドを実行します。 このコマンドを実行することで正しい `/etc/passwd` ファイルが作成され、ユーザは許可されたクラスタにのみアクセスできるようになります。 もしそうしなければユーザがクラスタあるいはサブクラスタ内の本来認められていないコンピュータにもログインすることが可能になるかも知れません。

このスクリプトは`mk_node` で自動的に実行されます。 スレーブノードの移動或いはユーザの許可の変更を行った時には手動で実行してください。

- **オプション (ALINKA RAISIN ユーザは上記の内容を無視してください)**

`-h`: ヘルプ

`-v`: 詳細表示モード

`-c`: `clustername` で指定されたクラスタ内のノードの許可の同期をとる。

`-i`: `clusterid` で指定されたクラスタ内のノードの許可の同期をとる。

`-n`: `hostname` で指定されたノードの許可のみ同期をとる。

- **EXIT CODES**

1: 使用

2: `lcm.conf` が読めないか, `clustername` で指定されたクラスタが存在しない。

□ 付録

ALINKA LCM データベースの内容 — 付録 A

- `sq_`で始まるアイテムはシーケンスです.
- `tb_`で始まるアイテムはテーブルです.
- `tb_cluster`
クラスタとサブクラスタのテーブルです.
- `id`
クラスタの ID.
- `id_cluster`
親クラスタの ID.
- `id_node`
未使用.
- `clustername`
このクラスタの名前を示すストリング.
- `tb_ipaddr`
新しいノードに与えることができる全ての IP アドレスのテーブル.
- `tb_node`
クラスタに含まれるノードのテーブル.
- `id`
ノードの ID.
- `id_cluster`
親クラスタの id.
- `id_node_type`
ノードのタイプの id(下記の`tb_node_type`を参照).
- `hostname`
ノードのホスト名.
- `ipaddr`
IP アドレス.
- `macaddr`
MAC アドレス.
- `status`
ステータス (詳細は付録 B を参照).
- `hd_status`
ハードディスクのステータス (詳細は付録 B を参照)

- `tb_node_type`
ノードのタイプのテーブル.
- `id`
ノードのタイプを扱う数字.
- `processors`
CPU.
- `modelname`
CPU の種類 (i586, i686, ...).
- `tb_permission`. 許可テーブル.
- `id_user`
関連するユーザのデータベースid.
- `id_cluster`
関連するクラスタのid.
- `permission`
与えられた許可 (使用または管理).
- `tb_task`
タスクのテーブル.
- `id`
タスクの ID.
- `id_cluster`
実行するクラスタのid.
- `id_user`
実行するユーザ
- `status`
状態 (waiting, running, done).
- `library`
(pvm または mpi) と一緒に使われる並列ライブラリ.
- `program`
実行するプログラムの名前.
- `directory`
作業用ディレクトリ.
- `nice`
プライオリティレベルレベル: 0=高, 39=低.
- `at`
与えられた時間.
- `ts_begin`
開始している場合は実行を開始した時間.

- `ts_end`
終了している場合は実行を停止した時間.
- `tb_user`
ユーザのテーブル.
- `id`
ユーザの ID.
- `username`
ログインしたユーザ名.

ノードの状態 – 付録 B

付録 A からお解りのようにノードには 2 つの状態があります. つまり実行状態 (running status) とハードディスク状態 (hard disk status) です.

実行状態とは:

- Running (ステータブル) ノードは OK です.
- Testing (アンステータブル) ノードは直近の `rsh` リクエストに 응답しませんでした.
- Testing₂ (アンステータブル) ノードは前の前の `rsh` に 응답しませんでした.
- Down (ステータブル) ノードは `rsh` リクエストに一切 응답しませんでした.
- Rebooting (アンステータブル) ノードはリブート中です.
- Booting. (アンステータブル) ノードはブート中です.
- Booting.. (アンステータブル) ノードが未だ `rsh` リクエストに 응답しません.
- Booting... (アンステータブル) ノードが未だ `rsh` リクエストに 응답しません.
- Boot_failed (ステータブル) ノードはブートできないようです.
- ToHalt (アンステータブル) Halt するように指示しましたが, 未だ `rsh` リクエストに 응답しています. 暫くお待ちください.
- Halted (ステータブル) ノードは Halt しました.

ハードディスク状態とは

- None – ローカルディスクを使用しない (スレーブにはハードディスクは不要).
- Waiting – ノードはハードディスクの設定を待っています.
- Formatting – ノードに `local_hd_setup - erase_hard_drive` コマンドが与えられました.
- Copying – ノードはフォーマットを終わり, `tgz` ファイルの `rcp` を待っています.
- Ready_{2nd} – `tgz` ファイルは `/tmp` に格納され, `local_hd_setup` の第 2 段階の準備ができました.
- Uptodate – 第 2 段階が終了し, ノードは自分のハードディスクを使用しています.

/etc/lcm.conf – 付録 C

/etc/lcm.conf ファイルには次のフィールドがあります。

- LCM_VERSION – バージョン.
- LCM_DISTRIBUTION – マスターサーバで実行中のディストリビューション.
- LCM_ADMINISTRATOR – ルート クラスタ管理者のユーザ名.
- LCM_ROOT – alinka-lcm ディレクトリの位置.
- LCM_BIN – alinka-lcm/bin ディレクトリの位置.
- LCM_SBIN – alinka--lcm/sbin ディレクトリの位置.
- LCM_PHP3 – alinka--lcm/php3 ディレクトリの位置.
- LCM_LOG – alinka--lcm/log ディレクトリの位置.
- LCM_TMP – alinka--lcm/tmp ディレクトリの位置.
- LCM_TFTPBOOT – tftpboot ディレクトリの位置.
- LCM_TEMPLATE – テンプレートのある位置 (tftpboot と同じパーティションになければなりません).
- LCM_PHP3CGI – php3 cgi のある位置. PostgreSQL データベースへ接続できなければなりません.
- LCM_DB – PostgreSQL データベース. ALINKA LCM ユーザの名前.
- LCM_INTERNET_IF – 外部ネットワークと接続するネットワークカードの名前.
- LCM_LOCALNET_IF – クラスタへ接続するネットワークカードの名前.
- LCM_CLASS – クラスタの IP クラス. C, B, A のいずれか.
- LCM_NETWORK – クラスタのネットワークアドレス.
- LCM_NETMASK – クラスタのネットマスク.
- LCM_BROADCAST – クラスタのブロードキャストアドレス.
- LCM_MASTER – マスターサーバの IP アドレス.
- LCM_MASTERNAME – マスターサーバの名前.
- LCM_NODENAME – ノードの “root” 名.
- LCM_DOMAINNAME – クラスタのドメイン名 (この名前はローカルなもので, DNS に追加する必要はありません).
- USE_NODE_HARD_DISKS – スレーブでのハードディスク使用の有無.
- FILES_IMAGE_LOCATION – マスタの /bin, /sbin, /lib, /dev と /usr を含む big.tar.gz ファイルのパス.
- NODE_SLASH – スレーブの / のサイズ. Fdisk format を使用する.

- **NODE_SWAP** – スレーブのスワップのサイズ. Fdisk format を使用する.
- **NODE_TEMP** – スレーブの /tmp のサイズ. Fdisk format を使用する.
- **RAISIN_PATH** – alinka-raisin ディレクトリの位置. ALINKA RAISIN を使用するのためにのみ必要.
- **PVMROOT** – PVM_ROOT ディレクトリ. PVM を使用するのために必要.
- **LCM_TEST_FREQ** – ノードがステータスモードでない時, gt_load コマンドを実行する頻度を分で示す.
- **LCM_LOAD_FREQ** – ノードがステータスモードの時, gt_load を実行する頻度を分で示す.
- **LCM_LOAD** – ログファイルを記録する間隔 (1/5/15) 分.
- **RAISIN_KEY** – ALINKA RAISIN に対するカスタマー・キー. ALINKA RAISIN を使用するのためにだけ必要.

■ その他, 有用なソフトウェア

ジョブ・スケジューリングソフト

- **LSF** (<http://www.platform.com/>) 商用のスケジューリングソフト
- **PBS** (<http://pbs.mrj.com/>)
- **Maui Scheduler**(<http://www.mhpc.edu/maui/>)
- **JOB manager** (<http://bond.imm.dtu.dk/jobd/>)

パフォーマンスモニタ

- **Beowulf Watch** (<http://www.sci.usq.edu.au/staff/jacek/bWatch/>)

Host Name	Num Users	TTime	1 min Load	5 min Load	15 min Load	Total Mem	Free Mem	Shared Mem	Buffers	Cache	Total Swap	Free Swap
node1	4 users	4:17	0.00	0.07	0.15	377 Mb	6476 Kb	53 Mb	7 Mb	337 Mb	762 Mb	762 Mb
node3	0 users	4:17	4.15	3.85	2.47	61 Mb	38288 Kb	11 Mb	0 Mb	19 Mb	0 Mb	0 Mb
node4	0 users	4:17	1.08	0.94	0.56	61 Mb	40376 Kb	8 Mb	0 Mb	18 Mb	0 Mb	0 Mb
node5	0 users	4:17	1.02	0.98	0.67	61 Mb	32880 Kb	9 Mb	0 Mb	20 Mb	0 Mb	0 Mb
node6	0 users	4:17	1.08	0.92	0.52	61 Mb	40544 Kb	7 Mb	0 Mb	18 Mb	0 Mb	0 Mb
node7	0 users	4:17	0.99	0.89	0.51	61 Mb	40552 Kb	7 Mb	0 Mb	18 Mb	0 Mb	0 Mb
node8	0 users	4:18	0.99	0.89	0.51	61 Mb	41532 Kb	7 Mb	0 Mb	17 Mb	0 Mb	0 Mb
node9	0 users	4:18	3.15	2.75	1.60	61 Mb	41188 Kb	7 Mb	0 Mb	17 Mb	0 Mb	0 Mb
node10	0 users	4:18	2.07	1.82	1.06	61 Mb	41552 Kb	7 Mb	0 Mb	17 Mb	0 Mb	0 Mb
node11	0 users	4:18	0.99	0.90	0.51	62 Mb	27040 Kb	8 Mb	0 Mb	32 Mb	0 Mb	0 Mb
node12	0 users	4:18	3.07	2.74	1.61	62 Mb	44032 Kb	6 Mb	0 Mb	16 Mb	0 Mb	0 Mb
node13	0 users	4:19	14.27	11.64	6.22	62 Mb	1636 Kb	3 Mb	48 Mb	1 Mb	0 Mb	0 Mb

Refresh Exit

- **lperfex** (<http://www.osc.edu/~troy/lperfex/>)
Cray のhpm や SGI のprefex と同等の機能を有する.

クラスタ構築支援システム

- FAI (<http://www.informatik.uni-koeln.de/fai/>) Debian ベースの PC Cluster 構築支援システム
- LUI (<http://oss.software.ibm.com/developerworks/opensource/linux/projects/loi/>) RPM ベースの PC Cluster 構築支援システム

並列処理入門

同志社大学 工学部 知識工学科

三木 光範

mmiki@mail.doshisha.ac.jp

■ はじめに

並列処理 (Parallel Processing) とは、一般的な意味では「仕事を並列に進めること」である。「並列」だけの意味は「並び連なること」(大辞林)であるが、これでは直列との違いがわからない。一方、「並列処理」は専門用語で、「コンピュータで、一連の処理を複数台の処理装置で同時に並行して行うこと」(大辞林)という意味である。並列処理と同じような言葉に **並行処理** がある。複数の CPU を使って複数のプログラムを(真に)同時に実行することを並列 (Parallel) 実行、一つの CPU で擬似的に実行することを疑似並列 (Pseudo-parallel) とか並行 (Concurrent) 実行とよぶ。また、並行という言葉は、真の並列と疑似並列の両方を総括して示す場合にも使われる¹⁾。

一方、マルチプロセスとは複数のプログラムが同一のコンピュータを同時に活用できる機能のことである。通常のコンピュータには CPU は一つしかないので、二つ以上のプログラムを同時に実行することはできない。マルチプロセスを提供するコンピュータでは 1/60 秒程度の短い間隔で複数のプログラムを順に実行し、同時に実行されているかのような錯覚をユーザに与えているのである。ただし、こうした環境下でもあるプログラムが入力待ちなどがあるので、システム全体の処理効率が期待できる¹⁾。

ここでは、並列処理、すなわちひとまとまりの処理を複数の CPU を用いて行う方法について述べる。並列処理の背景には並列計算機や PC クラスタなどのハードウェアの発達と並列化コンパイラやプロセッサ間通信のためのドライバなどソフトウェアの発達がある。このため、並列処理を考える場合には、「そもそも仕事を並列に進めるとはどういうことか」という観点と「並列処理のためのハードウェアとソフトウェアはどうなっているのか」という観点が不可欠である。本講習会では特に PC クラスタに焦点を当て、こうしたことが初心者にも分かるように構成されている。本稿ではその最初の導入として、並列処理のための概要を述べるとともに、「そもそも仕事を並列に進めるとはどういうことか」という点について考える。

■ 超並列計算機の発達

1997 年 5 月 11 日、ロシア人のチェス世界チャンピオン、ガルリ・カスパロフは IBM 社のコンピュータ "Deep Blue" とシリーズ第 6 戦を行い、2 勝 1 敗 3 引き分けで Deep Blue が勝った²⁾。これは、チェスという「頭を使う」ゲームにおいてコンピュータが人間のチャンピオンを負かしたという画期

Rank	Manufacturer	Computer	Rmax	Installation Site	Country	Year	Area of Installation	# Proc	Rpeak	Nmax	N1/2
1	Intel	ASCI Red	2379.6	Sandia National Labs Albuquerque	USA	1999	Research	9632	3207	362880	75400
2	IBM	ASCI Blue-Pacific SST, IBM SP 604e	2144	Lawrence Livermore National Laboratory Livermore	USA	1999	Research Energy	5808	3868	431344	.
3	SGI	ASCI Blue Mountain	1608	Los Alamos National Laboratory Los Alamos	USA	1998	Research	6144	3072	374400	138000
4	IBM	SP Power3 375 MHz	1417	IBM/Naval Oceanographic Office (NAVOCEANO) Poughkeepsie	USA	2000	Vendor Aerospace	1336	2004	374000	.
5	Hitachi	SR8000-F1/112	1035	Leibniz Rechenzentrum Muenchen	Germany	2000	Academic	112	1344	120000	15160
6	Hitachi	SR8000-F1/100	917.2	High Energy Accelerator Research Organization /KEK Tsukuba	Japan	2000	Research	100	1200	115000	15000
7	Cray Inc.	T3E1200	891.5	Government	USA	1998	Classified	1084	1300.8	259200	26400
8	Cray Inc.	T3E1200	891.5	US Army HPC Research Center at NCS Minneapolis	USA	2000	Research	1084	1300.8	259200	26400
9	Hitachi	SR8000/128	873.6	University of Tokyo Tokyo	Japan	1999	Academic	128	1024	120000	16000
10	Cray Inc.	T3E900	815.1	Government	USA	1997	Classified	1324	1191.6	134400	26880
11	IBM	SP Power3 375 MHz	723.4	Oak Ridge National Laboratory Oak Ridge	USA	2000	Research	704	1056	187000	37500
12	SGI	ORIGIN 2000 250 MHz	690.9	Los Alamos National Laboratory /ACL Los Alamos	USA	1999	Research	2048	1024	229248	80640
13	Cray Inc.	T3E900	675.7	Naval Oceanographic Office (NAVOCEANO) Bay Saint Louis	USA	1999	Research Weather	1084	975.6	.	.
14	Cray Inc.	T3E1200	671.2	CSAR at the University of Manchester Manchester	UK	2000	Academic	812	974.4	.	.
15	Cray Inc.	T3E1200	671.2	Deutscher Wetterdienst Offenbach	Germany	1999	Research Weather	812	974.4	.	.

図 6.1: 2000 年 6 月時点での高性能コンピュータの世界ランキング (15 位まで) ³⁾

的な出来事である。この Deep Blue は過去 100 年の序盤戦と、数十億の終盤戦シナリオのデータベースを搭載し、256 個のプロセッサを持つ超並列スーパーコンピュータであり、1 秒間に数億通りの局面を読むことのできる能力を持っており、気の遠くなるような膨大な計算を行ってチャンピオンに対抗した。いわばしらみつぶしに近い力業で勝利した。

このようなコンピュータの力業は近年ますます重要になってきている。科学・工学におけるシミュレーションは取り扱う現象のレベルがミクロのスケールに近づくようになり、できるだけ普遍的で、近似の少ない、原理的な法則を基にコンピュータの力業で複雑な現象を解析するアプローチが注目を集めている。これは明らかにコンピュータの驚異的な発達に依存している。力業が行える驚異的な力が得られるようになったからであり、その驚異的な力が半導体技術の高度化とプロセッサの並列化という二つの技術によって支えられている。

プロセッサ単独の計算速度は次第に限界に近づいてきた。プロセッサの計算速度はクロック周波数に依存している。このクロック周波数が 1GHz になると周期は 1ns となり、電気信号はその間に 30cm しか進まず、集積回路の大きさや配線基盤の寸法を考えるとこれ以上クロック周波数を大幅に上げることは原理的に困難になってくる。一方、1ns より早いスイッチング速度を持つ半導体材料についても極めて限られてくる。材料の面から言っても限界は近い。さらに、いくらプロセッサが高速に動作してもメモリーからのデータの供給がなければ必要な演算を行うことはできず、メモリーのアクセス速度の高速化が困難な現状では単独のプロセッサの演算速度はかなり限界に近づいたと言える。こうして、大規模シミュレーションなどの力業を行うコンピュータは必然的に超並列計算機となり、並列処理技術の高度化が次の時代を開く。

■ 計算の高速化と並列処理

計算機の高速化の要求には終わりが無い。この進歩を可能にする計算機技術が演算パイプライン方式であり、さらにそれを多重にした多重パイプラインであり、こうした方式の計算機は **スーパーコン**

ピュータと呼ばれる。一方、CPU を極めて多数並列化した計算機が **パラレルコンピュータ** である。

現在の高性能計算機のランキングを図 6.1 に示す。これはドイツのマンハイム大学スーパーコンピュータセンターが毎年 2 回調査して発表しているもので、世界最大の能力を持つ 500 のコンピュータシステムのランキングであり、性能は LINPACK ベンチマーク³⁾ で比較している。この表で Rmax が達成された最高の LINPACK 性能であり、#Proc はプロセッサの数である。この表より、高性能のコンピュータシステムはすべて超並列システムであり、この流れは今後一層加速し、多くの高性能コンピュータは数千から数万のプロセッサを持たざるを得なくなることがわかる。

プロセッサの並列化はメモリーから考えても必然である。たとえば典型的な DRAM はメモリーアクセス時間が 80 ns であり、この時間で 1 word のデータを読むことができるから 1 秒間では 12.5 MW のデータが読める。この数字は典型的な SRAM になると **メモリーアクセス時間** が 10 ns であり、この場合は 100MW のデータが読める。ここで 1 word を 32bit、すなわち 4B (バイト) とすると、それぞれ 50 および 400 MB/s のデータ読みだし速度となる。これを **メモリーのバンド幅** という (図 6.2)。



図 6.2: メモリーのバンド幅

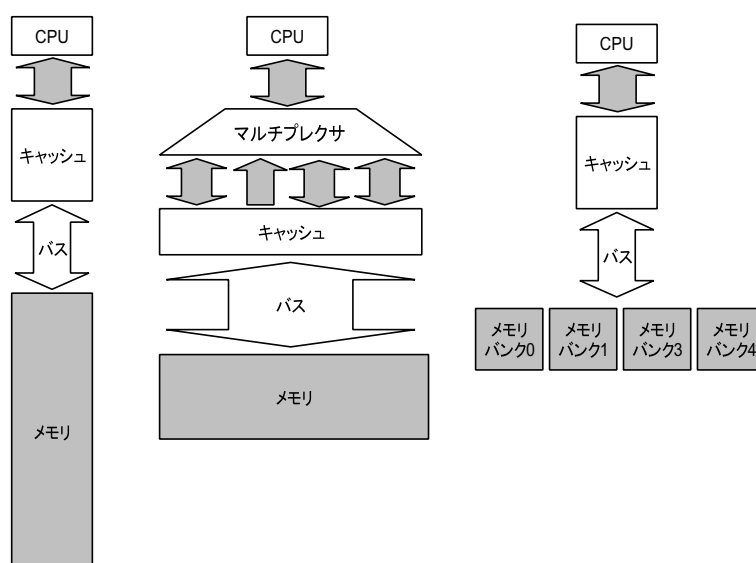
ここでたとえば単精度 (4 B/W) の演算を 20 GFLOPS の速度で行おうとすると 40 GW/s のデータ読みだしが必要になり、必要なメモリーバンド幅は 160 GB/s となる。これだけのバンド幅を確保するためには高速の SRAM を用いても 400 個のメモリーバンクから並列に読みだしを行わなければならない。もし、低速の DRAM を用いるなら 3200 個のメモリーバンクから並列に読みだしを行わなければならない。結局、どれだけ高速のメモリーを用いてもメモリーの並列化は不可避であり、この並列化に適合するアーキテクチャはプロセッサの並列化であると言える。図 6.3 にバンド幅を上げる方法を示す。

こうして、超高速演算を可能にするコンピュータは、光の速度の有限性の点からも、半導体材料の面からも、そしてメモリーバンド幅の点からも超並列となることがわかる。

■ 並列計算機の分類

並列計算機の分類には **Flynn の分類** が広く用いられている。これは命令の流れとデータの流れが単一か複数かで計算機を図 6.4 の 4 種類に分類するものである¹⁾。

並列計算機のアーキテクチャはまず次の二つに大別される。それらは **SIMD** (Single Instruction Multiple Data) と **MIMD** (Multiple Instruction Multiple Data) である。前者では多くのプロセッサが局所的なメモリーを有し、すべてのプロセッサは完全な同期の下でコントローラーから送られる単一の命令を同時に実行する。そしてその結果が集約されて一つの結果となる。たとえば Thinking Machines 社の CM-2 では 2048 個の 32 ビット浮動小数点演算プロセッサが演算対象データを分割して同じ処理を行う負荷分散に用いられる。この方式は同期をとるためのオーバーヘッドがないこと、単純なプロセッサで良いのでプロセッサ数を多くできること、負荷分散に対する要求は多いので利用価値が高いこと、などの利点があるが、同期をとるためのハードウェアとブロードキャストおよび演算

図 6.3: バンド幅を上げる方法⁴⁾

結果の集約を効率化する仕組みが重要になること、そして多くの応用においては負荷を一様に分散させることが困難で、プロセッサのロードバランスが悪くなることなどが避けられないことなどの欠点がある。一方、後者では各プロセッサは異なった処理を行うことができる。すなわち、プロセッサごとにコントローラーが独立して配置されており、SIMD アーキテクチャよりも柔軟な利用が可能となる。

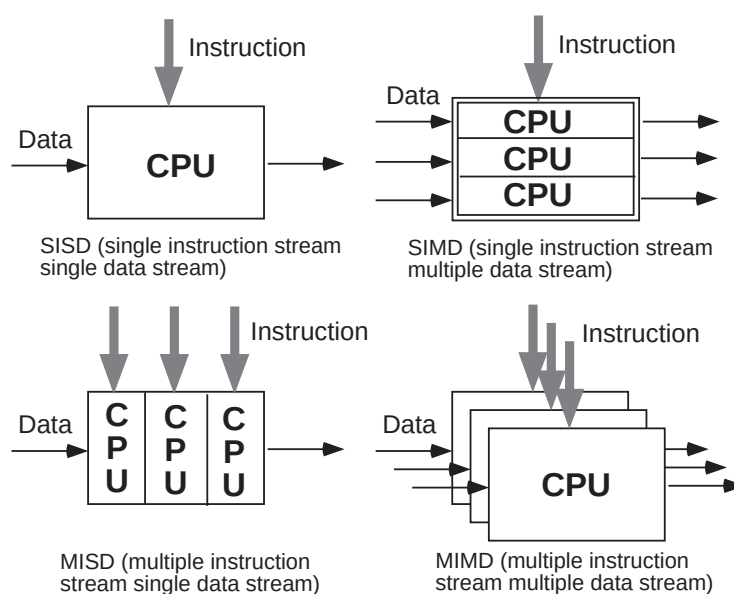


図 6.4: 並列計算機の種類

■ 並列計算機の構成方式

並列計算機全体のアーキテクチャを決定する最も大きな要因は、**プロセッサ間通信モデル**として何を採用するかである。すなわち、プロセッサ間でデータ授受をどのように行うか、および、メモリー

をどのように構成するかである。 **プロセッサ間通信** は以下の観点から分類する。

1) **共有メモリモデル**：論理的にメモリを共有する。複数のプロセッサが通常のアドレス指定で直接的に読み書きができる共有メモリが存在する。

(a) **集中共有メモリ** (centralized shared memory)：プロセッサとメモリが対称になることから **対称型マルチプロセッサ** (SMP : symmetric multiprocessor) とよぶ。

(b) **分散共有メモリ** (distributed shared memory)：ローカルメモリとして分散配置する。

2) **メッセージ交換モデル**：論理的にせよメモリは共有しない。 **メッセージの交換** という形でデータ授受を行う。 **NORA** (no remote access) モデルともいう。

(a) メッセージの送信と受信は同期するか、非同期か。

(b) メッセージの送受信においてブロック (封鎖：完了するまで処理を止める) するかどうか。

これらの分類を図示したものが図 6.5 である。なお、送受信のブロッキングはソフトウェアの問題であり、ハードウェアには現れない。このほかに、キャッシュの配置、コヒーレンスなどの観点から分類が可能であるが、専門的であるのでここでは割愛する。

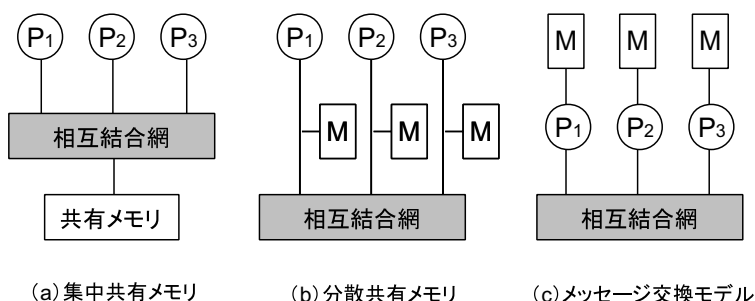


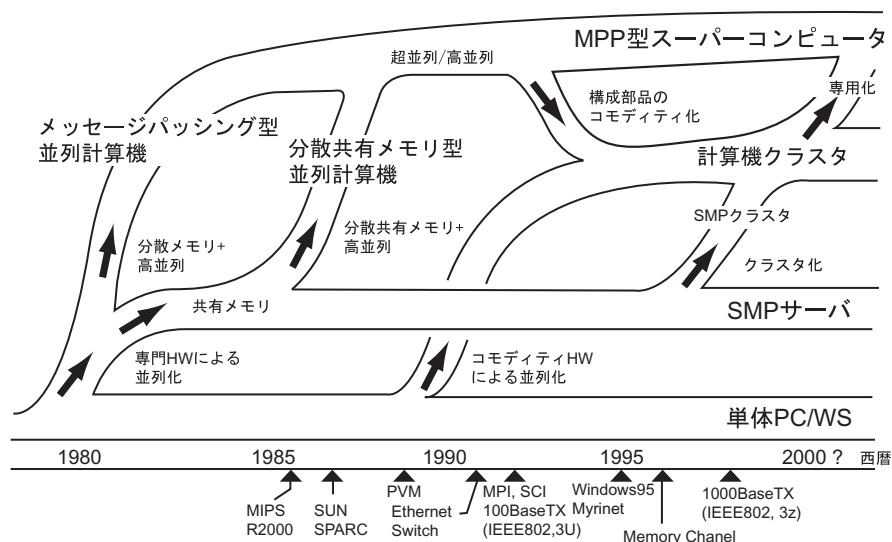
図 6.5: 並列計算機の構成法式

メッセージ交換モデルではネットワークのアーキテクチャや性能を除けば、慣用的なコンピュータをネットワークで結合したものと本質的には同じである。一方、共有型では対象問題を一括して一つのメモリに展開することが容易で使いやすいが、その反面、共有メモリのバンド幅は一定であるから、プロセッサ数はそれほど多くできない。普通は 4~20 ぐらいである。

1990 年前後には TCP/IP ベースのネットワークで接続された複数のコンピュータの集合体を、仮想的に 1 台の大きな並列計算機として見せるためのソフトウェアである **PVM** (Parallel Virtual Machine) が開発された。これにより「計算機クラスタ」と呼ぶべき物理的な計算機群が登場した。1995 年前後には、イーサネットスイッチや 100Mbit Ethernet などの技術が一般にも普及し、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。さらには計算機クラスタを指向した専用の高速ネットワークが登場し、一方では PC の高性能化と低廉化により、コストパフォーマンスの優れた計算機クラスタの実現が可能となった。図 6.6 は計算機クラスタへの道程を示したものである。

■ プロセッサ間ネットワーク (相互結合網)

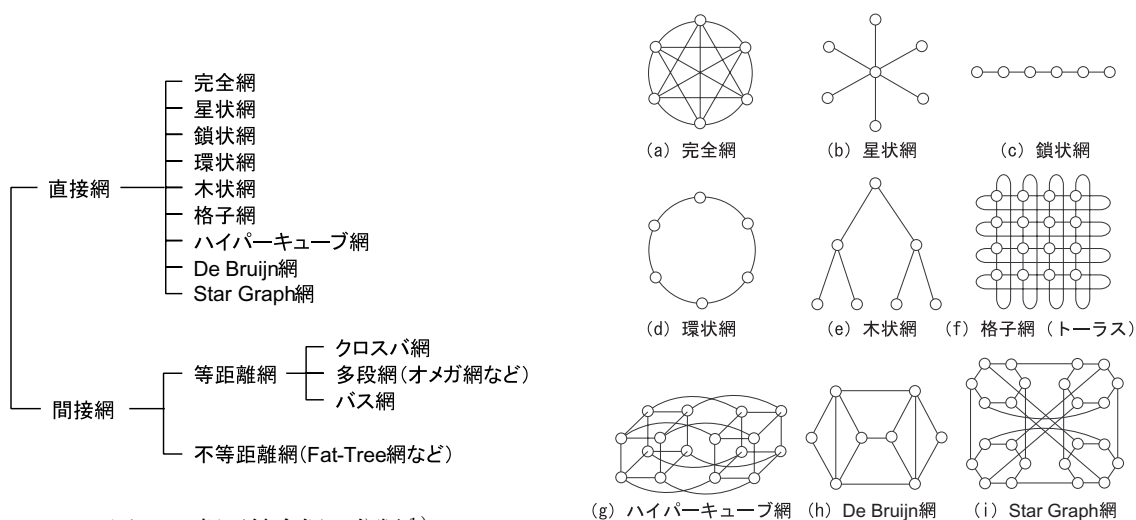
相互結合網 (interconnection network) は並列計算機を構成する複数のプロセッサやメモリを相互に結合し、その間における通信路を提供する。目標は **通信遅延** (**レイテンシ** : latency) を最小にし、スループット (throughput : 処理能力) を最大にすることである。

図 6.6: 計算機クラスタへの道程⁵⁾

相互結合網のトポロジ（topology：幾何学的位相形状）の分類を図 6.7 に示す。ここで直接網とは入出力ノード（相互結合網に結合されるプロセッサやメモリなど）間の結合がはじめてから固定されているものであり、間接網では結合が接続要求に応じて決まる。代表的な直接網を図 6.8 に示す。

上で述べた相互結合網は主として専用並列計算機のためのものである。すなわち、各相互結合網にはそれぞれの長所と短所があり、その専用並列計算機の目的に応じて相互結合網を設計することになる。一方、PC クラスタではコストパフォーマンスをあげるために汎用のネットワークハードウェアを用いるか、もしくはクラスタ用に設計された汎用のネットワークハードウェアを用いることになる。このため、相互結合網の種類は用いるネットワークハードウェアに依存する。

たとえば、TCP/IP ベースの通常のハブを用いた場合では相互結合網はバス型となり、二つのプロセッサが通信しているときには他のプロセッサは通信が行えない。一方、レベルの高いスイッチングハブを用いれば独立したプロセッサ間通信が同時に行えることになる。

図 6.7: 相互結合網の分類¹⁾図 6.8: 代表的な直接網¹⁾

■ 並列処理のプログラム

並列計算機を用いるには並列処理のためのプログラムを作成しなければならない。これは用いる並列計算機的环境によって異なったアプローチとなる。たとえば、専用並列計算機でもっとも一般的な方法は **HPF** (High-Performance Fortran) などの並列プログラミング言語を用いて配列に関わる演算を中心に並列化を行う。こうした言語を用いればコンパイラが自動的に並列処理のコードを生成するため、複数のプロセッサを使っているという意識なしに並列計算が行える。

しかしながら、こうした言語では大規模な配列やマトリックスに関わる演算では極めて有効であるが、それ以外の利用では柔軟性が少ない。そうした場合には通常の逐次処理プログラミング言語とプロセッサ間通信用の関数を用いてメッセージパッシング型のプログラムを利用者が作成しなければならない。これには PVM (Parallel Virtual Machine) や **MPI** (Message Passing Interface) を用いる方法、あるいは各社仕様の C 言語や FORTRAN 言語を用いる。PC クラスタでは PVM や MPI を用いてプログラム開発を行う。

■ 並列処理の効率

並列処理において最も重要な観点は複数のプロセッサを効率的に動作させることであり、このためには対象問題、アルゴリズム、実装などの種々の局面で内在する並列性を抽出しなければならない。図 6.9 は問題領域からハードウェアに至るまでにおける並列性の度合いを示したものである。縦軸は並列して行える独立なオペレーションの数を示している。OS とハードウェアでのギャップは、たとえばプロセッサの処理速度とプロセッサネットワークのバンド幅の相違による。

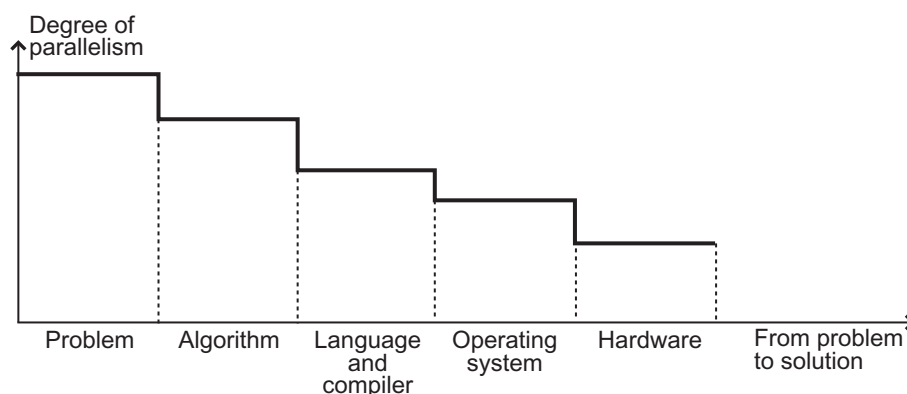


図 6.9: 各レベルにおける並列性⁵⁾

マトリックス演算 $C = AB$ を考える。各マトリックスのサイズを $n \times n$ とすると、並列計算の複雑度は $O(\log n)$ である。すべての要素の乗算は並列で行うことができるが、 n 回の加算に時間 $\log n$ が必要となる。しかし、 n 回の加算を逐次的に行うとこのアルゴリズムの複雑度は $O(n)$ となる。ここに問題とアルゴリズムとの並列度のギャップが生じる。

種々の問題を並列計算機に写像し、プロセッサ間のロードバランスをとるのは極めて難しい問題である。さらに、これがうまくできてもどこかの段階に修正が生じると全体を見直す必要が生じる。すなわち、並列計算は図 6.9 のすべての段階が密に関連している。

$$E_p = \frac{T_1}{pT_p} \quad (6.1)$$

並列処理の効率 は簡単には式 6.1 で表される．すなわち， p 個のプロセッサを用いたときの効率は 1 個のプロセッサで行ったときの逐次処理計算時間 T_1 を並列処理したときの計算時間 T_p と p の積で除したものである．ここで，逐次処理では最良のアルゴリズムを用いる必要がある．

$$R(f) = \frac{1}{(1-f) + f/R_p} \quad (6.2)$$

並列処理の数学的な効率としては **Amdahl の法則** がある．これは並列処理の部分とそうでない部分の比率の関数として全体としての処理速度を表したもので，全体の処理速度が低速部分に大きく影響されることをよく理解することができる．

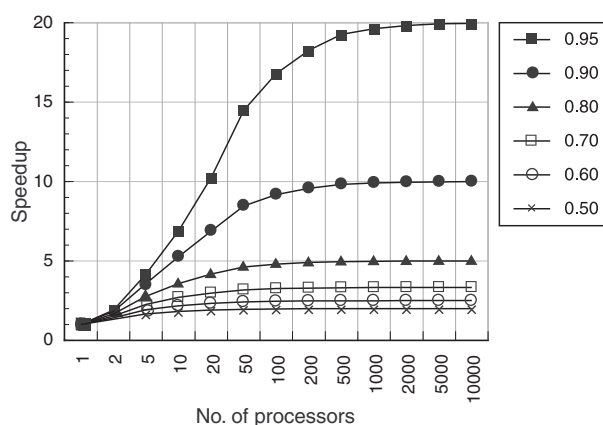


図 6.10: 並列化の速度向上

ここで， f は全体の処理における並列化可能な処理の割合であり， $R(f)$ は総合的な速度向上率， R_p は並列処理の **速度向上率** である．これを図示すると図 6.10 となる．たとえば逐次部分の割合がたとえ 5 % と少なくとも，逆に言えば並列処理部分が 95 % でも，1 万台のプロセッサで得られるスピードアップはわずか 20 倍にしかない．いかに逐次処理部分の割合が全体の効率を下げているかが分かる．

並列処理における効率に重大な影響を及ぼすもう一つのファクターはプロセッサ間通信の速度である．この速度が遅く，しかもプロセッサ間通信が多い場合には並列処理の効率は著しく悪くなる．

プロセッサ間通信の頻度は処理の **粒度** (grain size or granularity) に関係している．粒度とは複数のプロセッサに与えられるタスクの大きさのことである．一つのステートメントあるいはそれ以下のタスクを **細粒度** (fine grain)，ループまたはサブルーチンレベルのタスクを粗粒度 (coarse grain)，それらの中間的な大きさのタスクを **中粒度** (medium grain) とよぶ．粒度が小さくなればプロセッサ間通信が多くなり，反対に粒度が大きくなればプロセッサ間通信は少なくなる．このため，プロセッサ間通信の速度が遅く，細粒度の場合は並列処理の効率が著しく悪くなる．特に，PC クラスタの場合，一般にはプロセッサ間通信の速度が遅く，このため，細粒度での並列処理を行わせると複数台のプロセッサを用いているにも拘わらず，1 台のプロセッサを用いたときより遅くなることもある．このため，特に PC クラスタを用いる場合には，できるだけ粒度を大きくしなければならない．

並列処理の効率に大きな影響を与えるもう一つのファクターは **ロードバランス** である．ロードバランスとは並列計算機の各プロセッサに与えられる計算負荷の均等性のことであり，並列処理の効率を高めるには，できるだけロードをバランスさせなければならない．

各プロセッサに与えられる負荷が不均一であると，処理に不可避な同期待ちが多くなり，処理効率は著しく悪化する．たとえば，負荷が 2 倍異なれば計算時間が 2 倍異なり，同期をとるために遅いプロセッサを待つことになり，いくら他のすべてのプロセッサが早く処理を済ませてでも一番遅いプロセッサに律速される．このため，処理効率はこのことだけで 50 % 程度に悪化する．

こうした問題を克服するため、各プロセッサに与える負荷は出来るだけ均一になるようにしなければならない。しかしながら、これは難しい問題である。なぜなら、画像処理などのようにデータの量に応じて負荷が決まる場合はデータを分散させればロードバランスはとれるが、連立一次方程式の Gauss-Seidel 法に基づく解法や傾斜法に基づく最適化計算など、多くの繰り返し計算手法ではデータの質（内容）によって収束までの時間が異なるからである。

そのために、最初から静的に負荷を分割するのではなく、その時々各プロセッサの負荷を測定し、負荷を動的に均一化させる方法も有用である。しかし、この方法では負荷を管理するプロセッサが別に必要となり、しかも、負荷の測定のためにプロセッサ間通信が必要となり、別のオーバーヘッドが大きくなる。

■ 数値計算の構造

次に考えなければならない点は、数値計算に用いる概念モデルと計算モデルの組み合わせを数値計算の構造と呼ぶことにすると、数値計算の構造と並列処理の適合性はどうか、ということである。

図 6.11 はこの関係を示している。最初に特定のドメインでの問題があり、それを解くための概念モデルが作られる。このモデルの種類によってこの図でいう **数値ソルバー** を用いるか、**自然ソルバー** を用いるかが決まる。そしていずれの場合にも分割と写像が重要なポイントとなる。このブレイクダウンができれば仮想的な計算機モデルを作ることができ、そのモデルに依存してどのようなアーキテクチャの計算機を用いるかが決まる。たとえば固体解析に有限要素法を用いるとして、それを並列化する場合には粗粒度の分割、すなわち領域分割により計算の並列化を行うこともでき、一方、細粒度の分割、すなわち内部でのベクトル計算のループを並列化することもできる。最近ではシミュレーテッドアニーリング、遺伝的アルゴリズム、あるいはニューラルネットワークなどの、この図でいうところの自然ソルバーを用いる場合も多く、この場合にも異なった並列化の計算モデルが考えられる。たとえば、遺伝的アルゴリズムで、全体的母集団における個体を複数プロセッサに割り当てて進化を進める方法や島モデルで個体群を分散させて独立して進化させ、ときどき移民させるというモデルもある。いずれにせよ、シミュレーションで重要な点は最初概念モデルであり、そして次の計算モデルであり、これらの組み合わせで多くのタイプの並列処理が可能となる。

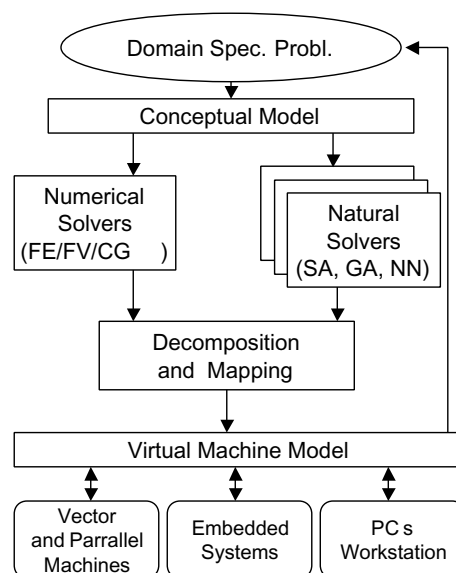


図 6.11: 数値計算の構造⁷⁾

■ 数値計算の並列化

□ 差分法の並列化

ラプラスの方程式やポアソン方程式で表される種々の物理現象（重力場、静電場、静磁場、定常熱伝導、定常物質拡散、渦なし非圧縮性流れ、弾性体の平衡、結晶の成長など）の **シミュレーション** は差分法に基づいて行われることが多い。この場合、並列化は細粒度では Red-Black 法などによって色分けされた並列化可能な格子点をグループ化したものをプロセッサに割り当てるか、あるいは中粒度では複数の格子点を格子状にグループ化した領域を並列可能性に基づいてプロセッサに割り当てる。もしくは大粒度では全体をプロセッサ数だけの領域に分割して行う。いずれの方法でも Jacobi 型の緩和法を用いるのか Gauss-Seidel 型の緩和法を用いるのかによって計算モデルが異なり、計算速度と並列化効率のバランスが変わる。

□ 有限要素法における並列化

有限要素法で定式化されたシミュレーションモデルでは細粒度ではベクトル計算の並列化となり、これは HPF などの並列プログラミング言語で比較的容易に並列化できる。大粒度での並列化では領域分割法（Domain Decomposition）が中心となっている。**領域分割法** は有限要素法におけるデータ分割あるいは負荷分散である。この方法では対象を幾何学的に複数の領域に分割し、それらに対応するプロセッサに割り当てる。そして、領域内の計算が終了すれば互いに境界条件を交換し、再度、領域内の計算を行う。これを反復することで全領域の計算を行う。領域分割法では領域の分割の方法によってプロセッサのロードバランスが大幅に異なるため、並列化効率を上げるにはこの点に注意する必要がある。

□ 粒子法による物質の挙動シミュレーション

多数の粒子の相互作用を基本として各粒子の位置や速度を求め、粒子全体からなるシステムの挙動を計算するアプローチを **粒子法**（particle method⁸⁾）という。相互作用が重力のときは天体力学における N 体問題となる。一方、相互作用がクーロン力の場合には分子動力学などのアプローチを用いて原子あるいは分子の配置を決定する解析が行える。

N 体問題の場合、重力は長距離力であり、 N 個の粒子の計算には $N(N-1)$ 回の相互作用を求める必要がある。これは N が大きくなると膨大な計算となる。これを回避するには空間を複数のセルに分割し、そのセルに含まれる粒子の平均的挙動をセルの状態として考え、ある一定の距離以上のセル内の粒子についてはセルとしての代表的な状態で近似する高速粒子法が必要となる。これによって計算回数は $N \log N$ ぐらいまで減少させることができる。ただし、この場合には空間に固定されたセル内の粒子の出入りを監視することが必要になる。

一方、分子動力学では原子間のクーロンポテンシャルが長距離で減衰が激しいことを利用し、ある一定の距離以上では他の粒子を考えなくても良い。これによって膨大な数の原子を取り扱うことができる。この場合に原子間距離の計算回数はやはり $N(N-1)$ 回となるため、たとえポテンシャルを計算しなくても距離計算の負荷が大きく、これを避けるには着目している原子の近傍粒子とそうでない粒子を分類し、その分類自体は時折行うことで毎回の距離計算自体を無くす **粒子登録法**（Bookkeeping method）が用いられる。

このような粒子法の並列化は大粒度では空間でのセルあるいはクラスター化した粒子群を一つのプロセッサに割り当てる方法であり、細粒度での並列化はベクトル計算における inner loop の並列化である。ただし、後者においてもプロセッサのロードバランスを良好にするために単一プロセッサの場合とは異なるアルゴリズムが必要となる。

■ おわりに

現在の自動並列化プログラミング言語は基本的に DO-LOOP の並列化が中心である。特に inner loop の並列化には効果的である。しかしながら、大規模シミュレーションの多くが最終的には膨大なベクトル計算に帰着するのはあくまでも図 6.9 でいうところの数値ソルバーを用いる場合である。しかしながら、同図でいう自然ソルバーにはまだまだ大きな発展の可能性があり、それらのアプローチでは種々の並列化の方法が可能となる。しかも、そうした自然ソルバーは本質的に並列処理を内在している場合が多い。このため、そうした自然な並列性を抽出して並列処理モデルを構築することはこれからの並列処理の可能性を大きく広げるものと考えられる。

■ 参考文献

- 1) 情報処理学会編. 情報処理ハンドブック. オーム社.
- 2) IBM. Kasparov vs. deep blue, 1999. <http://www.research.ibm.com/deepblue/>.
- 3) Mannheim University and Netlib. Top500 supercomputing sites, 1999. <http://www.netlib.org/benchmark/top500.html>.
- 4) パターソン, ヘネシー, 成田訳. コンピュータの構成と設計. 日経 BP 社.
- 5) 森真一郎, 富田真治. 並列計算機アーキテクトからみた計算機クラスタ. 情報処理, Vol. 39, No. 11, pp. 1073–, Nov. 1998.
- 6) D.I. Moldovan. *Parallel Processing from Applications to Systems*. Morgan Kaufmann.
- 7) P.M.A. Sloot. *Modelling for Parallel Simulation: Possibilities and Pitfalls*. EUROSIM.
- 8) E.F. Van de Velde. *Concurrent Scientific Computing*. Springer-Verlag.

アプリケーションの並列化の例

大分大学 工学部 生産システム工学科

三村泰成

mimura@cc.oita-u.ac.jp

■ はじめに

「並列処理」と一口に言っても、実際の応用となるとピンとこない方もいるかもしれません。特に実際の工学的な問題に適用しようとする、どのようにアプローチしていいかが分からないことがよくあります。ここでは、効率などはとりあえず後回しにして、とにかく解析領域を分割して、「えいや！」と並列化することを考えます。実際には、

- 差分法の並列化
熱伝導問題を差分法を使って解析します。まず、熱伝導方程式、差分法の基礎について簡単に述べ、それを領域分割を用いて並列化する方法を考えます。
- 遺伝的アルゴリズムの並列化
遺伝的アルゴリズム (以下、GA と称します) は、生物の進化を模倣した確率的最適化手法です。確率論的な手法の方が、アルゴリズムに依存関係が少なく、並列化に都合がいいことも少なくありません。そこで、GA の並列化モデルについて考えてみます。

の2つの例を見ていきます。

■ 領域分割を用いた並列計算

□ 熱伝導方程式 ¹⁾

熱伝導 は基礎的な物理現象の一つであり、**熱伝導方程式** は、数値解析技術の上からも比較的取扱が容易なので、数値解析の入門として一般的に用いられています。

簡単のため、まず、1次元問題を考えてみます。個体内に温度分布が存在すると熱の流れが生じます。単位時間 (s)、単位面積 (m^2) に流れる熱量を熱流束 q (J/m^2s あるいは W/m^2) と呼びます。熱流束は、個体内の温度勾配 (dT/dx) に比例し

$$q = -\lambda \frac{\partial T}{\partial x} \quad (7.1)$$

の関係があります。ここで λ は熱伝導率 (W/mK) と呼ばれる物性値です。通常、温度が減少する熱流の方向を正とすることから、式 (7.1) のように負号を付けます。また、微小体積についての熱の流入および流出を考慮した熱バランスは、

$$[\text{蓄熱による温度上昇}] = [\text{流入熱量}] - [\text{流出熱量}]$$

となり、

$$\rho c \frac{\partial T}{\partial t} = - \frac{\partial q}{\partial x} \quad (7.2)$$

が得られます。ここで、 ρ は個体の密度 (kg/m^3)、 c は比熱 (J/kgK) です。式 (7.3) に式 (7.1) を代入すると、

$$\rho c \frac{\partial T}{\partial t} = \frac{\partial}{\partial x} \left(\lambda \frac{\partial T}{\partial x} \right) \quad (7.3)$$

となり、さらに λ を一定とすれば

$$\frac{\partial T}{\partial t} = a \frac{\partial^2 T}{\partial x^2} \quad (7.4)$$

が得られます。これが 1 次元での熱伝導方程式です。ここで $a = \lambda/(\rho c)$ は、個体の温度伝導率 (m^2/s) と呼ばれます。同様に、2 次元の場合は、

$$\frac{\partial T}{\partial t} = a \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right) \quad (7.5)$$

と表されます。式 (7.4)、式 (7.5) の偏微分方程式を解くことで、物体の温度分布を求めることができます。

□ 差分法 ¹⁾

差分化方程式の導出

解を求めようとする領域を格子で分割し、かつ偏導関数を差分式で近似することにより偏微分方程式の数値解を求める方法を、一般的に「**差分法** (Finite Difference Method)」と呼んでいます。時間方向にも差分化を行うことによって、非定常解を求めることもできます。典型的な格子の例を図 7.1 に示します。

差分近似には 3 種類あり、図 7.2 中の点 P における微分値を考えると、以下のようになります。

- 前進差分 点 P と x の増加 (前進) 方向の点 E とを結んだ直線の傾きを求める方法

$$\frac{T_E - T_P}{\Delta x} \quad (7.6)$$

- 後退差分 点 P と x の減少 (後退) 方向の点 W とを結んだ直線の傾きを求める方法

$$\frac{T_P - T_W}{\Delta x} \quad (7.7)$$

- 中心差分 点 P を中心として、点 W と点 E とを結んだ直線の傾きを求める方法

$$\frac{T_E - T_W}{2\Delta x} \quad (7.8)$$

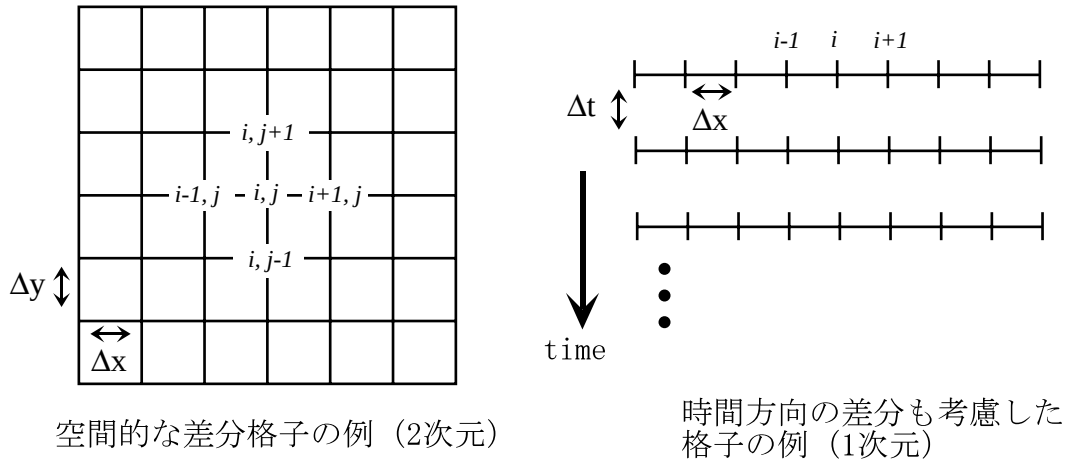


図 7.1: 差分格子の例

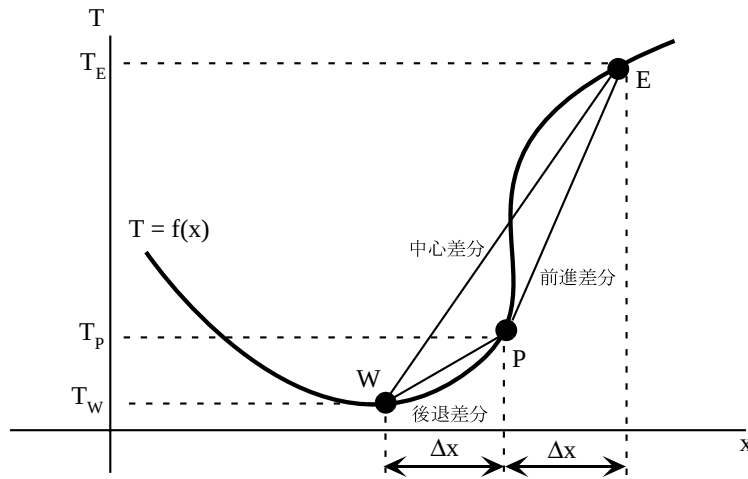


図 7.2: 差分近似の種類

簡単のため、1次元の熱伝導方程式の差分表示を考えてみます。点 i における空間座標に関する2階微分 $\left(\frac{\partial^2 T}{\partial x^2}\right)$ を求めるのに、図 7.3 を考えてみます。図中で点 A から点 B の区間をコントロールボリュームと呼び、これを用いて差分表示式を求める方法を「コントロールボリューム法」と呼びます。まず、図 7.3 中の点 A 、点 B における一階微分は、中心差分を用いると

$$\left(\frac{\partial T}{\partial x}\right)_A = \frac{T_i - T_{i-1}}{\Delta x}, \quad \left(\frac{\partial T}{\partial x}\right)_B = \frac{T_{i+1} - T_i}{\Delta x} \quad (7.9)$$

のようになります。2階微分は、 $\left(\frac{\partial T}{\partial x}\right)_A$ と $\left(\frac{\partial T}{\partial x}\right)_B$ の中心差分より、

$$\frac{\partial^2 T}{\partial x^2} = \frac{\left(\frac{\partial T}{\partial x}\right)_B - \left(\frac{\partial T}{\partial x}\right)_A}{\Delta x} = \frac{T_{i+1} + T_{i-1} - 2T_i}{\Delta x^2} \quad (7.10)$$

となります。したがって、式 (7.4) の1次元の熱伝導方程式は、

$$\frac{\partial T}{\partial t} = a \frac{T_{i+1} + T_{i-1} - 2T_i}{\Delta x^2} \quad (7.11)$$

となります。さらに、時間に関する前進差分を考慮し、時刻 n における点 i の温度を $T_i^{(n)}$ 、 Δt 秒後（時刻 $n+1$ ）の温度を $T_i^{(n+1)}$ とすれば、

$$\frac{T_i^{(n+1)} - T_i^{(n)}}{\Delta t} = a \frac{T_{i+1}^{(n)} + T_{i-1}^{(n)} - 2T_i^{(n)}}{\Delta x^2} \quad (7.12)$$

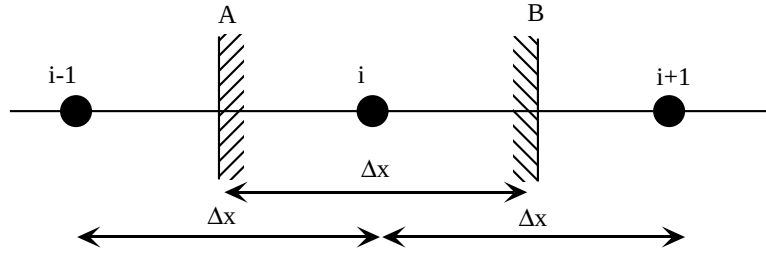


図 7.3: コントロールボリューム (1 次元)

となり、最終的に、

$$T_i^{(n+1)} = \gamma (T_{i+1}^{(n)} + T_{i-1}^{(n)} - 2T_i^{(n)}) + T_i^{(n)} \quad (7.13)$$

が得られ、これが 1 次元熱伝導方程式の差分化方程式となります。ここで γ は、 $\gamma = a\Delta t/\Delta x^2$ と定義します。同様に、2 次元の場合を考えると、式 (7.5) は、

$$\frac{T_{i,j}^{(n+1)} - T_{i,j}^{(n)}}{\Delta t} = a \left(\frac{T_{i+1,j}^{(n)} + T_{i-1,j}^{(n)} - 2T_{i,j}^{(n)}}{\Delta x^2} + \frac{T_{i,j+1}^{(n)} + T_{i,j-1}^{(n)} - 2T_{i,j}^{(n)}}{\Delta y^2} \right) \quad (7.14)$$

となり、 $\Delta x = \Delta y = h$ と置くと、

$$T_i^{(n+1)} = \gamma (T_{i+1,j}^{(n)} + T_{i-1,j}^{(n)} + T_{i,j+1}^{(n)} + T_{i,j-1}^{(n)} - 4T_{i,j}^{(n)}) + T_{i,j}^{(n)} \quad (7.15)$$

という、2 次元熱伝導方程式の差分化方程式が得られます。ここで γ は $\gamma = \Delta t/h^2$ と定義します。この方法は、時刻 n の温度分布から直ちに時刻 $n+1$ の温度分布が得られることから「陽解法」と呼ばれ、簡便なことから、多くの工学分野で用いられています。しかしながら、安定な解を得るには $0 \leq \gamma \leq 1/2$ という制限があり、 h^2 に比例して Δt を小さく取る必要があります。

偏微分方程式を解くための定式化を行ってきましたが、図 7.4 に示す計算過程を見れば、自身の状態量 (温度) と近傍点の状態量から、次の時刻の状態量が簡単に計算できることが分かります。これらの計算手順は、セルオートマトンなどとはほとんど同様であり、技術計算に詳しくない人でも比較的容易に理解できます。それゆえ、並列計算入門の例として適当なものの一つと言ってもよいでしょう。

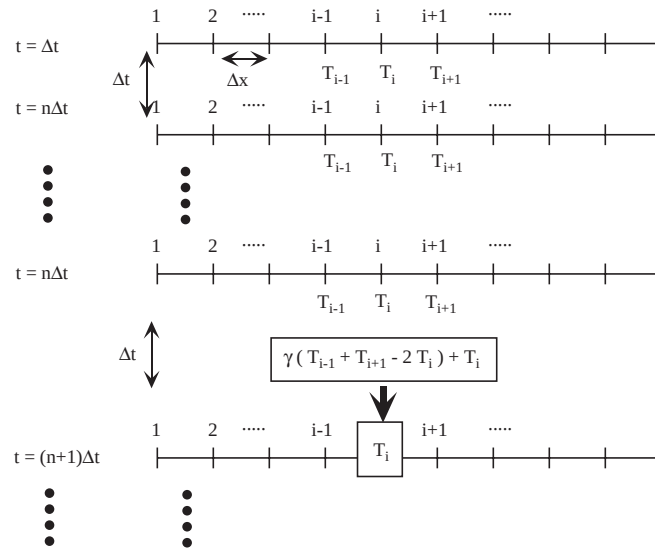


図 7.4: 差分法の計算手順 (1 次元)

境界条件

ここでは、境界条件として以下の2種を扱います。

- 断熱境界 熱の出入りが存在しない境界.

$$\left(\frac{\partial T}{\partial n}\right) = 0 \quad (7.16)$$

- 表面温度指定 温度が常に一定に保たれる境界.

$$T_B = T_0 \quad (const.) \quad (7.17)$$

ここで、境界面に垂直な座標を n 、境界面を B とします. 差分法における境界条件の指定方法を図 7.5 に示します.

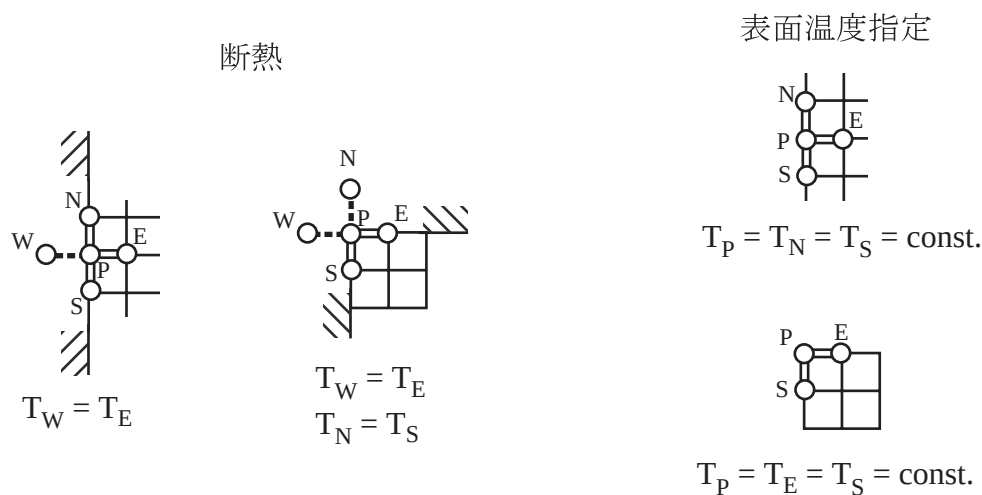


図 7.5: 境界条件

簡単な計算例

境界条件を温度指定のみを扱った解析対象と計算結果を図 7.6 に示し、ソースリスト 1 に解析プログラムを載せます. このような例では、120 行程度の簡単なものになっているのが分かります.

□ 領域分割による並列化

領域分割の種類

解析対象の分割方法は、簡単に考えると、1次元、2次元、3次元的な分割の3種類に分類できます. これらは、コンピュータのアーキテクチャや解析対象の複雑さなどを考慮して選択されるべきですが、今回は簡単のため解析対象の次元数にかかわらず、図 7.7 のように、1次元的に分割する例を考えます. また、並列化ライブラリとして PVM を用いています.

1次元熱伝導問題

並列化の説明のため、まず、図 7.8 のような1次元熱伝導問題を考えます. 計算の流れを図 7.9 に示し、ソースリスト 2 にプログラムを示します. プログラムの簡単な説明を以下に示します.

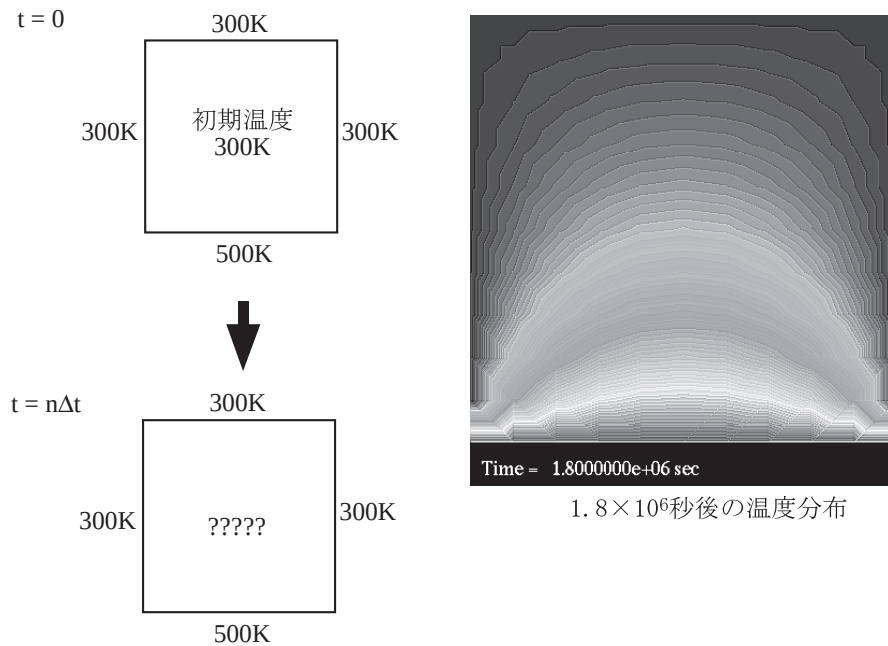


図 7.6: 温度指定のみを用いた解析例

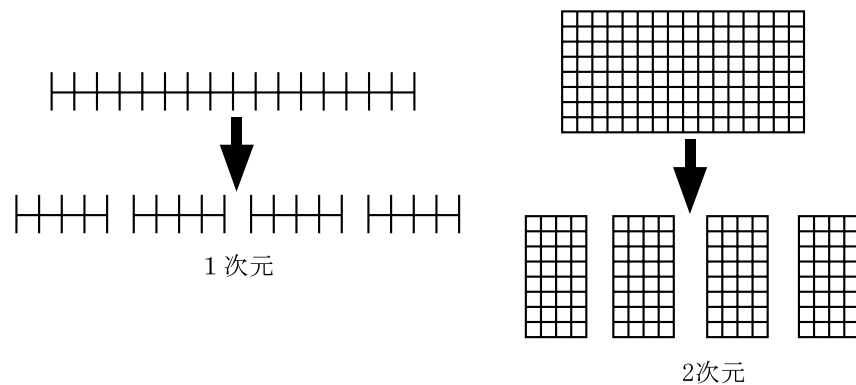


図 7.7: 1次元的な領域分割

- 66 行目：自分のタスク ID の取得.
- 71 行目：タスクのグループ化.
- 73～76 行目：もし、最初のタスクなら、他のタスクを立ち上げる.
- 77 行目：同期待ち.
- 79～94 行目：タスク ID の情報を全てのタスクに通知する.
- 96～99 行目：自分の受け持ちの計算幅の決定.
- 100～114 行目：初期温度の設定.
- 116～195 行目：差分計算のループ.
 - 119～132 行目：左右の境界温度の差分計算.
 - 133～138 行目：内部の差分計算.
 - 140～162 行目：隣の CPU との通信.

- 164～179 行目：流入熱量と流出熱量の計算.
- 188 行目：1 ステップごとに同期をとるところ.
- 190～193 行目：温度の更新.
- 197～201 行目：PVM の後処理.

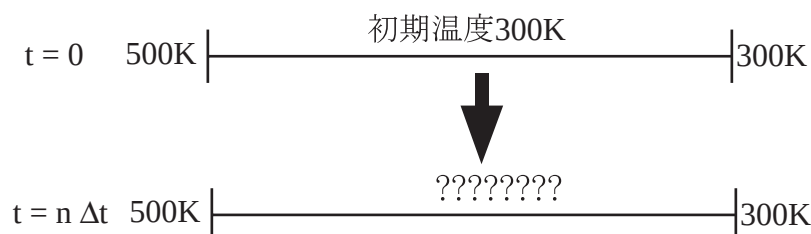


図 7.8: 解析対象 (1 次元)

2 次元熱伝導問題

図 7.10 のような 2 次元熱伝導問題を考えます. ここでは境界条件として, 断熱も扱っています. 1 次元問題と同様に, 解析対象を 1 次元的に分割することで, 並列計算を実現します. それゆえ, 計算手順は, 図 7.9 と同様になります. 逐次型のプログラムをソースリスト 3, 並列化のプログラムをソースリスト 4 に示します. ソースリスト 2 と 4 を比較すれば, 2 次元問題も 1 次元問題と同じ手順で並列計算が行われているのが分かります. また, ソースリスト 5 は, 1 次元配列, 2 次元配列のメモリ確保のためのユーティリティ関数です.

□ 並列計算の結果

並列計算の計測には, 「Pentium-III 500MHz (dual) メモリ 256MB」というスペックのコンピュータを 8 台接続した PC クラスタを用いました. また, 各ノードは, HUB(スイッチングなし)を通じて, 10BASE の Ethernet で接続されています. 上述の 2 次元熱伝導問題を, 格子サイズ「5001 × 2501」として計算した場合の加速率のグラフを図 7.11 に示します. 4 タスクの所までは, 理想的な加速を得ています. しかしながら, それよりも多いタスクを使つての計算は, あまりいい加速は得られているとはいえません. 16 タスクで良好な加速率を得るためには, 解析対象がもっと大きい必要があります.

□ 領域分割を用いたその他の応用

解析領域を分割して並列化する手法は, 差分法以外にも用いられています. 以下にその例を示します.

分子動力学

分子動力学 とは, 分子間になんらかのポテンシャルが存在すると定義し (通常は分子間距離や角度の関数として定義されます), そのポテンシャルから, 分子に作用する力を計算し, その力から, 運動方程式 ($F = ma$) を解くことで, 時々刻々の分子の位置や速度を求める手法です²⁾.

ポテンシャルは大きく分けて, 長距離力 MD, と短距離力 MD の 2 種に分けることができます. 長距離力 MD の場合, 1 個の分子に作用力を求めるのに, 解析領域全体を考慮する必要がありますが, 短距離力 MD の場合, 近傍の分子のみを考慮すれば, 作用力を求めることができます. それゆえ, 差分法と同様な領域分割法を用いることができます^{3) 4)}. 分子動力学に限らず, 以下の

- 近傍のデータの交換
- 陽解法

という性質を満たすだけでよい手法であれば、領域分割法をすみやかに適用することができると言えます。

有限要素法

有限要素法 を並列化する手法としても、領域分割法が使われています^{5) 6)}。しかしながら、構造解析などの定常問題を解く場合は、陽解法と同じ手法を使うわけにはいきません。領域境界の物理量 (例えば力と変位) は、初期の計算では、各領域ごとに異なったものとなります。それゆえ、解が収束するまで反復計算する必要があります。

■ 遺伝的アルゴリズムの並列化

申し訳ありません。

後日、この部分の資料は後日、配布致します。

■ その他 (並列以外のツール)

PC クラスタは、日頃から使っている「普通」のコンピュータを並列に接続しています。それゆえ、超並列計算機などの特殊なハードウェアと異なり、日頃から使っている便利なツール群をそのまま利用することができます。それゆえ、結果の表示なども安価で簡単に実現できます。以下に紹介するツールは、Debian Linux などのパッケージから容易にインストール可能なものばかりですので、もし、興味がある方は使ってみて下さい。ちなみに、今回は、GTK+ と MPEGENCODER を結果の表示に用いました。

- **GTK+**⁷⁾
X-Window 上の GUI 構築ライブラリです。現時点で以下の言語から利用できます。
Ada95, C++, Dylan, Eiffel, Guile, Haskell, JavaScript, Objective-C, Objective-Caml, Pascal, Perl, Pike, Python, Tom.
- **GULT**⁸⁾, **MESA**⁹⁾
OpenGL¹⁰⁾ 仕様の 3 次元表示用ライブラリです。C, C++ などのプログラミング言語から使うことができます。
- **VRweb, VRwave**¹¹⁾
VRML (the Virtual Reality Modeling Language) のブラウザです。VRML で解析モデルを記述しておけば、簡単な表示に使えます。
- **POV-Ray**¹²⁾
3 次元レイトレーシングソフトです。並列計算の実装も施されており、綺麗な 3 次元画像を比較的簡単に作ることができます。
- **MPEGENCODER**¹³⁾
GIF, PPM, JPEG などの画像から、簡単にパラパラアニメーションを作ることができるソフトです。

■ おわりに

今回は、いくつかの並列計算の応用を考えましたが、必ずしも効率がよいものとは言えないかもしれません。しかしながら、並列計算を「とりあえず」始めるには、よい方法であると考えられます。

今回の講習が、皆さんのお役に立てば幸いです。

■ 参考文献

- 1) 河村洋, 差分法と熱伝導, **日本機械学会講習会 伝熱工学数値計算入門 教材**, No.900-68, pp.1-10 (1990).
- 2) 方平洋右, 分子動力学法とモンテカルロ法, 講談社サイエンティフィク (1994).
- 3) 三村泰成, 赤星保浩, 原田昭治, 並列計算機 AP1000 を用いた分子動力学シミュレーション, **日本機械学会 平成8年度材料力学部門講演会講演論文集**, pp.199-200 (1996).
- 4) 松原正純, 板倉憲一, 朴泰佑, 超並列計算機 CP-PACS における大規模分子動力学法シミュレーション, **並列処理シンポジウム '98 論文集**, pp.215-222(1998).
- 5) YAGAWAQ, G., SONEDA, N., and YOSHIMURA, S. A large scale finite element analysis using domain decomposition method using parallel computer, *Computational Plasiy Models, Software and Applications*, pp.177-197 (1989).
- 6) 吉岡顕, 大規模有限要素法のためのネットワークコンピューティング, 東京大学大学院博士論文 (1990).
- 7) 田中ひろゆき, GTK+入門 基礎からはじめる X プログラミング, ソフトバンク (1999).
- 8) http://reality.sgi.com/mjk_asd/glut3/glut3.html
- 9) <http://www.mesa3d.org/>
- 10) <http://www.sgi.com/software/opengl/>
- 11) <http://www.iicm.edu/vrwave>
- 12) 小室日出樹, POV-Ray ではじめるレイトレーシング, アスキー (1998).
- 13) <http://bmrc.berkeley.edu/frame/research/mpeg/index.html>

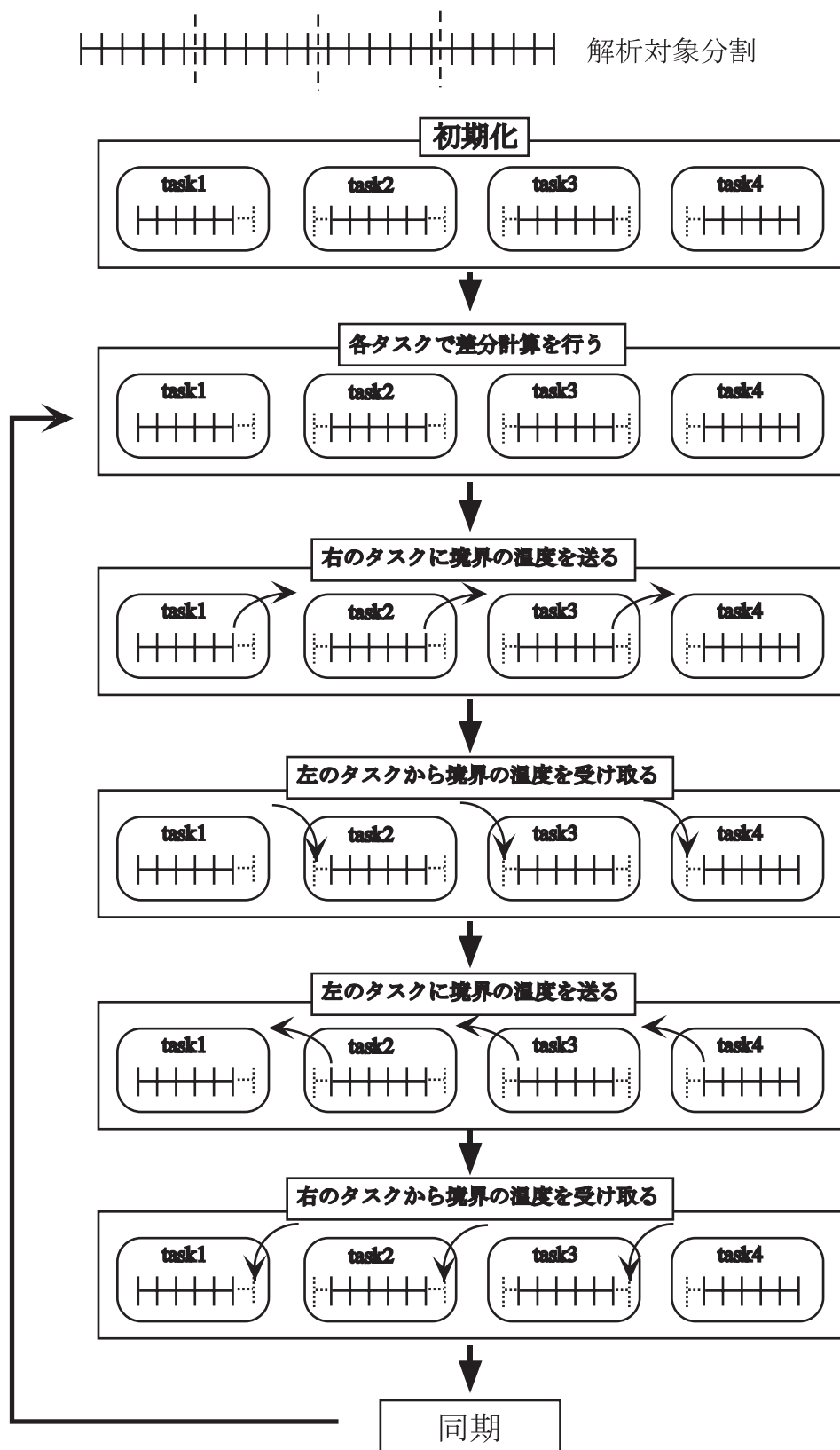


図 7.9: 並列計算の流れ

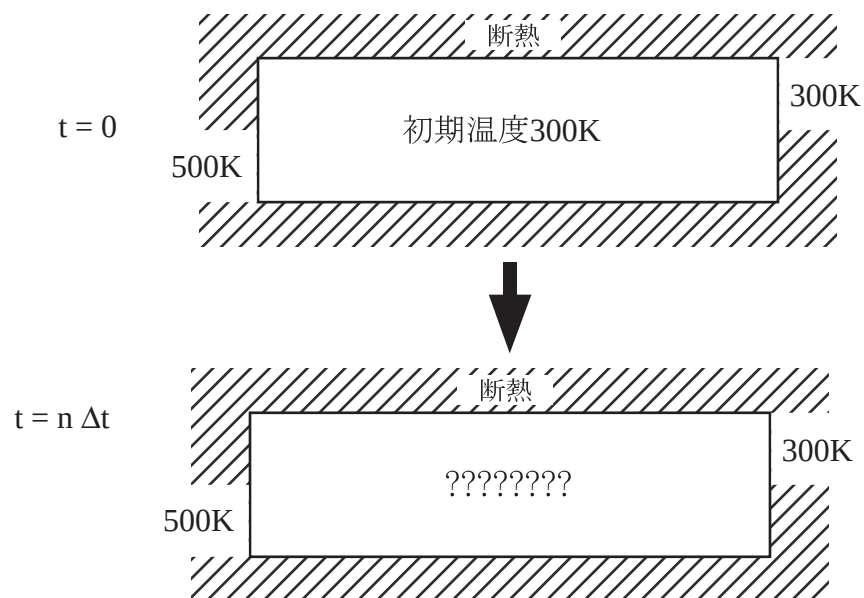


図 7.10: 解析対象 (2次元)

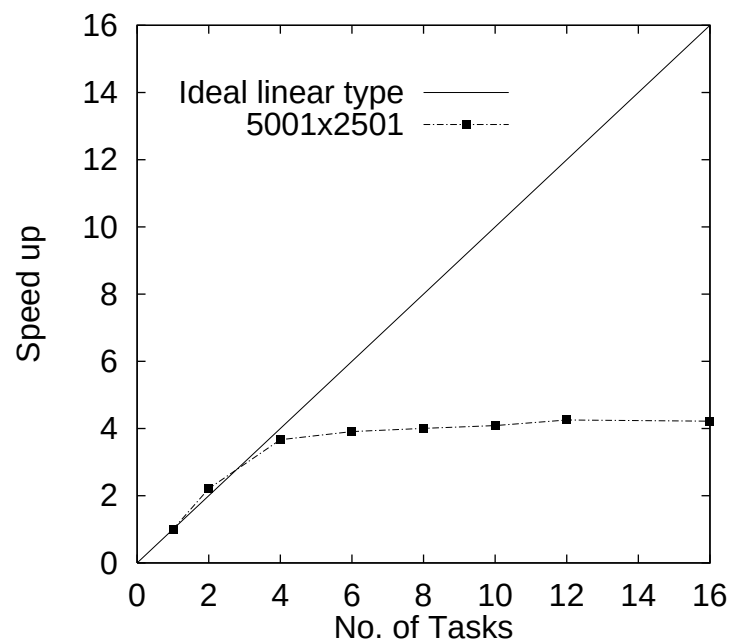


図 7.11: 加速率

ソースリスト 1 (efdm.c)

```

1: /*
2:  Filename: efdm.c
3:  Description: Simple example of 2D FDM.
4:  Writer: Yasunari Mimura
5:  Date: 1999.9.10
6:  */
7:
8: #include <stdlib.h>
9: #include <stdio.h>
10: #include <math.h>
11:
12: #define NW 11
13: #define NH 11
14:
15: int nlimit = 180;
16:
17: double width = 10.0; /* m */
18: double height = 10.0; /* m */
19: double h = 1.0; /* dx = dy = h */
20: double dt = 10000.0; /* sec */
21:
22: double T[NW][NH]; /* K */
23: double T_new[NW][NH]; /* K */
24:
25: double temp_high = 500.0;
26: double temp_low = 300.0;
27:
28: /* property (carbon steel) */
29: double rou = 7.8e3; /* kg / (m^3) */
30: double c = 420.0; /* J/(kg K) */
31: double k = 50.0; /* W/(m K) */
32:
33: void print_temp(FILE *fp, double t)
34: {
35:     int i, j;
36:
37:     fprintf(fp, "time\t%15.7e\n", t);
38:     for(i=0; i<NW; i++) {
39:         for(j=0; j<NH; j++) {
40:             fprintf(fp, "%15.7e\t", T_new[i][j]);
41:         }
42:         fprintf(fp, "\n");
43:     }
44: }
45:
46: int main()
47: {
48:     int nprint;
49:     int i, j;
50:     int count;
51:     double a, t;
52:     double Te, Tw, Ts, Tn;
53:     double gamma;
54:     FILE *fp;
55:
56:     a = k / rou / c;
57:     gamma = a * dt / (h*h);
58:
59:     if( gamma > 0.5 ) {
60:         printf("gamma = %e\n", gamma);
61:         exit(0);
62:     }
63:
64:     /* Init. Temp. */
65:     for(i=0; i<NW; i++) {
66:         for(j=0; j<NH; j++) {
67:             T[i][j] = temp_low;
68:             T_new[i][j] = temp_low;
69:         }
70:     }
71:     for(i=0; i<NW; i++) {
72:         T[i][0] = temp_high;
73:         T_new[i][0] = temp_high;
74:     }
75:
76:     fp = fopen("temp.dat", "w");
77:     fprintf(fp, "n\t%d\nnh\t%d\nw\t%e\nh\t%e\n",
78:             NW, NH, width, height);
79:     fprintf(fp, "T_max\t%e\nT_min\t%e\n",
80:             temp_high, temp_low);
81:
82:     nprint = 0;
83:     t = 0.0;
84:     print_temp(fp, t);
85:     nprint++;
86:
87:     for(count=1; count <= nlimit; count++) {
88:         t += dt;
89:
90:         /* Calc. inner temp. */
91:         for(i=1; i<NW-1; i++) {
92:             for(j=1; j<NH-1; j++) {
93:                 Tw = T[i-1][j];
94:                 Te = T[i+1][j];
95:                 Ts = T[i][j-1];
96:                 Tn = T[i][j+1];
97:                 T_new[i][j] = gamma*(Tw + Te + Ts + Tn - 4.0*T[i][j]) + T[i][j];
98:             }
99:         }
100:
101:         /* write data */
102:         if( (count%1) == 0 ) {
103:             printf("t = %f\n", t);
104:             print_temp(fp, t);
105:             nprint++;
106:         }
107:
108:         /* Update temp. */
109:         for(i=1; i<NW-1; i++) {
110:             for(j=1; j<NH-1; j++) {
111:                 T[i][j] = T_new[i][j];
112:             }
113:         }

```

```

114:     }
115:
116:     fclose(fp);
117:     fp = fopen("nprint.dat", "w");
118:     fprintf(fp, "%d\n", nprint);
119:     fclose(fp);
120:     return 0;
121: }

```

ソースリスト 2 (pfdm1d.c)

```

1: /*
2:  Filename: pfdm1d.c
3:  Description: 1D Finite difference method for Multi-CPU.
4:  Writer: Yasunari Mimura
5:  Date: 1999.9.10
6:  */
7:
8: #include <stdlib.h>
9: #include <stdio.h>
10: #include <math.h>
11: #include <pvm3.h>
12:
13: #define TID_MSG 0
14: #define T_LR_MSG 1
15: #define T_RL_MSG 2
16: #define Q_MSG 3
17:
18: #define NW 101
19: int nlimit = 40000;
20:
21: int my_nw;
22: double dx = 0.1; /* width / (NW-1) */
23: double dt = 100.0; /* sec */
24:
25: double width = 10.0; /* m */
26:
27: double T[NW]; /* K */
28: double T_new[NW]; /* K */
29: double T_left;
30: double T_right;
31: double temp_high = 500.0;
32: double temp_low = 300.0;
33:
34: /* property (carbon steel) */
35: double rou = 7.8e3; /* kg / (m^3) */
36: double c = 420.0; /* J/(kg K) */
37: double k = 50.0; /* W/(m K) */
38:
39: /* for parallel variables*/
40: int mytid;
41: int my_i;
42: int iproc;
43: int nproc = 4;
44: int tids[4]; /* array of task IDs */
45:
46: int main(int argc, char **argv)
47: {
48:
49:     int i, j;
50:     int count;
51:     double a, t;
52:     double Te, Tw, Ts, Tn;
53:     double gamma;
54:     double q_in, q_out;
55:     double dq = DBL_MAX;
56:
57:     a = k / rou / c;
58:     gamma = a * dt / (dx*dx);
59:
60:     if( gamma > 0.5 ) {
61:         printf("gamma = %e\n", gamma);
62:         exit(1);
63:     }
64:
65:     /* enroll in pvm */
66:     mytid = pvm_mytid();
67:
68:     /* Join a group and, if I am the first instance,
69:     iproc=0, spawn more copies of myself
70:     */
71:     iproc = pvm_joygroup("FDM");
72:
73:     if (iproc == 0) {
74:         tids[0] = pvm_mytid();
75:         pvm_spawn("pfdm1d", &argv[1], 0, NULL, nproc-1, &tids[1]);
76:     }
77:     pvm_barrier("FDM", nproc);
78:
79:     /* get task IDs */
80:     if( iproc == 0 ) {
81:         pvm_initsend(PvmDataRaw);
82:         pvm_pkint(&tids[0], nproc, 1);
83:         pvm_bcast("FDM", TID_MSG);
84:         my_i = 0;
85:     } else {
86:         pvm_recv( -1, TID_MSG);
87:         pvm_upkint(&tids[0], nproc, 1);
88:         for(i=1; i<nproc; i++) {
89:             if(tids[i] == mytid) {
90:                 my_i = i;
91:                 break;
92:             }
93:         }
94:     }
95:
96:     my_nw = NW / nproc;
97:     i = NW % nproc;
98:     if(my_i < i)
99:         my_nw++;
100:
101:     /* Init. Temp. */
102:     for(i=0; i<my_nw; i++) {

```

```

103:     T[i] = temp_low;
104:     T_new[i] = temp_low;
105: }
106:
107: T_left = temp_low;
108: T_right = temp_low;
109: t = 0.0;
110:
111: if( my_i == 0 ) {
112:     T[0] = temp_high;
113:     T_new[0] = temp_high;
114: }
115:
116: for(count=1 ; count<=nlimit ; count++) {
117:     t += dt;
118:
119:     /* Left boundary */
120:     if( my_i != 0 ) {
121:         i = 0;
122:         Tw = T_left; /* left side data */
123:         Te = T[i+1];
124:         T_new[i] = gamma*(Tw + Te - 2.0*T[i]) + T[i];
125:     }
126:     /* Right boundary */
127:     if( my_i != (nproc-1) ) {
128:         i = my_nw-1;
129:         Tw = T[i-1];
130:         Te = T_right; /* right side data */
131:         T_new[i] = gamma*(Tw + Te - 2.0*T[i]) + T[i];
132:     }
133:     /* Calc. inner temp. */
134:     for(i=1 ; i<my_nw-1 ; i++) {
135:         Tw = T[i-1];
136:         Te = T[i+1];
137:         T_new[i] = gamma*(Tw + Te - 2.0*T[i]) + T[i];
138:     }
139:
140:     /*--- Exchange data ---*/
141:     if( my_i != nproc - 1 ) {
142:         /* send to right side */
143:         pvm_initsend(PvmDataRaw);
144:         pvm_pkdouble( &T[my_nw-1], 1, 1);
145:         pvm_send( tids[my_i+1], T_LR_MSG);
146:     }
147:     if( my_i != 0 ) {
148:         /* receive from left side */
149:         pvm_recv( tids[my_i-1], T_LR_MSG);
150:         pvm_upkdouble( &T_left, 1, 1);
151:     }
152:     if( my_i != 0 ) {
153:         /* send to left side */
154:         pvm_initsend(PvmDataRaw);
155:         pvm_pkdouble( &T[0], 1, 1);
156:         pvm_send( tids[my_i-1], T_RL_MSG);
157:     }
158:     if( my_i != nproc - 1 ) {
159:         /* receive from right side */
160:         pvm_recv( tids[my_i+1], T_RL_MSG);
161:         pvm_upkdouble( &T_right, 1, 1);
162:     }
163:
164:     if( (count%100) == 0 ) {
165:         if( my_i == 0 ) {
166:             q_in = T[0] - T[1];
167:             pvm_recv( tids[nproc-1], Q_MSG);
168:             pvm_upkdouble( &q_out, 1, 1);
169:             dq = q_in - q_out;
170:             printf("t = %e\n", t);
171:             printf("Q_in = %e, Q_out = %e, dq = %e\n",
172:                 q_in, q_out, dq );
173:         } else if( my_i == nproc - 1 ) {
174:             q_out = T[my_nw - 2] - T[my_nw - 1];
175:             pvm_initsend(PvmDataRaw);
176:             pvm_pkdouble( &q_out, 1, 1);
177:             pvm_send( tids[0], Q_MSG);
178:         }
179:     }
180:
181:     /* write data */
182:     /*
183:     if( (count%100) == 0){
184:         --- print Temp.!! ---
185:     }
186:     */
187:
188:     pvm_barrier("FDM", nproc);
189:
190:     /* Update temp. */
191:     for(i=0 ; i<my_nw ; i++) {
192:         T[i] = T_new[i];
193:     }
194: }
195:
196:
197: /* Check program finished, leave group, exit pvm */
198: pvm_barrier("FDM", nproc);
199: pvm_lvgroup("FDM");
200: pvm_exit();
201: return(0);
202: }

```

```

11:
12: int nlimit = 10000;
13: #define NW 21
14: #define NH 11
15: double h = 1.0; /* dx = dy = h */
16: double dt = 10000.0; /* sec */
17: double width = 20.0; /* m */
18: double height = 10.0; /* m */
19: double dd = 1.0; /* m */
20: double T[NW][NH]; /* K */
21: double T_new[NW][NH]; /* K */
22: double temp_high = 500.0;
23: double temp_low = 300.0;
24:
25: /* property (carbon steel) */
26: double rou = 7.8e3; /* kg / (m^3) */
27: double c = 420.0; /* J/(kg K) */
28: double k = 50.0; /* W/(m K) */
29:
30: double q_in, q_out;
31: int h_mid;
32:
33: void print_temp(FILE *fp, double t)
34: {
35:     int i, j;
36:     fprintf(fp, "time\t%15.7e\n", t);
37:     for(i=0 ; i<NW ; i++) {
38:         for(j=0 ; j<NH ; j++) {
39:             fprintf(fp, "%15.7e\t", T_new[i][j]);
40:         }
41:         fprintf(fp, "\n");
42:     }
43: }
44:
45: void calc_q()
46: {
47:     int j;
48:     q_in = 0.0;
49:     for(j=0 ; j<h_mid+1 ; j++) {
50:         q_in += k * (T_new[0][j] - T_new[1][j]) * dd;
51:     }
52:     q_out = 0.0;
53:     for(j=h_mid ; j<NH ; j++) {
54:         q_out += k * (T_new[NW-2][j] - T_new[NW-1][j]) * dd;
55:     }
56: }
57:
58: int main()
59: {
60:     int i, j;
61:     int count;
62:     int nprint;
63:     double a, t;
64:     double Te, Tw, Ts, Tn;
65:     double gamma;
66:     double dq = DBL_MAX;
67:     FILE *fp;
68:
69:     a = k / rou / c;
70:     gamma = a * dt / (h*h);
71:     h_mid = NH / 2;
72:
73:     printf("gamma = %e\n", gamma);
74:     if( gamma > 0.5 ) {
75:         exit(0);
76:     }
77:
78:     /* Init. Temp. */
79:     for(i=0 ; i<NW ; i++) {
80:         for(j=0 ; j<NH ; j++) {
81:             T[i][j] = temp_low;
82:             T_new[i][j] = temp_low;
83:         }
84:     }
85:     for(j=0 ; j<h_mid+1 ; j++) {
86:         T[0][j] = temp_high;
87:         T_new[0][j] = temp_high;
88:     }
89:
90:     fp = fopen("temp.dat", "w");
91:     fprintf(fp, "nw\t%d\nnh\t%d\nw\t%e\nh\t%e\n",
92:         NW, NH, width, height);
93:     fprintf(fp, "T_max\t%e\nT_min\t%e\n",
94:         temp_high, temp_low);
95:
96:     t = 0.0;
97:     nprint = 0;
98:     calc_q();
99:     dq = q_in - q_out;
100:     printf("t = %f\n", t);
101:     printf("Q_in = %e, Q_out = %e, dq = %e\n",
102:         q_in, q_out, dq );
103:     print_temp(fp, t);
104:     nprint++;
105:
106:     for(count=1 ; count<=nlimit ; count++) {
107:         t += dt;
108:         /* Left boundary */
109:         i = 0;
110:         for(j=h_mid+1 ; j<NH-1 ; j++) {
111:             Tw = T[i+1][j]; /* insulation */
112:             Te = T[i+1][j];
113:             Ts = T[i][j-1];
114:             Tn = T[i][j+1];
115:             T_new[i][j] = gamma*(Tw + Te + Ts + Tn - 4.0*T[i][j]) + T[i][j];
116:         }
117:         /* Right boundary */
118:         i = NW-1;
119:         for(j=1 ; j<h_mid ; j++) {
120:             Tw = T[i-1][j];
121:             Te = T[i-1][j]; /* insulation */
122:             Ts = T[i][j-1];
123:             Tn = T[i][j+1];
124:             T_new[i][j] = gamma*(Tw + Te + Ts + Tn - 4.0*T[i][j]) + T[i][j];
125:         }
126:         /* Upper boundary */

```

ソースリスト 3 (sfdm2d.c)

```

1: /*
2:  Filename: sfdm2d.c
3:  Description: 2D FDM for single CPU.
4:  Writer: Yasunari Mimura
5:  Date: 1999.9.10
6:  */
7:
8: #include <stdlib.h>
9: #include <stdio.h>
10: #include <math.h>

```

```

126:     j = NH-1;
127:     for(i=1 ; i<NW-1 ; i++) {
128:         Tw = T[i-1][j];
129:         Te = T[i+1][j];
130:         Ts = T[i][j-1];
131:         Tn = T[i][j+1]; /* insulation */
132:         T_new[i][j] = gamma*(Tw + Te + Ts + Tn - 4.0*T[i][j]) + T[i][j];
133:     }
134:     /* Lower boundary */
135:     j = 0;
136:     for(i=1 ; i<NW-1 ; i++) {
137:         Tw = T[i-1][j];
138:         Te = T[i+1][j];
139:         Ts = T[i][j+1]; /* insulation */
140:         Tn = T[i][j+1];
141:         T_new[i][j] = gamma*(Tw + Te + Ts + Tn - 4.0*T[i][j]) + T[i][j];
142:     }
143:     /* corner */
144:     i = 0;
145:     j = NH-1;
146:     Tw = T[i+1][j]; /* insulation */
147:     Te = T[i+1][j];
148:     Ts = T[i][j-1];
149:     Tn = T[i][j-1]; /* insulation */
150:     T_new[i][j] = gamma*(Tw + Te + Ts + Tn - 4.0*T[i][j]) + T[i][j];
151:
152:     i = NW-1;
153:     j = 0;
154:     Tw = T[i-1][j];
155:     Te = T[i-1][j]; /* insulation */
156:     Ts = T[i][j+1]; /* insulation */
157:     Tn = T[i][j+1];
158:     T_new[i][j] = gamma*(Tw + Te + Ts + Tn - 4.0*T[i][j]) + T[i][j];
159:
160:     /* Calc. inner temp. */
161:     for(i=1 ; i<NW-1 ; i++) {
162:         for(j=1 ; j<NH-1 ; j++) {
163:             Tw = T[i-1][j];
164:             Te = T[i+1][j];
165:             Ts = T[i][j-1];
166:             Tn = T[i][j+1];
167:             T_new[i][j] = gamma*(Tw + Te + Ts + Tn - 4.0*T[i][j]) + T[i][j];
168:         }
169:     }
170:
171:     calc_q();
172:     dq = q_in - q_out;
173:
174:     /* write data */
175:     if( (count%100) == 0 ) {
176:         printf("t = %f\n", t);
177:         printf("Q_in = %e, Q_out = %e, dq = %e\n",
178:             q_in, q_out, dq );
179:         print_temp(fp, t);
180:         nprint++;
181:     }
182:
183:     if( dq < 1.0e-3 )
184:         break;
185:
186:     /* Update temp. */
187:     for(i=0 ; i<NW ; i++) {
188:         for(j=0 ; j<NH ; j++) {
189:             T[i][j] = T_new[i][j];
190:         }
191:     }
192: }
193:
194: printf("t = %f\n", t);
195: printf("Q_in = %e, Q_out = %e, dq = %e\n",
196:     q_in, q_out, dq );
197: print_temp(fp, t);
198: nprint++;
199: fclose(fp);
200: fp = fopen("nprint.dat", "w");
201: fprintf(fp, "%d\n", nprint);
202: fclose(fp);
203: return 0;
204: }

```

```

32: double width = 20.0; /* m */
33: double height = 10.0; /* m */
34:
35: double dd = 1.0; /* m */
36:
37: double **T; /* K */
38: double **T_new; /* K */
39:
40: double *T_left;
41: double *T_right;
42:
43: double temp_high = 500.0;
44: double temp_low = 300.0;
45:
46: /* property (carbon steel) */
47: double rou = 7.8e3; /* kg / (m^3) */
48: double c = 420.0; /* J/(kg K) */
49: double k = 50.0; /* W/(m K) */
50:
51: double q_in, q_out;
52: int h_mid;
53:
54: void calc_Qin()
55: {
56:     int j;
57:
58:     q_in = 0.0;
59:     for(j=0 ; j<h_mid+1 ; j++) {
60:         q_in += k * (T_new[0][j] - T_new[1][j]) * dd * h;
61:     }
62: }
63:
64: void calc_Qout()
65: {
66:     int j;
67:
68:     q_out = 0.0;
69:     for(j=h_mid ; j<NH ; j++) {
70:         q_out += k * (T_new[my_nw-2][j] - T_new[my_nw-1][j]) * dd * h;
71:     }
72: }
73: }
74:
75:
76: int main(int argc, char **argv)
77: {
78:     int mytid;
79:     int my_i;
80:     int iproc;
81:     int nproc;
82:     int *tids; /* array of task IDs */
83:     double *buf;
84:
85:     int i, j;
86:     int count;
87:     double a, t;
88:     double Te, Tw, Ts, Tn;
89:     double gamma;
90:     double dq = DBL_MAX;
91:
92:     a = k / rou / c;
93:     gamma = a * dt / (h*h);
94:     h_mid = NH / 2;
95:
96:     if( gamma > 0.5 ) {
97:         printf("gamma = %e\n", gamma);
98:         exit(1);
99:     }
100:
101:     if( argc != 2 ) {
102:         fprintf(stderr, "Usage: %s [n_tasks]\n", argv[0]);
103:         pvm_exit();
104:         exit(1);
105:     }
106:     nproc = atoi(argv[1]);
107:     tids = alloc_int(nproc);
108:
109:     /* enroll in pvm */
110:     mytid = pvm_mytid();
111:
112:     /* Join a group and, if I am the first instance,
113:        iproc=0, spawn more copies of myself
114:        */
115:     iproc = pvm_joygroup("FDM2D");
116:
117:     if (iproc == 0) {
118:         tids[0] = pvm_mytid();
119:         pvm_spawn("pfdm2d", &argv[1], 0, NULL, nproc-1, &tids[1]);
120:     }
121:     pvm_barrier("FDM2D", nproc);
122:
123:     /* get task IDs */
124:     if( iproc == 0 ) {
125:         pvm_initend(PvmDataRaw);
126:         pvm_pkint(&tids[0], nproc, 1);
127:         pvm_bcast("FDM2D", TID_MSG);
128:         my_i = 0;
129:     } else {
130:         pvm_recv(-1, TID_MSG);
131:         pvm_upkint(&tids[0], nproc, 1);
132:         for(i=1 ; i<nproc ; i++) {
133:             if(tids[i] == mytid) {
134:                 my_i = i;
135:                 break;
136:             }
137:         }
138:     }
139:
140:     my_nw = NW / nproc;
141:     i = NW % nproc;
142:     if(my_i < i)
143:         my_nw++;
144:
145:     T = alloc_darray2(my_nw, NH);
146:     T_new = alloc_darray2(my_nw, NH);

```

ソースリスト 4 (pfdm2d.c)

```

1: /*
2:  * Filename: pfdm2d.c
3:  * Description: 2D FDM for Multi-CPU.
4:  * Writer: Yasunari Mimura
5:  * Date: 1999.9.10
6:  */
7:
8: #include <stdlib.h>
9: #include <stdio.h>
10: #include <math.h>
11: #include <pvm3.h>
12:
13: extern int *alloc_int(size_t n);
14: extern double *alloc_double(size_t n);
15: extern double **alloc_darray2(size_t nrow, size_t ncol);
16: extern void free_darray2(double **m);
17:
18: #define TID_MSG 0
19: #define T_LR_MSG 1
20: #define T_RL_MSG 2
21: #define Q_MSG 3
22:
23: int nlimit = 65000;
24:
25: #define NW 81
26: #define NH 41
27:
28: int my_nw;
29: double h = 0.25; /* dx = dy = h */
30: double dt = 1000.0; /* sec */
31:

```

```

147: T_left = alloc_double(NH);
148: T_right = alloc_double(NH);
149: buf = alloc_double(NH);
150:
151: /* Init. Temp. */
152: for(i=0 ; i<my_nw ; i++) {
153:   for(j=0 ; j<NH ; j++) {
154:     T[i][j] = temp_low;
155:     T_new[i][j] = temp_low;
156:   }
157: }
158: for(j=0 ; j<NH ; j++) {
159:   T_left[j] = temp_low;
160:   T_right[j] = temp_low;
161: }
162:
163: t = 0.0;
164:
165: if( my_i == 0 ) {
166:
167:   for(j=0 ; j<h_mid+1 ; j++) {
168:     T[0][j] = temp_high;
169:     T_new[0][j] = temp_high;
170:   }
171:
172:   calc_Qin();
173:   q_out = 0.0;
174:   dq = q_in - q_out;
175:   printf("t = %f\n", t);
176:   printf("Q_in = %e, Q_out = %e, dq = %e\n",
177:     q_in, q_out, dq );
178: }
179:
180: for(count=1 ; count<=nlimit ; count++) {
181:   t += dt;
182:
183:   /* Left boundary */
184:   if( my_i == 0 ) {
185:     i = 0;
186:     for(j=h_mid+1 ; j<NH-1 ; j++) {
187:       Tw = T[i+1][j]; /* insulation */
188:       Te = T[i+1][j];
189:       Ts = T[i][j-1];
190:       Tn = T[i][j+1];
191:       T_new[i][j] = gamma*(Tw + Te + Ts + Tn - 4.0*T[i][j]) + T[i][j];
192:     }
193:
194:     /* corner */
195:     i = 0;
196:     j = NH-1;
197:     Tw = T[i+1][j]; /* insulation */
198:     Te = T[i+1][j];
199:     Ts = T[i][j-1];
200:     Tn = T[i][j-1]; /* insulation */
201:     T_new[i][j] = gamma*(Tw + Te + Ts + Tn - 4.0*T[i][j]) + T[i][j];
202:
203:   } else {
204:     i = 0;
205:
206:     /* Lower boundary */
207:     j = 0;
208:     Tw = T_left[j]; /* left side data */
209:     Te = T[i+1][j];
210:     Ts = T[i][j+1]; /* insulation */
211:     Tn = T[i][j+1];
212:     T_new[i][j] = gamma*(Tw + Te + Ts + Tn - 4.0*T[i][j]) + T[i][j];
213:
214:     for(j=1 ; j<NH-1 ; j++) {
215:       Tw = T_left[j]; /* left side data */
216:       Te = T[i+1][j];
217:       Ts = T[i][j-1];
218:       Tn = T[i][j+1];
219:       T_new[i][j] = gamma*(Tw + Te + Ts + Tn - 4.0*T[i][j]) + T[i][j];
220:     }
221:
222:     /* Upper boundary */
223:     j = NH-1;
224:     Tw = T_left[j]; /* left side data */
225:     Te = T[i+1][j];
226:     Ts = T[i][j-1];
227:     Tn = T[i][j-1]; /* insulation */
228:     T_new[i][j] = gamma*(Tw + Te + Ts + Tn - 4.0*T[i][j]) + T[i][j];
229:
230:   }
231:
232:   /* Right boundary */
233:   if( my_i == (nproc-1) ) {
234:     i = my_nw-1;
235:     for(j=1 ; j<h_mid ; j++) {
236:       Tw = T[i-1][j];
237:       Te = T[i-1][j]; /* insulation */
238:       Ts = T[i][j-1];
239:       Tn = T[i][j+1];
240:       T_new[i][j] = gamma*(Tw + Te + Ts + Tn - 4.0*T[i][j]) + T[i][j];
241:     }
242:
243:     /* corner */
244:     i = my_nw-1;
245:     j = 0;
246:     Tw = T[i-1][j];
247:     Te = T[i-1][j]; /* insulation */
248:     Ts = T[i][j+1]; /* insulation */
249:     Tn = T[i][j+1];
250:     T_new[i][j] = gamma*(Tw + Te + Ts + Tn - 4.0*T[i][j]) + T[i][j];
251:
252:   } else {
253:     i = my_nw-1;
254:
255:     /* Lower boundary */
256:     j = 0;
257:     Tw = T[i-1][j];
258:     Te = T_right[j]; /* right side data */
259:     Ts = T[i][j+1]; /* insulation */
260:     Tn = T[i][j+1];
261:     T_new[i][j] = gamma*(Tw + Te + Ts + Tn - 4.0*T[i][j]) + T[i][j];
262:
263:     for(j=1 ; j<NH-1 ; j++) {
264:       Tw = T[i-1][j];
265:       Te = T_right[j]; /* right side data */
266:       Ts = T[i][j-1];
267:       Tn = T[i][j+1];
268:       T_new[i][j] = gamma*(Tw + Te + Ts + Tn - 4.0*T[i][j]) + T[i][j];
269:     }
270:
271:     /* Upper boundary */
272:     j = NH-1;
273:     Tw = T[i-1][j];
274:     Te = T_right[j]; /* right side data */
275:     Ts = T[i][j-1];
276:     Tn = T[i][j-1]; /* insulation */
277:     T_new[i][j] = gamma*(Tw + Te + Ts + Tn - 4.0*T[i][j]) + T[i][j];
278:
279:   }
280:
281:   /* Upper boundary */
282:   j = NH-1;
283:   for(i=1 ; i<my_nw-1 ; i++) {
284:     Tw = T[i-1][j];
285:     Te = T[i+1][j];
286:     Ts = T[i][j-1];
287:     Tn = T[i][j-1]; /* insulation */
288:     T_new[i][j] = gamma*(Tw + Te + Ts + Tn - 4.0*T[i][j]) + T[i][j];
289:
290:   }
291:
292:   /* Lower boundary */
293:   j = 0;
294:   for(i=1 ; i<my_nw-1 ; i++) {
295:     Tw = T[i-1][j];
296:     Te = T[i+1][j];
297:     Ts = T[i][j+1]; /* insulation */
298:     Tn = T[i][j+1];
299:     T_new[i][j] = gamma*(Tw + Te + Ts + Tn - 4.0*T[i][j]) + T[i][j];
300:
301:   }
302:
303:   /* Calc. inner temp. */
304:   for(i=1 ; i<my_nw-1 ; i++) {
305:     for(j=1 ; j<NH-1 ; j++) {
306:       Tw = T[i-1][j];
307:       Te = T[i+1][j];
308:       Ts = T[i][j-1];
309:       Tn = T[i][j+1];
310:       T_new[i][j] = gamma*(Tw + Te + Ts + Tn - 4.0*T[i][j]) + T[i][j];
311:     }
312:
313:     /* Exchange data */
314:     if( my_i != nproc - 1 ) {
315:       /* send to right side */
316:       for(j=0 ; j<NH ; j++)
317:         buf[j] = T[my_nw-1][j];
318:       pvm_initsend(PvmDataRaw);
319:       pvm_pkdouble( buf, NH, 1);
320:       pvm_send( tids[my_i+1], T_LR_MSG);
321:     }
322:
323:     if( my_i != 0 ) {
324:       /* receive from left side */
325:       pvm_recv( tids[my_i-1], T_LR_MSG);
326:       pvm_upkdouble( T_left, NH, 1);
327:     }
328:
329:     if( my_i != 0 ) {
330:       /* send to left side */
331:       pvm_initsend(PvmDataRaw);
332:       for(j=0 ; j<NH ; j++)
333:         buf[j] = T[0][j];
334:       pvm_pkdouble( buf, NH, 1);
335:       pvm_send( tids[my_i-1], T_RL_MSG);
336:     }
337:
338:     if( my_i != nproc - 1 ) {
339:       /* receive from right side */
340:       pvm_recv( tids[my_i+1], T_RL_MSG);
341:       pvm_upkdouble( T_right, NH, 1);
342:     }
343:
344:     if( count%100 == 0 ) {
345:       if( my_i == 0 ) {
346:         calc_Qin();
347:         pvm_recv( tids[nproc-1], Q_MSG);
348:         pvm_upkdouble( &q_out, 1, 1);
349:         dq = q_in - q_out;
350:         printf("t = %e\n", t);
351:         printf("Q_in = %e, Q_out = %e, dq = %e\n",
352:           q_in, q_out, dq );
353:       } else if( my_i == nproc - 1 ) {
354:         calc_Qout();
355:         pvm_initsend(PvmDataRaw);
356:         pvm_pkdouble( &q_out, 1, 1);
357:         pvm_send( tids[0], Q_MSG);
358:       }
359:     }
360:
361:     /* write data */
362:     /*-----*/
363:     if( (count%100) == 0 ) {
364:       /* Print Temp. !! */
365:       /*-----*/
366:
367:       pvm_barrier("FDM2D", nproc);
368:
369:       /* Update temp. */
370:       for(i=0 ; i<my_nw ; i++) {
371:         for(j=0 ; j<NH ; j++) {
372:           T[i][j] = T_new[i][j];
373:         }
374:       }
375:
376:       free_darray2(T);

```

```

377:  free_darray2(T_new);
378:  free(T_right);
379:  free(T_left);
380:  free(buf);
381:  free(tids);
382:
383:  /* Check program finished, leave group, exit pvm */
384:  pvm_barrier("FDM2D", nproc);
385:  pvm_lvgroup("FDM2D");
386:  pvm_exit();
387:  return(0);
388: }

```

ソースリスト 5 (alloc_f.c)

```

1: /*
2:  Filename: alloc_f.c
3:  Description: Functions for memory allocation.
4:  Writer: Yasunari Mimura
5:  Date: 1999.9.10
6: */
7:
8: #include <stdlib.h>
9: #include <stdio.h>
10:
11: int *alloc_int(size_t n)
12: {
13:     int *pt;
14:     size_t ni, i;
15:
16:     ni = n*sizeof(int);
17:     if(( pt = (int *)malloc(ni) ) == NULL ) {
18:         fprintf(stderr, " error malloc at alloc_int. ");
19:         exit(1);
20:     }
21:
22:     for( i=0 ; i<n ; i++)
23:         *(pt + i) = 0;
24:
25:     return pt;
26: }
27:
28: double *alloc_double(size_t n)
29: {
30:     double *pt;
31:     size_t ni, i;
32:
33:     ni = n*sizeof(double);
34:     if(( pt = (double *)malloc(ni) ) == NULL ) {
35:         fprintf(stderr, " error malloc at alloc_dbl. ");
36:         exit(1);
37:     }
38:
39:     for( i=0 ; i<n ; i++)
40:         *(pt + i) = 0.0;
41:
42:     return pt;
43: }
44:
45:
46: double **alloc_darray2(size_t nrow, size_t ncol)
47: {
48:     size_t i;
49:     double **m;
50:
51:     m = (double**)malloc( sizeof(double*) * nrow );
52:     if( m == NULL ) {
53:         fprintf(stderr, " error malloc at alloc_darray2. ");
54:         exit(1);
55:     }
56:
57:     m[0] = (double*)malloc( sizeof(double) * nrow * ncol );
58:     if(m[0] == NULL) {
59:         fprintf(stderr, " error malloc at alloc_darray2. ");
60:         exit(1);
61:     }
62:
63:     for( i=1 ; i<nrow ; i++)
64:         m[i] = m[i-1] + ncol;
65:
66:     for( i=0 ; i < (nrow*ncol) ; i++)
67:         *(m[0] + i) = 0.0;
68:
69:     return m;
70: }
71:
72:
73: void free_darray2(double **m)
74: {
75:     if( m[0] ) free(m[0]);
76:     if( m ) free(m);
77: }
78:

```

並列化手法のご紹介と，クラスタを組む 上での各種 Tips

住商エレクトロニクス株式会社 関西支社 技術部

柴崎 勝憲

shiba@osaka.sse.co.jp

■ はじめに

一般に並列プログラム，パラレルプログラムと呼ばれているものは，大きく分けて2つのものがあります。一つは，世の中一般の事象をそのまま抽象化（場合によっては並列性を抜き出して抽象化）し，並列コンピュータ上に実装してしまうもの，ここではこれを並列プログラムと呼ぶことにします。この方法の利点は，世の中の事象をそのまま，並列コンピュータ上に実装してしまうため，精度上のロスが少なかったりと，実際の事象に非常に近いプログラムが組めます。

ただし，この方法は大元の事象を抽象化することからはじめなければなりません。そのため，この方法でプログラムを組めるプログラマは非常に限られています。{(並列であろうとなかろうと) プログラムを組む上で重要なのは，大元の事象を抽象化する能力であり，並列に関する技術や知識は枝葉でしかないと思います。しかし，大元の事象を抽象化するというのは非常に難しく，IT 革命とか高度情報化社会とかいわれている割には，いまだに個人のセンスに依存している部分が多々あります。} 逆に言えば，大元の事象を正確に理解しているのであれば，並列プログラムはあまり難しく組めるのではないかと考えます。{並列プログラムの基礎など，そんなに難しくなく習得できます。}

もう一つは，既に1 CPUで動作しているプログラムを，並列コンピュータ上に移植してしまうことです。この時に，複数のCPUで効率よく動作するようプログラムし直すことを並列化プログラムと呼ぶことにします。

今回，ここでは，この並列化プログラムを取り上げます。また，その際に実際にどのようなプログラムを組めば良いのか理解しやすいように，単純化したプログラムではありますが，OpenMP と MPI を使用して組んだプログラムを提示いたします。

■ テスト 環境

☐ OpenMP テスト 環境

CPU: Pentium 2 266MHz x 2
Memory: 128MB
OS: Kondara MNU/Linux 1.2
C 言語: gcc 2.95.2
OpenMP: Omni Compiler 1.2

☐ MPI テスト 環境

CPU: Celeron 566MHz
Memory: 128MB
OS: Kondara MNU/Linux 1.2
C 言語: gcc 2.95.2
MPI: LAM 6.3.2

■ 並列化の実際

並列化プログラムと一口に言いまして，非常に多岐にわたっていますので，すべてをここでご説明することは不可能かとは思いますが，ここでその一端だけでも分かっていただければ幸いです。

まずは，並列化プログラムの中でも特にデータ並列（for ループの並列化）に関してご説明いたします。

□ パターン 1

データの入力量が 1（定数）に対して，データの出力量が N の場合

例)

レイトレーシング

マンデルブロート集合

この場合は，とても簡単に並列化が行えます。つまり，出力データを得るための計算に関して依存性がまったくないため，単純に複数 CPU に計算を割り当てることができます。

並列化前のモデルプログラムは test1.c

OpenMP を使用した並列化プログラムは test1-omp.c

MPI を使用した並列化プログラムは test1-mpi.c

を参照してください。

□ パターン 2

データの入力量が N に対して，データ出力量が N の場合

例)

画像処理や流体等 数値演算のほとんど？

この場合はそれぞれの出力データを得るために必要となるデータをあらかじめ，複数の CPU に割り当てる必要があります。また，モデルプログラムでは出力データ $out[i]$ を得るために必要となるデータは $in[i - 0], in[i], in[i + 1]$ です。そのため，ある CPU に割り当てられた計算すべき出力データが 1 から m の範囲であれば，その CPU が必要とするデータは $in[l - 1]$ から $in[m + 1]$ までのデータとなります。ちなみに出力データの範囲よりはみ出た入力データ部分を俗に ”のりしろ” と呼び他のデータの範囲と区別する場合があります。これは，このようにはみ出たデータに関して特別な処理をしなければならない（ことが多い）ため，このように呼び区別するからです。

並列化前のモデルプログラムは test2.c

OpenMP を使用した並列化プログラムは test2-omp.c

MPI を使用した並列化プログラムは test2-mpi.c

を参照してください。

□ パターン 3（そのままでは並列化できないもの）

データの入力量が 1（定数）に対して、データの出力量が 1（定数）の場合

この様な例の場合は、このまま for ループを並列化することはできません。そのため、並列化を行う場合は、calc 関数の中身を吟味し並列性を抜き出す事からはじめる必要があります。

モデルプログラムとして test3.c

を参照してください。

手持ちのプログラム中にこの様な個所のあるプログラムをどうすれば並列化できるのでしょうか？私であれば、まずこの部分に関しては ”後回しにして別の部分から並列化を行う ”という手段をとります。もちろん、あらかじめプロファイル等によりこの部分の計算量が極めて大きい場合には、この様な手段はとれませんが、無視できるほど小さくはないが、計算量が最も大きい部分でもない場合には、後回しにしたほうが効率が良いでしょう。この理由は、以下のとおりです。

並列化を行うと、場合により遊んでしまう CPU が発生してきます。その場合、並列化が困難な部分を遊んでしまっている CPU にすべて分担させてしまえば遊んでいる CPU を活かすことができます。もちろん、無視できるほど小さいのであれば、当然ながら無視してしまいましょう。ただ、無視する場合には注意しなければならないことが一つあります。それは、この部分に関して 1 つの CPU で計算し結果を他の CPU に送るようなプログラムを組んでしまうことです。

少し考えてみましょう...

どうせ、この部分を 1 つの CPU が計算しているときに他の CPU は何もせずに遊んでいるわけですから、であれば、遊ばさずに同じ計算をさせましょう。一見、無駄に見えますがこのようにすべて CPU での計算を行っていれば、計算結果を他の CPU に送る必要がなくなるため通信時間を稼ぐことができます。結果として無駄に計算したにもかかわらず、速度が向上するのです。ついでに、通信が入らないためプログラムが簡単になるというメリットもあります。ただし、これは少なくとも使用している CPU に関しては独占的に使用できるという前提の基の話です。

□ モデルプログラムについて

モデルプログラムをご覧になられて、

”こんな単純なプログラムを並列化するのではなく、もっと現実にそくした複雑なプログラムを並列化するにはどうしたらいいのか？”

と思われる方も多いと思います。ですが、実際にプログラムを記述しそれを利用する場合には、単純なほうが良いのは当たり前です。もし、複雑なプログラムをそのまま並列化してしまえば、いったい誰がメンテナンスできるというのでしょうか？おそらく、プログラムを並列化した本人でさえメンテナンス出来ないのではないのでしょうか？

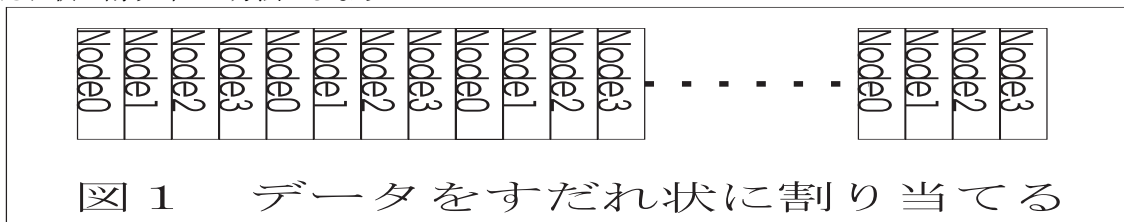
そのため プログラムを並列化する場合には、いきなり並列化を行うのではなく、並列化しやすいようにプログラムの構成を変更することから始めてください。もっと言えば、最終的に並列化を行うことを考えているのであれば、できるだけ並列化しやすいよう単純化したプログラムを組むように心がけてください。（もっとも、プログラムの並列化よりも並列化しやすいようにプログラムを単純化するほうがよほど難しいのですが...）こうしておけば、（さらに並列化効率を求めなければ）並列化など元のプログラムそのものを組むことにくらべれば、まったく簡単なことです。

■ ほんの少し高度な並列化

ここまで、データ並列について非常に簡単に記述してきましたが、こんな単純な並列化で速度は稼げるのでしょうか？実は、こんな単純な並列化でも結構速度は稼げるのです。しかし、さらに上を目指して速度向上を計ろうと思えば、やはり、もう少し手の込んだ並列化が必要になってきます。ここでは、その中でも比較的簡単にできるデータの割り当て方を変えることにより、速度向上をねらってみます。また、ここで対象とするプログラムはパターン 1 のモデルプログラムとします。そして、このような手法がどの程度の速度向上につながるかは、結果がわかりやすいようもっと計算量が多い別のプログラムで示します。

□ データをすだれ状に割り当てる

データの割り当て方法を変更すると記述しましたが、さてどのようにデータを割り当てれば良いのでしょうか？単純に先頭から CPU 数で等分しただけのデータ量を各 CPU 毎に割り当てるのがもっとも簡単です。（先の並列化モデルプログラムでもそのようにプログラムしています。）しかし、もし各データの要素ごとに計算量が異なっている場合はどうでしょうか？そうです、それぞれの CPU 毎に計算時間がまちまちになり、ロスが発生するのです。これでは、並列化性能の向上は期待できません。では、どの様にデータを割り振れば良いのでしょうか？もっとも簡単な方法は 以下のようにデータをすだれ状に割り当てる方法でしょう。



まずは、この方法に従った並列化モデルプログラムを示します。

並列化前のモデルプログラムは test1.c

OpenMP を使用した並列化プログラムは test1-omp-sudare.c

MPI を使用した並列化プログラムは test1-mpi-sudare.c

を参照してください。

□ データを動的に割り当てる

先のデータをすだれ上に割り当てる方法より、高度な方法として、データを動的に割り当てるという方法があります。

この方法はまず、それぞれの CPU に割り当てるデータ量を少なくします。そして、各 CPU に少しずつデータを渡し、計算が終了した CPU に対して改めてデータを割り当てるという方法です。これだと、ほぼ CPU をあまらせることなく計算が行なえるでしょう。また、各 CPU の性能がまちまちであっても、性能の高い CPU が性能の低い CPU に引きずられることなく、性能の高い CPU もそうでない CPU も性能を活かすことができます。

では、この方法を使ってモデルプログラムを並列化してみましょう。

並列化前のモデルプログラムは test1.c

OpenMP を使用した並列化プログラムは test1-omp-dynamic.c

MPI を使用した並列化プログラムは `test1-mpi-dynamic.c`

を参照してください。

□ 結局どの手法が良いのか？

ここまで、いろいろな並列化手法を記述してきました。実際にプログラムを並列化する場合は、データを動的に割り当てる方法がメリットも多くよさそうです。

しかし、いつもいつもこの方法が良いかというと...

そうでもありません、大きな問題としてプログラムが他の方法に比べ複雑になってしまうという問題があります。また、速度的な問題として通信や管理のためのオーバーヘッドが大きくなるというデメリットがあります。また、このオーバーヘッドは管理する CPU に集中するため、場合によっては管理用の CPU を占有してしまう場合もあるでしょう。

さらに、オーバーヘッドが大きくなると..

そうです、管理用の CPU が 1 つでは足りずに 2 つ 3 つと必要になる場合もあります。最も良いのは、上記のような様々な方法をそれぞれのプログラムと動作システムごとに検証してみることでしょう。面倒なことではありますが、速度を向上させようと思えば手間がかかってしまうということです。

■ 閑話休題

今回のテーマは並列化手法のご紹介ということでしたが、実際には並列化プログラムの実際という内容になってしまいました。また、MPI での並列化を中心に記述してきたのですが、記述すればするほど、MPI よりも OpenMP の方がプログラムが簡単ということがはっきりわかる結果となりました。一応、MPI のプログラムを組んだ後、OpenMP のプログラムを組んだのですが...

皆さんも既にお気づきでしょうが、(MPI にくらべると) ほとんど並列化らしきものは何もすることなく出来上がってしまいました。私自身、MPI のプログラムに慣れていない (メッセージパッシング型の並列化は慣れているのですが、MPI のライブラリに慣れていないという意味です。) ため、並列化に時間がかかったという面もありますが、それにしてもこのプログラミングの容易さや自由度の高さは非常に大きな OpenMP のメリットでしょう。もし可能であれば (そして十分に高速であるなら)、MPI でプログラムを組むより、OpenMP でプログラムを組みたいところです。でも...

残念ながら、OpenMP は共有メモリアイプの並列計算機でしか利用できません。

そのため、安価なクラスタタイプの並列計算機では基本的に利用することができません。

しかし、幸いなことに今回利用した RWCP Omni OpenMP コンパイラ version 1.2 は (ベータリリースではありますが) SCore Cluster System Software の API レベルでの分散共有メモリである SCASH に対応しているのです。

そのため、Score + Omni Compiler であれば、安価なクラスタタイプの並列計算機上でも OpenMP でプログラムを組むことが可能であるということです。

もちろん、最高速を求めたい場合は MPI (などのメッセージパッシング型) で並列化を行わないといけないでしょうから、MPI をマスターしておいて無駄になることは絶対にないでしょう。

■ クラスタを組む上での Tips

ここまで、プログラムの並列化ということで記述してきましたが、速度を求めるにはやはりシステムそのものの性能や使い勝手も重要です。ここでは、私が実際にお客様に納入させていただいた際に気づいた点やトラブルに基づく Tips を記述いたします。

□ ネットワークの2重化

クラスタを組む場合、おそらくほとんどの方が少なくとも各ユーザのホームディレクトリ（場合によってはシステム全部）は NFS で共有することでしょう。これは、各マシン間でファイルシステムを共有しないとプログラム起動時に一々バイナリを総てのマシンにコピーしなければならないなどかなり面倒なためです。

ここで、考えてみてください...

NFS で使用しているのは Ethernet

MPI で使用しているのも Ethernet

ただでさえ、Ethernet の速度が遅いから Gigabit Ether や Myrinet 等高速なデバイスを使用するケースが多いのにさらに NFS 等で帯域を圧迫して速度が出るものでしょうか？

もちろん、データ転送にあまり負荷のかからないものや、プログラム内でファイルアクセスなど皆無という場合にはあまり問題にはならないかもしれませんが、しかし、快適なシステムを構成する上ではやはり MPI の通信ラインと NFS 等 IP のラインは分けておいた方がトータルの性能としては向上するのは自明です。これは、もちろん Myrinet や GigabitEther 等高速デバイスを使用する場合にもあてはまりますので、試してみてもいいでしょうか？

□ ディスクレス Linux

ディスクレス Linux でクラスタを組む場合、皆さんはどの様にして shutdown をかけているのでしょうか？何も考えずに、電源を切っているだけでしょうか？もちろん、これでも特に問題になることはないでしょう。しかし、台数が増えてくると結構面倒ですね？また、電源を切らなければならない場合に実機の前まで行かないといけないというデメリットもあります。ということで、正常にリブート処理やシャットダウン処理を行なう方法を考えてみましょう。（正常にシャットダウンできれば、APM の自動電源 OFF の機能により自動的に電源が切れます。）結局は、終了処理の途中で NFS の機能が切れてしまい、ファイルシステムにアクセスできないため正常にシャットダウン処理が出来ずにエラーで落ちてしまうわけです。ですから、終了処理の中の NFS にかかわる機能を OFF にするところを削除してしまえばこの問題は起きません。実際には、

```
/etc/rc.d/rc0.d/K90network  
/etc/rc.d/rc6.d/K90network  
/etc/rc.d/rc0.d/K75netfs  
/etc/rc.d/rc6.d/K75netfs
```

を削除してしまえば良いのです。（RedHat 6.0 系統の Linux の場合）もちろん、ディストリビューションによってはこれだけではだめな場合もありますが概ねこれで大丈夫でしょう。

ちなみに Kondara MNU/Linux 1.1 の場合は上記に加え

```
/etc/rc.d/rc0.d/S00killall を /etc/rc.d/rc0.d/K97killall に変更  
/etc/rc.d/rc0.d/S01reboot を /etc/rc.d/rc0.d/S00reboot に変更  
/etc/rc.d/rc6.d/S00killall を /etc/rc.d/rc6.d/K97killall に変更  
/etc/rc.d/rc6.d/S01reboot を /etc/rc.d/rc6.d/S00reboot に変更
```

と4つのファイルの名前を変更する必要があります。当然ながらこれらはディスクレスで動作させているマシンでのみ行なってください。

□ Wake on Lan

上記の設定により、自動的に電源 OFF を行なう方法はわかりました。では、自動的に（ネットワークで）電源を立ち上げる方法はないでしょうか？そうです、マジックパケット (Wake on Lan) を用いて電源をネットワークから立ち上げることが出来ます。もちろん、これを用いるには電源（ケース）やマザーボード、ネットワークカードが対応している必要があります。しかし、今からクラスタを構築するのであれば、そんなに苦労せずに構成を組むことが出来るでしょう。注意していただきたいのは、この機能はあくまで電源を ON にする機能しかないということです。OS の起動等は自動的には行ないませんので、利用する方自身で考える必要があります。

注意点：ネットワークからの電源の制御は場合によりシステムの安全性に問題を引き起こす場合がありますので、充分注意してください。

□ Linux カーネル ネットワークドライバ

Linux カーネルは様々な Ethernet カードに対応しています。しかし、それぞれの Ethernet カードごとに複数のドライバが存在する場合があることはあまり知られていません。そして、複数存在するドライバは、カードによっては正常に動作するものしないもの、速度が出るもの出ないものと動作が異なる場合が良くあります。これにより非常に困ったことが起こり得ます。それは...

Linux の各種ディストリビューションをそのままインストールすると、最初に認識したドライバを使用するため、それがカードにとって相性が良いか悪いかはわかりません。

つまり、まったく動作しなかったり、動作しても速度が出ない場合があるのです。このような現象が出た場合は、使用するドライバを変更してみましょう。

ちなみにドライバが複数あるカードは

Intel Etherexpress PRO 10/100 シリーズ（2種類）
DEC 21x4x チップを使用した Ethernet カード（3種類）

です。特に DEC のチップセットを使用した Ethernet カードはドライバの違いによる性能差が大きいのので注意してください。

■ 参考文献

- 1) 新情報処理開発機構
並列分散システムパフォーマンスつくば研究室
RWCP Omni OpenMP コンパイラ version 1.2
<http://pdplab.trc.rwcp.or.jp/pdperf/home.ja.html>
- 2) 新情報処理開発機構
並列分散システムソフトウェアつくば研究室
SCore Cluster System Software 3.1
<http://pdswww.rwcp.or.jp/>
- 3) Laboratory for Scientific Computing
Department of Computer Science and Engineering
University of Notre Dame
LAM 6.3.2
<http://www.lsc.nd.edu/>
- 4) Kondara Project
Kondara MNU/Linux 1.2
<http://www.kondara.org/>
- 5) Myrinet
Myricom, Inc.
<http://www.myri.com/>

```
モデルプログラム test1.c
#define N

int  calc(int a);

main(int argc, char *argv[])
{
    int  *out, i;

    out  = (int *)calloc(N, sizeof(int));

    for (i = 0; i < N; i++) {
        out[i]  = calc(i);
    }
}
```

```
OpenMP モデルプログラム test1-omp.c
#define N

int  calc(int a);

main(int argc, char *argv[])
{
    int  *out, i;

    out  = (int *)calloc(N, sizeof(int));

    #pragma omp parallel shared(out) private(i)
    {
        #pragma omp for schedule(static)
        for (i = 0; i < N; i++) {
            out[i]  = calc(i);
        }
    }

    free(out);
}
```

```
MPI モデルプログラム test1-mpi.c
#include <mpi.h>

#define N

int calc(int a);

main(int argc, char *argv[])
{
    MPI_Status status;
    int node_id, node_size;
    int start, end;

    int *out, i;
    out = (int *)calloc(N, sizeof(int));

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &node_id);
    MPI_Comm_size(MPI_COMM_WORLD, &node_size);

    start = node_id * (N / node_size);
    end = (node_id + 1) * (N / node_size);

    for (i = start; i < end; i++) {
        out[i] = calc(i);
    }

    if (node_id == 0) {
        for (j = 1; j < node_size; j++) {
            MPI_Recv(out + j * N / node_size, N / node_size * sizeof(int), MPI_INT, j, 100, &status);
        }
    }
    else {
        MPI_Send(out + start, N / node_size * sizeof(int), MPI_INT, 0, 100);
    }

    MPI_Finalize();

    free(out);
}
```

```
モデルプログラム test2.c
#define N

void Data_read(int *a);
int calc(int a, b, c);

main(int argc, char *argv[])
{
    int i;
    int *in;
    int *out;

    in = (int *)calloc(N, sizeof(int));
    out = (int *)calloc(N, sizeof(int));

    Data_read(in);

    out[0] = calc(0, in[0], in[1]);
    for (i = 1; i < N - 1; i++) {
        out[i] = calc(in[i - 1], in[i], in[i + 1]);
    }
    out[N - 1] = calc(in[N - 2], in[N - 1], 0);

    free(in);
    free(out);
}
```

```
OpenMP モデルプログラム test2-omp.c
#define N

void Data_read(int *a);
int calc(int a,b,c);

main(int argc,char *argv[])
{
    int i;
    int *in;
    int *out;

    in = (int *)calloc(N,sizeof(int));
    out = (int *)calloc(N,sizeof(int));

    Data_read(in);

    out[0] = calc(0,in[0],in[1]);
    #pragma omp parallel shared(in,out) private(i)
    {
        #pragma omp for schedule(static)
        for (i = 1;i < N - 1;i++) {
            out[i] = calc(in[i - 1],in[i],in[i + 1]);
        }
    }
    out[N - 1] = calc(in[N - 2],in[N - 1],0);

    free(in);
    free(out);
}
```

```
MPI モデルプログラム test2-mpi.c
#include <mpi.h>

#define N

void Data_read(int *a);
int calc(int a,b,c);

main(int argc,char *argv[])
{
    MPI_Status status;
    int node_id,node_size;
    int start,end;

    int i;
    int *in;
    int *out;

    in = (int *)calloc(N,sizeof(int));
    out = (int *)calloc(N,sizeof(int));

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&node_id);
    MPI_Comm_size(MPI_COMM_WORLD,&node_size);

    if (node_id == 0) {
        Data_read(in);
        for (j = 1;j < node_size;j++) {
            MPI_Send(in,N*sizeof(int),MPI_INT,j,10,MPI_COMM_WORLD);
        }
    }
    else {
        MPI_Recv(in,N*sizeof(int),MPI_INT,0,10,MPI_COMM_WORLD,&status);
    }

    start = node_id*(N/node_size);
    end = (node_id + 1)*(N/node_size);

    if (node_id == 0) {
        out[0] = calc(0,in[0],in[1]);
        start++;
    }
    if (node_id == node_size - 1) {
        out[N - 1] = calc(in[N - 2],in[N - 1],0);
        end--;
    }

    for (i = start;i < end;i++) {
        out[i] = calc(i);
    }

    if (node_id == 0) {
        for (j = 1;j < node_size;j++) {
            MPI_Recv(out + j*N/node_size,N/node_size*sizeof(int),\
                MPI_INT,j,100,MPI_COMM_WORLD,&status);
        }
    }
    else {
        MPI_Send(out + start,N/node_size*sizeof(int),MPI_INT,0,100,MPI_COMM_WORLD);
    }

    MPI_Finalize();

    free(in);
    free(out);
}
```

```
モデルプログラム 3 test3.c
#define N

void Data_read(int *a);
int calc(int a);

main(int argc, char *argv[])
{
    int a;
    int i;

    Data_read(&a);

    for (i = 0; i < N; i++) {
        a = calc(a);
    }
}
```

```
OpenMP モデルプログラム test1-omp-sudare.c
#define N

int calc(int a);

main(int argc, char *argv[])
{
    int *out, i;

    out = (int *)calloc(N, sizeof(int));

    #pragma omp parallel shared(out) private(i)
    {
        #pragma omp for schedule(static, 1)
        for (i = 0; i < N; i++) {
            out[i] = calc(i);
        }
    }

    free(out);
}
```

```
MPI モデルプログラム test1-mpi-sudare.c
#include <mpi.h>

#define N

int calc(int a);

main(int argc, char *argv[])
{
    MPI_Status status;
    int node_id, node_size;
    int start, end;

    int *out, i;
    out = (int *)calloc(N, sizeof(int));

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &node_id);
    MPI_Comm_size(MPI_COMM_WORLD, &node_size);

    for (i = node_id; i < N; i += node_size) {
        out[i] = calc(i);
    }

    if (node_id == 0) {
        for (j = node_id; j < N; j += node_size) {
            if (j % node_size == node_id) {
                continue;
            }
            MPI_Recv(out + j, sizeof(int), MPI_INT, j, 100, &status);
        }
    }
    else {
        for (j = node_id; j < N; j += node_size) {
            MPI_Send(out + j, sizeof(int), MPI_INT, 0, 100);
        }
    }

    MPI_Finalize();

    free(out);
}
```

```
OpenMP モデルプログラム test1-omp-dynamic.c
#define N

int calc(int a);

main(int argc, char *argv[])
{
    int *out, i;

    out = (int *)calloc(N, sizeof(int));

    #pragma omp parallel shared(out) private(i)
    {
        #pragma omp for schedule(dynamic, 1)
        for (i = 0; i < N; i++) {
            out[i] = calc(i);
        }
    }

    free(out);
}
```

```
MPI モデルプログラム test1-mpi-dynamic.c
#include <mpi.h>

#define N

int calc(int a);

main(int argc, char *argv[])
{
    MPI_Status status;
    int node_id, node_size;
    int start, end;

    int *out, i, buf;
    int end_flag = -1;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &node_id);
    MPI_Comm_size(MPI_COMM_WORLD, &node_size);

    if (node_id == 0) {
        out = (int *)calloc(N, sizeof(int));
        send_count = 0;
        for (i = 1; i < node_size; i++) {
            MPI_Send(&send_count, sizeof(int), MPI_INT, i, 10000, MPI_COMM_WORLD);
            send_count++;
        }
        for (i = 0; i < N; i++) {
            MPI_Recv(&buf, sizeof(int), MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            out[status.MPI_TAG] = buf;
            if (send_count < N) {
                MPI_Send(&send_count, sizeof(int), MPI_INT, status.MPI_SOURCE, 10000, MPI_COMM_WORLD);
                send_count++;
            }
            else {
                MPI_Send(&end_flag, sizeof(int), MPI_INT, status.MPI_SOURCE, 10000, MPI_COMM_WORLD);
                send_count++;
            }
        }
    }
    else {
        i = 0;
        MPI_Recv(&i, sizeof(int), MPI_INT, 0, 10000, &status);
        while(i >= 0) {
            buf = calc(i);
            MPI_Send(&buf, sizeof(int), MPI_INT, 0, i, MPI_COMM_WORLD);
            MPI_Recv(&i, sizeof(int), MPI_INT, 0, 10000, MPI_COMM_WORLD, &status);
        }
    }
    MPI_Finalize();

    free(out);
}
```

グローバルコンピューティング

– Condor と Globus Toolkit の紹介 –

大阪産業大学 工学部 情報システム工学科

小坂 隆浩

koita@ise.osaka-sandai.ac.jp

■ はじめに

グローバルコンピューティング^{1, 2, 3, 4)}のためのソフトウェアシステム Condor⁵⁾ とソフトウェアツール群 Globus Toolkit⁶⁾ を紹介する。

これまで、グローバルコンピューティングの研究は、世界中にあるコンピュータを使い、地球規模の大規模処理の実現を目指してきた。そのためのソフトウェアシステムやツール等も数多く開発されてきた。それらのソフトウェアシステムやツールは、物理的に分散された様々な性能・管理者の計算機をシームレスに扱うことを可能とし、PC クラスタ以上の大規模処理を提供できる可能性を秘めている。

一方、自分の周りにある多くの PC は、CPU がほとんど使われていないことが多い。演習室の PC、隣の研究室の PC、家にある PC 等、ほとんどの場合、PC に向かっている時間は長くても CPU の利用率は非常に低い。PC クラスタ利用者にとって、多くの PC の CPU がアイドル状態にあることは「もったいない」状態である。これらの PC をうまく活用することによって、アプリケーションによっては、PC クラスタ以上の処理が実現可能となる。

本講習では、自分の周りにある PC をうまく活用するために、グローバルコンピューティングのソフトウェアを紹介する。自分の周りにある多くの PC を無駄なく、誰にも迷惑をかけることなく、思う存分使うためのソフトウェアシステム Condor と、グローバルコンピューティングの構築に必要な基本的なサービスを提供するツール群 Globus Toolkit について、特長・インストール方法・使い方などを説明する。

□ もっと PC をタダで使いたい！

もっと PC を使いたい！PC クラスタを導入される方の多くは、少しでも多く、少しでも大きな処理を、少しでも短い時間で終わらせるために、誰もがそう願うのではないだろうか。予算があれば、すぐにでも PC クラスタ、あるいはサーバ機を購入すれば、この願いはかなえられる。しかし、それなりの PC クラスタやサーバ機を購入しようとする、それらの価格はそう安くはない。また、購入してもすぐに時代遅れのマシンとなってしまうことも少なくない。なんとかして、タダでもっと多くの PC は使えないのだろうか？

ふと自分の周りを見まわすと、実に多くの PC が遊んでいる。例えば、筆者の周りには、隣の研究室に二十数台、学科の演習室に百数十台、隣の学科の建物に百数十台の PC がある。これらの PC をどうにかして使うことができれば、少なくとも数百台の PC がタダで使えることになる。これらの PC は、情報端末としての利用率は高いかもしれないが、一般に CPU の利用率は低い。Bertil によると、一般的な PC のアイドル率は 69 % で CPU のアイドル率は 93 % と、ほとんど使われていない状態であることが多い⁷⁾。

しかし、他人の PC を使うとなると、自分専用の PC クラスタのように好き勝手には使えない。一時的な利用ならば、ディスクレスクラスタ⁸⁾として利用させてもらえるかもしれないが、長期的に利用させてもらうには、いろいろな問題が考えられる。例えば、使わせる側としては、「重くなるのはいやだ」、「セキュリティがしっかりしていないといやだ」、「ディスクはあんまり使わないで」、「PC から離れたときだけ使って」など、いろいろな要求がある。また、使う側としても、「プログラムは書き換えたくない」、「あちこちファイルがあるのはいやだ」、「管理が面倒くさいのはいやだ」など、いろいろな要求がある。

これまでの、PC クラスタ向けの多くのソフトウェアは、PC クラスタを占有して利用するには適しているが、周りにあるいろんな PC を使わせてもらうにいくつか問題が考えられる。そこで、グローバルコンピューティングのためのソフトウェアの利用を考える。

□ グローバルコンピューティング

グローバルコンピューティング は、地球規模で物理的に分散した PC・WS・並列計算機・スパコンなどを利用して、これまでにない大規模な処理をおこなうものである。グローバルコンピューティングを実現するための研究としては、ハイパフォーマンスを実現する技術と共に、さまざまな管理者の PC からスパコンまでをうまく利用する技術に関する研究もおこなわれている。

グローバルコンピューティングを実現するために重要な要素は、地域的なものを全然意識なくいいシームレス (seamless) であることと、高いトランスペアレンシ (transparency) である。これらのための技術は、グローバルコンピューティング実現のために重要であるが、同時に周りにあるさまざまな PC を活用するためにも有効である。

グローバルコンピューティングの研究プロジェクトは、シームレスで高いトランスペアレンシを実現するための多くのソフトウェアを公開している。Condor や Globus も、グローバルコンピューティングのためのソフトウェアであり、他の人の PC を使うために有効な手段を提供する。Condor は、多くの PC を無駄なく獲得し、利用するためのソフトウェアであり、Globus Toolkit は、グローバルコンピューティングの構築に必要なとされる基本的なサービスのソフトウェアツール群である。これらのソフトウェアを使うことにより、どんなことができるのか、そしてどんなことがうれしいのか、次に説明していく。

■ Condor

□ Condor とは

Condor は、University of Wisconsin-Madison の Condor Team により作成された、分散された PC を使って高スループットを実現するためのソフトウェアシステムである。Condor を使うことのメリットは、他人の PC を迷惑をかけることなく無駄なく利用できるということである。また、MPI¹⁾

¹⁾2000.8.25 現在、MPI 用のモジュールは公開されていない。Condor Team に問い合わせたところ、「もうすぐ release される」らしい。

や PVM などソースプログラムを変更することなく利用できる。Condor の特長としては、次の 2 点があげられる。

空いてる PC の獲得

ユーザがジョブを Condor に投入するとすると、Condor は空いている PC を見つけ、その PC でジョブを実行する。この時、その PC がなんらかの理由で利用できなくなった (PC の所有者が作業をはじめた or 計算機の負荷が重くなった) 場合、Condor はそのジョブを違う PC に移動させ、ジョブを再開させる。もし、適当な PC が見つからない場合、ジョブは適当な PC が見つかるまでサスペンドされる。このように、利用している PC の状態によってジョブを移動やサスペンドすることにより、PC 提供者に迷惑をかけることなく PC が利用できる。

PC の需給調停

Condor では、ClassAds (Classified Advertisements) という仕組みを用いて、PC 提供者と PC 利用者の需給調停を行っている。PC 提供者は、どれくらいの資源を、どんな状態の時に提供するか、詳細に指定することができる。また、PC 利用者も、どのような PC を、どのような優先順位で必要としているか指定することができる。これにより、提供者は自分の都合の良いときだけ PC を提供し、PC 利用者は欲しい PC を利用することができる。

□ Condor のインストール

以下に、簡単なインストール手順を示す。詳細については、「Condor Version 6.1.14 Manual」の Administrator's Manual の章を参照されたい。Condor のインストールは、condor_install コマンドを実行し、対話式におこなわれ、非常に簡単である。どの PC を管理用 PC (central manager) とするかだけを決めておけば、あとはほとんど default でインストールが完了する。但し、PVM や SMP を使う場合や、状態監視モニタなどを導入する場合、別途モジュールをインストールする必要がある。

利用可能な OS

- SGI IRIX 6.2 から 6.4
- Solaris 2.5.1 から 2.6
- Linux 2.0.X 以上 (libc5 or glibc20 以上)
- HP-UX 10.20
- Digital Unix 4.0
- WindowsNT

インストール手順

1. Condor の入手 (<http://www.cs.wisc.edu/condor/downloads/>)
バイナリのみがダウンロード可能
2. ユーザ condor の作成 (root)
3. Condor の展開

4. condor_install の実行
5. condor_config の編集 (必要ならば)
6. condor_master の実行
7. 起動スクリプト (condor.boot) のコピー

[インストール例]

```
$ tar xzvf condor-6.1.14-linux-x86-glibc21-dynamic.tar.gz
$ cd condor-6.1.14
$ su
Password:
# /usr/sbin/adduser condor
# ./condor_install
略
# vi /usr/local/condor/etc/condor_config
# ln -s /usr/lib/libbfd-2.9.1.0.25.so /usr/lib/libbfd-2.9.1.0.23.so
(古い libbfd を読みにいくので, link でごまかす)
# /usr/local/condor/sbin/condor_master
# ps ax | grep condor_
(central manager の場合)
16880 ?      S      0:00 /usr/local/condor/sbin/condor_master
16881 ?      S      0:00 condor_collector -f
16882 ?      S      0:00 condor_negotiator -f
16883 ?      R      0:01 condor_startd -f
16884 ?      S      0:00 condor_schedd -f
(central manager 以外の場合)
4600 ?      S      0:00 /usr/local/condor/sbin/condor_master
4601 ?      R      0:05 condor_startd -f
4602 ?      S      0:00 condor_schedd -f
# cp /usr/local/condor/etc/examples/condor.boot /etc/rc.d/init.d
# ln -s /etc/rc.d/init.d/condor.boot /etc/rc.d/rc5.d/S96condor
```

Condor により PC を提供する側では、設定ファイル (condor.config) により、どのように PC を提供するか、細かく設定することができる。キーボードの入力が何秒なくて、CPU 負荷がいくつ以下のときだけ PC を提供するというようなことが設定できる。例えば、下記の設定例の場合、CPU 負荷が 0.5 以上か、1 分以内にキーボード入力があった場合、ジョブはサスペンドされ、さらに 10 分経過しても CPU 負荷もキーボード状態も変化しない場合は、ジョブは違う PC に移動される。

[condor.config 設定例 (一部)]

```

MINUTE          = 60
HOURL           = (60 * $(MINUTE))
StateTimer      = (CurrentTime - EnteredCurrentState)
ActivityTimer   = (CurrentTime - EnteredCurrentActivity)
ActivationTimer = (CurrentTime - JobStart)
LastCkpt       = (CurrentTime - LastPeriodicCheckpoint)

## The JobUniverse attribute is just an int. These macros can be
## used to specify the universe in a human-readable way:
STANDARD        = 1
PVM              = 4
VANILLA         = 5

NonCondorLoadAvg = (LoadAvg - CondorLoadAvg)
BackgroundLoad   = 0.3
HighLoad         = 0.5
StartIdleTime    = 15 * $(MINUTE)
ContinueIdleTime = 5 * $(MINUTE)
MaxSuspendTime   = 10 * $(MINUTE)
MaxVacateTime    = 5 * $(MINUTE)

KeyboardBusy     = (KeyboardIdle < $(MINUTE))
CPU_Idle         = ($(NonCondorLoadAvg) <= $(BackgroundLoad))
CPU_Busy         = ($(NonCondorLoadAvg) >= $(HighLoad))
MachineBusy      = ($(CPU_Busy) || $(KeyboardBusy))

WANT_SUSPEND     : True
WANT_VACATE      : $(ActivationTimer) > 10 * $(MINUTE)
#WANT_SUSPEND_VANILLA : True
#WANT_VACATE_VANILLA  : True

START            : $(CPU_Idle) && KeyboardIdle > $(StartIdleTime)
SUSPEND          : $(MachineBusy)
CONTINUE         : $(CPU_Idle) && KeyboardIdle > $(ContinueIdleTime)
PREEMPT          : ( ($(ActivityTimer) > $(MaxSuspendTime)) && \
                    (Activity == "Suspended") ) || \
                    ( SUSPEND && (WANT_SUSPEND == False) )
KILL             : $(ActivityTimer) > $(MaxVacateTime)

```

□ Condor の使用例

Condor は、基本的に `condor_compile` と `condor_submit` の2つのコマンドで利用できる。サンプルプログラム (`test.c`) を用いて使用例を示す。サンプルプログラムは5分毎に、ジョブが実行されているホスト名と現在時刻を表示するプログラムである。

[サンプルプログラム (`test.c`)]

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <time.h>

int main()
{
    int dmy,i;
    char cur_name[255], before_name[255];
    time_t t;

    for(i=0;i<100;i++) {
        gethostname(cur_name, dmy); /* ホスト名の取得 */

        /* ホスト名が前と異なる場合はプロセスが移動されている */
        if(!strcmp(cur_name,before_name)) {
            printf("Moved from %s -> ",before_name);
        }

        /* 現在時刻とホスト名の表示 */
        time(&t);
        printf("Host name = %s : %s",cur_name,ctime(&t));
        strcpy(cur_name, before_name);

        /* 5分アイドル */
        sleep(300);
    }
}
```

まず、プログラムのコンパイルは `condor_compile` にておこなう。ソースコードに手を加えることなく、Condor で実行可能なチェックポイント機能が埋め込まれた実行ファイルができる。このため、実行ファイルのサイズは `gcc` などでもコンパイルした場合に比べて非常に大きなものとなる。ちなみに筆者の環境で、`test.c` をコンパイルすると、`gcc` で作成した実行ファイルは 12097 バイト、`condor_compile` で作成した実行ファイルは 2258687 バイトとなった。makefile を用いる場合は、リンカー (`ld`) を `condor` のものと置き換える必要がある (Manual 3.11.3 Full Installation of `condor_compile` 参照)。

次に、コンパイルした実行ファイルを `condor` に submit するための submit 記述ファイル (`test.cf`) を作成する。`test.cf` では、実行ファイル名などとともに、rank を記述することにより、優先的に利用

したい PC をランク付けすることができる。test.cf の場合、メモリ + mips 値の大きなものから優先的に割り当てを要求する。

[submit 記述ファイル (test.cf)]

```
Executable = test
Requirements = Memory >= 32 && OpSys == "Linux"
Rank = TARGET.Memory + TARGET.Mips
output = test.out
error = test.err
Queue
```

submit 記述ファイルを作成したら、condor_submit により、実行ファイルを condor に投入する。実行状態は、condor_q コマンドにより確認でき、システム全体の状態は、condor_status により確認できる。実行結果の出力は、submit 記述ファイルにある test.out に出力される。結果出力から pisces で実行されたファイルが taka-lab, keyton と移動していく様子がわかる。ここでは、Condor の動作を分かりやすくするため、pisces, taka-lab で、途中から人工的に負荷を重くした。それにより、ジョブがサスペンドされ、移動されている。

[実行例]

```
% condor_compile gcc test.c -o test
% condor_submit test.cf
Submitting job(s).
1 job(s) submitted to cluster 27.
% condor_q

-- Submitter: taka-lab.ise.osaka-sandai.ac.jp : <133.64.72.242:2144>
: taka-lab.ise.osaka-sandai.ac.jp
ID      OWNER      SUBMITTED      RUN_TIME ST PRI SIZE CMD
 27.0   condor      8/24 23:14    0+00:00:16 R  0   2.2  test

1 jobs; 0 idle, 1 running, 0 held

% condor_status
```

Name	OpSys	Arch	State	Activity	LoadAv	Mem	ActvtyTime
ishii.ise.osa	LINUX	INTEL	Unclaimed	Idle	0.000	61	0+03:33:56
keyton.ise.os	LINUX	INTEL	Unclaimed	Idle	0.000	61	0+00:39:15
moon.ise.osak	LINUX	INTEL	Unclaimed	Idle	0.000	61	0+01:18:40
newton.ise.os	LINUX	INTEL	Unclaimed	Idle	0.000	45	0+00:34:30
pisces.ise.os	LINUX	INTEL	Claimed	Busy	0.990	188	0+00:53:51
taka-lab.ise.	LINUX	INTEL	Claimed	Busy	0.300	119	0+00:00:11
triton.ise.os	LINUX	INTEL	Owner	Idle	3.000	61	0+00:31:46

	Machines	Owner	Claimed	Unclaimed	Matched	Preempting
	INTEL/LINUX	7	1	2	4	0
	Total	7	1	2	4	0

[結果出力]

```
Host name = pisces.ise.osaka-sandai.ac.jp : Fri Aug 25 00:04:52 2000
Host name = pisces.ise.osaka-sandai.ac.jp : Fri Aug 25 00:09:52 2000
Host name = taka-lab.ise.osaka-sandai.ac.jp : Fri Aug 25 00:24:41 2000
Host name = taka-lab.ise.osaka-sandai.ac.jp : Fri Aug 25 00:29:41 2000
Host name = taka-lab.ise.osaka-sandai.ac.jp : Fri Aug 25 00:34:41 2000
Host name = taka-lab.ise.osaka-sandai.ac.jp : Fri Aug 25 00:39:41 2000
Host name = keyton.ise.osaka-sandai.ac.jp : Fri Aug 25 00:50:21 2000
Host name = keyton.ise.osaka-sandai.ac.jp : Fri Aug 25 00:55:21 2000
Host name = keyton.ise.osaka-sandai.ac.jp : Fri Aug 25 01:00:21 2000
```

Condor では、25 個のコマンドが用意されているが、以下によく使われるコマンドと機能を示す。

condor_compile Condor 用実行ファイルのコンパイル・リンク

condor_submit Condor ジョブの投入

condor_status Condor Pool の PC の利用状態の表示

condor_q 投入されたジョブ状態の表示

condor_rm 投入されたジョブの削除

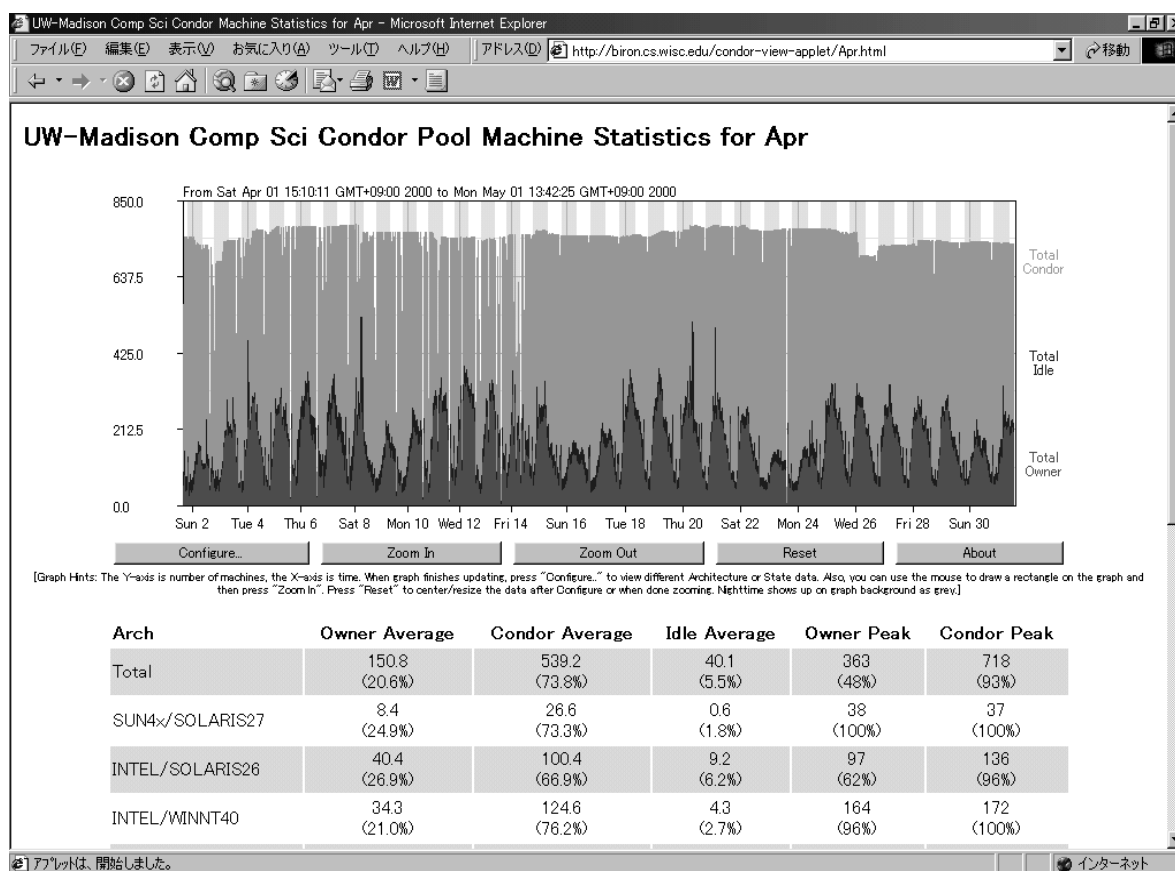


図 9.1: CondorView

Condor の PC 使用状況は、CondorView というモジュールをインストールすることにより、web ブラウザから確認できる。図 9.1 は、Condor の開発元である University of Wisconsin-Madison での PC の使用状況である。この図では、約 700 台以上の PC のうち、平均 500 台以上が Condor により利用されており、また PC が利用されているのは平均 150 台程度であることが示されている。このように、Condor を利用することにより、非常に多くの PC を余すことなく利用できるようになる。

□ Condor の問題点

Condor は、インストール・利用が簡単で、資源管理・獲得では非常にすぐれたソフトウェアであるが、いくつか問題点があげられる。

ソースの非公開

Condor は、バイナリですべてのソフトウェアが提供されている。バイナリで公開されているので、インストールは非常に容易ではあるが、実際自分なりにシステムをカスタマイズしようとしても限界がある。また、バグも多いようであるが、それを改良する手段がない。

MPI への対応

MPI への対応は、バージョン 6.1 以降のマニュアルには記述されているが、現在のところモジュールが公開されていない。PVM はジョブの動的な移動やサスペンドなど Condor の機能を十分に利用できるが、MPI はジョブ移動やサスペンドは今のところ対応していないらしい。

オーバーヘッド

Condor は、ジョブを移動するためのチェックポイントの埋め込みなどにより、実行のオーバーヘッドは少なくない。実行ファイルサイズも大きくなり、実行速度も遅くなる。また、各 PC の状態チェックも default では数分毎におこなわれるため、PC の状態が変化しても、すぐにジョブの移動はおこなえない。

ここでは、もっとも簡単に利用できるグローバルコンピューティングのソフトウェアシステム Condor を紹介した。次に、導入は少し複雑になるが、ソースが公開されており、もっと多くのサービスが利用可能な Globus Toolkit について説明する。

■ Globus Toolkit

□ Globus Toolkit とは

Globus Toolkit は、Globus プロジェクトにより作成された、グローバルコンピューティングシステムの構築に必要とされる基本的なサービスのツール群である。グローバルコンピューティングシステムを構築するためには、ユーザ認証、通信、リモートでのプロセス生成、情報サービスなど、さまざまな技術が必要となる。これまでは、それぞれが独自に、これらの技術を実装してきたが、Globus プロジェクトの発足により、Globus Toolkit を用いて上位のレベルのシステムを構築することが可能となった。例えば、通信やユーザ認証に Globus Toolkit を用いて MPICH を実装した MPICH-G や、Condor や LSF (Load Sharing Facility) へのジョブの投入を Globus Toolkit の認証や資源管理サービスなどを用いておこなうことができる。

Globus Toolkit のコアサービスは表 9.1 の通りである。これらのサービスを利用するために、それぞれのサービス毎にコマンドと API が用意されている。それらのコマンドと API を使うことにより、グローバルコンピューティングシステムの基本的なサービスを利用することができる。Condor の様に、インストールしたらすぐ何かができるという訳にはいかないが、グローバルコンピューティングシステムの何かを作りたい！自分でグローバルコンピューティングシステムを遊びたい！という方のために、ここでは簡単にインストール方法、利用例を紹介する。

サービス	名前	概要
資源管理	GRAM	リソースの割り当ておよびプロセスの生成
通信	Nexus	Unicast/Multicast 通信サービス
情報	MDS	システムの構造および状態に関する情報へのアクセス
セキュリティ	GSI	authentication などのセキュリティサービス
状態管理	HBM	システムの状況サービス
遠隔データアクセス	GASS	データへのリモートアクセスサービス
実行ファイル管理	GEM	実行ファイルの構築、キャッシングおよび配置

表 9.1: Globus Toolkit のコアサービス

□ Globus のインストール

利用可能な OS

- AIX 4.2.1 から 4.3.x
- IRIX 6.4 から 6.5
- Linux (Libc2 以上)
- Solaris 2.x, 7, 8
- FreeBSD 2.x から 3.x
- HP-UX 10.20 から 11.x
- UNICOS/mk 2.0.3.x

インストール手順

インストール手順の流れは、次のようになる。Condor に比べるとだいぶ手順が複雑である。詳細については、「Globus Toolkit 1.1.3 System Administration Guide」を参照にされたい。

1. SSLeay のインストール
(<ftp://ftp.psy.uq.oz.au/pub/Crypto/SSL/SSLeay-0.9.0b.tar.gz>)
2. OpenLDAP のインストール
(<ftp://ftp.globus.org/pub/globus/OpenLDAP-1.2.7-globus-latest.tar.gz>)
3. ユーザ globus の作成
4. /usr/local/globus の作成 (オーナー globus)
5. Globus Toolkit の入手 (<http://www.globus.org/toolkit/download/>)
6. Globus Toolkit の展開
7. globus_install の実行
8. 環境変数 GLOBUS_INSTALL_PATH の設定
9. globus-setup の実行
10. /opt/globus 作成 (オーナー globus)
11. globus-local-deploy の実行 (指示に従い mail を送る)
12. 設定ファイルの編集 (inetd.conf, services, profiles など)
13. ユーザの登録
14. grid_cert_request の実行

□ Globus の使用例

Globus Toolkit の動作の仕組みを、遠隔資源上でジョブを実行するためのコマンド globusrun コマンドを例にとり、Globus Toolkit のツール群がどのように機能して実際に動作するのかを説明する (文

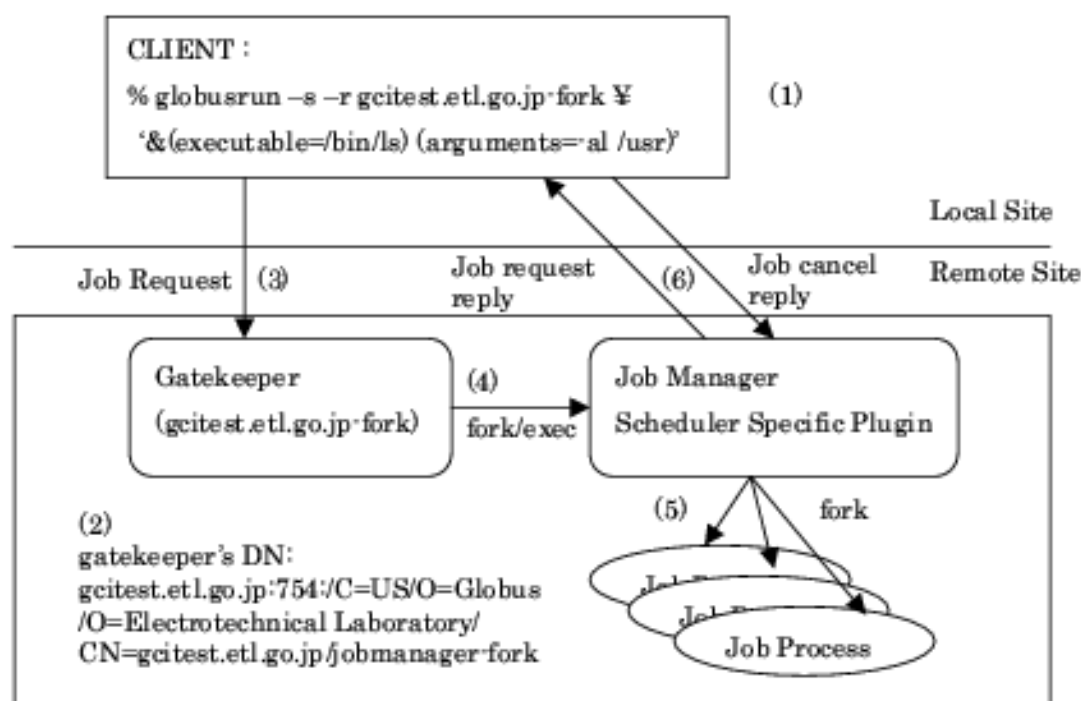


図 9.2: globusrun 実行の様子

献³⁾より)。

(1) globusrun コマンドは図 9.2 に示されている形式で実行する。gcitest.etl.go.jp-fork はジョブを投げける相手 (GRAM) の名前であり、最後の引数は実行ファイルおよび引数を RSL (Resource Specification Language) と呼ばれる方法に従って記述したものである。

(2) globusrun コマンドは MDS のエントリを検索し、gcitest.etl.go.jp-fork という GRAM に対するサービス要求を受け付ける gatekeeper の DN (Directory Name) を獲得する。獲得される DN は図に示されているようなものであり、gatekeeper のホスト名やポート番号などが含まれている。

(3) 獲得された DN を用いて gatekeeper にジョブの要求を送信すると、gatekeeper はパスワードの入力を求めて認証をおこなう。

(4) job manager は GRAM ごとに設定されたファイルを読み込んで、各 GRAM に応じた方針でジョブを生成する。例えば fork 型の GRAM の場合は fork 関数を使ってジョブプロセスを生成し、LSF 型の GRAM の場合は LSF の bsub コマンドを呼び出す。

(5) クライアントと job manager の間ではジョブ送信の成功／失敗の通知や、ジョブの取消要求などのやりとりがおこなわれる。job manager は生成したプロセスの実行が終了するまで、クライアントとジョブプロセスとの仲介をおこなう。ジョブプロセスの出力等は GASS を介してクライアント側の出力に送られる。

このような流れで、Globus Toolkit が提供するさまざまなツールを利用して、遠隔資源上でのジョブの実行が可能となる。

次に、Globus Toolkit の API を用いたサンプルプログラム (filecopy.c) と実行例を示す。このプロ

グラムでは GASS の API を用いてリモートのファイルをコピーする。このプログラムでは、GASS ではリモートファイルへのアクセスを、`globus_gass_xxx` 関数により、通常のローカルファイルへのアクセスと同様に実現している。

[サンプルプログラム (filecopy.c)]

```
/*
 * filecopy.c : copy utility program: to be extended to use the GASS file module
 */

#include <stdio.h>
#include <stdlib.h>
#include "globus_gass_file.h"

int main(int argc, char *argv[]) {
    FILE    *in,*out;
    char    buffer[1024];
    int      n;

    if (argc < 3) {
        fprintf(stderr,"provide arguments <source file> <destination file>\n");
        exit(1);
    }

    globus_module_activate(GLOBUS_GASS_FILE_MODULE);

    in = globus_gass_fopen(argv[1],"r");
    if (!in) {
        fprintf(stderr, "Can not read from %s\n", argv[1]);
        exit(1);
    }

    out = globus_gass_fopen(argv[2],"w");
    if (!out) {
        fprintf(stderr, "Can not write to %s\n", argv[2]);
        exit(1);
    }

    while (!feof(in)) {
        n = fread(buffer,sizeof(char),1024,in);
        if (n>0) fwrite(buffer,sizeof(char),n,out);
    }

    globus_gass_fclose(in);
    globus_gass_fclose(out);

    fprintf(stderr, "Successfully copied %s to %s\n", argv[1], argv[2]);
    globus_module_deactivate(GLOBUS_GASS_FILE_MODULE);
    return 0;
}
```

[Makefile]

```
#
# Makefile for the gass tutorial program
#
GLOBUS_ARCH = i686-pc-linux-gnu
THREADFLAVOR = nothreads

FLAVOR =
$(GLOBUS_INSTALL_PATH)/development/$(GLOBUS_ARCH)_$(THREADFLAVOR)_standard_debug

include $(FLAVOR)/etc/makefile_header

MYCFLAGS = -I$(includedir) $(CFLAGS)
MYLDFLAGS = -L$(libdir) $(LDFLAGS)
MYLIBS = $(LIBS)

WITHGASS_CFLAGS = -I$(includedir) $(GLOBUS_GASS_FILE_CFLAGS) $(CFLAGS)
WITHGASS_LDFLAGS = -L$(libdir) $(GLOBUS_GASS_FILE_LDFLAGS) $(LDFLAGS)
WITHGASS_MYLIBS = $(GLOBUS_GASS_FILE_LIBS) $(LIBS)

withgass: filecopy.c
$(CC) $(WITHGASS_CFLAGS) -o filecopy withgass.c \
    $(WITHGASS_LDFLAGS) $(WITHGASS_MYLIBS)

clean:
$(RM) -f filecopy *.o
```

[実行例]

```
$ make withgass
cc
-I/usr/local/globus/development/i686-pc-linux-gnu_nothreads_standard_debug/include
-I/usr/local/ssl/include -I/usr/local/ssl/include -g
-I/usr/local/globus/development/i686-pc-linux-gnu_nothreads_standard_debug/include
-o filecopy withgass.c \
-L/usr/local/globus/development/i686-pc-linux-gnu_nothreads_standard_debug/lib
-L/usr/local/ssl/lib -L/usr/local/ssl/lib
-L/usr/local/globus/development/i686-pc-linux-gnu_nothreads_standard_debug/lib
-lglobus_gass_file -lglobus_gass_transfer_assist -lglobus_gass_transfer
-lglobus_io -lglobus_gss_assist -lglobus_gss -lssl -lcrypto -lglobus_gaa
-lglobus_common -lglobus_gass_cache -lglobus_nexus -lglobus_mp
-lglobus_common -lglobus_dc -lglobus_io -lglobus_gss_assist -lglobus_gss
-lssl -lcrypto -lglobus_gaa -lglobus_common
$ globus-gass-server &
[1] 16204
https://virgo.ise.osaka-sandai.ac.jp:1855
$ ./filecopy https://virgo.ise.osaka-sandai.ac.jp:1855\
/home/koita/test.txt test.txt

Successfully copied https://virgo.ise.osaka-sandai.ac.jp:1855
/home/koita/test.txt to test.txt
```

ここでは、Globus Toolkit のごく一部の例について紹介した。これだけでは、実際にグローバルコンピューティングシステムは構築できないが、現在でも多くの研究プロジェクトが Globus Toolkit を用いて、グローバルコンピューティングの上位レベルのソフトウェアを開発している。先に紹介した、Condor も Globus Toolkit を用いた実装 Condor-G を開発している。また、他の研究プロジェクトでも、Nimrod/G、Ninf などを Globus Toolkit 上での実装をおこなっている。現在、Globus Toolkit は、グローバルコンピューティングのための標準になりつつある。より深く利用するためには、サンプルプログラムや web 上の API マニュアル、そして ML などを参照すると、自分でもグローバルコンピューティングのシステムの構築に近付くことが可能である。これから、PC クラスタを超えてグローバルコンピューティングにはまってみようと思われる方は、Globus Toolkit を思いっきり使ってみることをお勧めする。

■ 代表的なグローバルコンピューティングシステム

本講習では、もっとも簡単に使えて一番楽しそうな Condor と、もっとも標準的になりつつある Globus Toolkit を紹介したが、他にも多くのグローバルコンピューティングシステムがあるので、興味のある方はぜひ他のシステムの利用にも挑戦して頂きたい。以下、◎はソースからすべて公開、○はバイナリのみ公開、×はソフトウェアの公開なしを表している。

◎ Akenti (<http://www-itg.lbl.gov/Akenti/>), アメリカ

分散ネットワーク環境でのスケーラブルなセキュリティサービスを提供するセキュリティモデルおよ

びアーキテクチャ.

◎ Albatross (<http://www.cs.vu.nl/albatross/>), オランダ

グローバルコンピューティング環境におけるクラスタシステム (広域クラスタシステム) 上でのプログラミング方法や、高性能を得るための方法などに関する研究プロジェクト.

× AppLeS (<http://apples.ucsd.edu>), アメリカ

アプリケーションレベルでのスケジューリングに着目した研究プロジェクト.

○ Condor (<http://www.cs.wisc.edu/condor/>), アメリカ

分散配置された多数のワークステーション等を使って高スループットな計算を行うための開発環境等の枠組みを提供するシステム.

○ EuroTools (<http://www.irisa.fr/EuroTools/>), ヨーロッパ

ヨーロッパ全体で高性能計算・ネットワークを利用するためのプロジェクト.

◎ Globus (<http://www.globus.org>), アメリカ

アメリカを中心とした、現在最も大きなプロジェクト. グローバルコンピューティング環境のソフトウェアインフラストラクチャを構築するためのツール群を提供している.

× Grid Forum(<http://www.gridforum.org>), 世界中

グローバルコンピューティングシステムの標準化を目指し、さまざまな技術要素に関して意見交換を行う会合.

◎ IceT (<http://www.mathcs.emory.edu/icet>), アメリカ

グローバルコンピューティング環境における仮想的な計算機の併合/分割や、コードやデータの可搬性などの機能を備えたメタコンピューティングシステム.

× IPG (<http://www.nas.nasa.gov/Groups/Tools/IPG/>), アメリカ

コンセントにプラグを差し込むだけで誰もが電力の供給を受けることができるのと同様に、国中/世界中の情報を簡単に引き出すことができるような枠組み、仕組みを提供するシステムに関する研究プロジェクト.

○ Legion (<http://legion.virginia.edu/>), アメリカ

デスク上のパソコン/ワークステーションから世界中の資源にアクセスすることのできる、オブジェクトベースのメタシステムソフトウェアプロジェクト.

◎ NetSolve (<http://www.cs.utk.edu/netsolve/>), アメリカ

ネットワークを介して遠隔地にある科学技術計算用ライブラリを利用するためのクライアント・サーバモデルに基づくシステム.

◎ Ninf (<http://ninf.etl.go.jp/>), 日本

遠隔手続き呼出しの手法を利用し、プログラマが容易に遠隔地にあるハードウェア、ソフトウェア、データベース等を利用することのできるシステム.

◎ PACX-MPI (<http://www.hlr.de/structure/organisation/par/projects/pacx-mpi/>), ド

イツ

グローバルコンピューティング環境用に拡張した MPI 通信ライブラリ.

○ UNICORE (<http://www.kfa-juelich.de/unicore/>), ドイツ
ユーザが遠隔地にある計算資源にジョブを送信するためのシステム.

■ おわりに

グローバルコンピューティングのためのソフトウェアシステム Condor とソフトウェアツール群 Globus Toolkit を紹介した. まず, 簡単に遊びたいという方は Condor をインストールすることをお勧めする. 設定ファイルひとつで, かなりいろいろなことができ, 周りの PC をタダで思う存分利用できる. しかし, ソース非公開のつらさで, 使っていくうちに資源管理だけではいろいろ物足りなくなってくると思われる. そこで, Globus Toolkit を使ってさらに面白いシステム構築を目指してはいかがだろうか.

■ 参考文献

- 1) 関口智嗣: グローバルコンピューティングとは, Computer Today, No.95, pp.46–52, 2000.
- 2) 関口智嗣: グローバルコンピューティングテクノロジー, Computer Today, No.96, pp.66–72, 2000.
- 3) 田中良男: グローバルコンピューティングシステムの紹介, Computer Today, No.97, pp.60–66, 2000.
- 4) 建部修見: 米国から見た GCI/Network servers, Computer Today, No.98, pp.64–70, 2000.
- 5) Condor Project Homepage. <http://www.cs.wisc.edu/condor/>
- 6) The Globus Project. <http://www.globus.org/>
- 7) Bertil Follit and Pierre Sens.: “Load Sharing and Fault Manager”, High Performance Cluster Computing Vol.1, Chapter 22, Prentice Hall PTR(ISBN 0-13-013784-7), 1999.
- 8) 小坂隆浩: ディスクレスクラスタの構築, PC クラスタ超入門 講習会資料, pp.63–67, 超並列計算研究会, 1999.

SCore 型クラスタ

新情報処理開発機構

石川 裕

ishikawa@rwcp.or.jp

■ はじめに

技術研究組合 **新情報処理開発機構** は、国内外のコンピュータ関連企業、研究所から構成される研究組合で、通商産業省が 1992 年から 10 年計画で実施しているリアルワールドコンピューティング (RWC) プロジェクトを受託している。本稿では、当研究組合が開発してきた RWC PC クラスタハードウェアならびに SCore クラスタシステムソフトウェア Version 3 の特長を述べる。さらに、RWCPC クラスタ上で稼動している実用アプリケーションを紹介する。

■ 背景

ネットワークに繋がったパソコンを束ねて並列コンピュータ環境を実現する、いわゆるクラスタ、が普及し出している。特に既存フリーソフトウェアの組み合わせで製作された Beowulf 型クラスタが注目を集めているが、次の 2 点に問題があり、その利用範囲は限られたものである。低い通信性能 100 メガビット秒程度のネットワークを使い、インターネットで使用されている TCP/IP ネットワークプロトコルを使用しているために専用並列コンピュータに比べて通信性能で 1/10 以下の性能しかでない。非効率なコンピュータ管理各パソコン上で独立したオペレーティングシステムが動いていて、全てのパソコンを取りまとめる機能がないために、並列プログラムを効率良く実行することがない。

RWC プロジェクトでは、1994 年よりクラスタ構築を検討し、1995 年以降、Sun ワークステーション、Intel Pentium、Compaq Alpha を使用したクラスタを制作している。これらクラスタ上で、クラスタに必要なシステムソフトウェアを開発してきた。本システムソフトウェアを SCore クラスタシステムソフトウェアと呼び、SCore クラスタシステムソフトウェアを搭載しているクラスタを SCore 型クラスタと呼んでいる。SCore により、Beowulf 型クラスタの欠点を取り除き、専用並列コンピュータと同等の性能を有するクラスタが実現できた。

■ RWC クラスタ開発の歴史

1994 年、ワークステーションと高速ネットワークを使用したクラスタ開発を検討していた。その当時話題になっていた ATM/LAN を使用したクラスタ構築事例がいくつかあったが、我々は、ATM/LAN

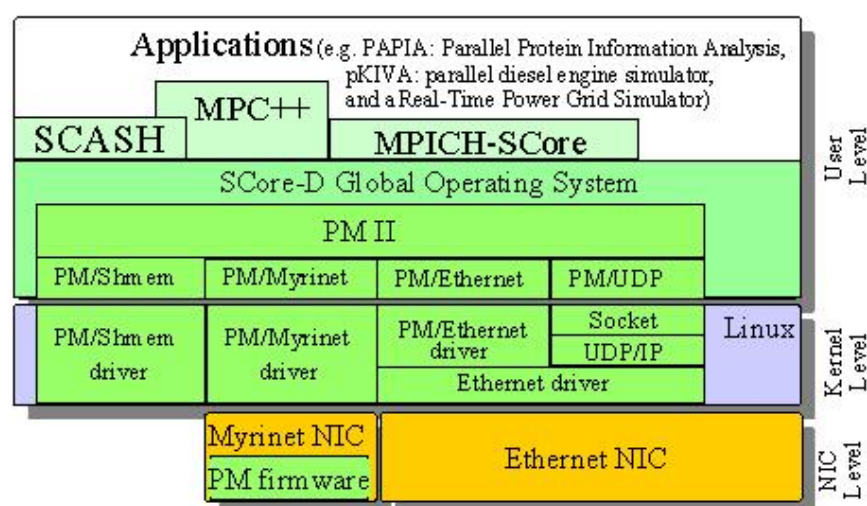


図 10.1: SCore クラスタシステムソフトウェア

以外のネットワークを探していた。なぜならば、ATM/LAN はパケット長が小さいだけでなく、物理層におけるパケットロストを許す仕様であったために、並列アプリケーションが要求する通信性能が提供出来ないと判断したからである。

1995 年春、**Myrinet ネットワーク**を知った。Myrinet は、物理層でギガビット／秒の転送スピードを有し、ハードウェアレベルでパケット到着を保証する仕様である。また、Myrinet NIC に搭載されているプロセッサのプログラムを開発することにより、独自のネットワークプロトコルを実現することができるのも魅力的であった。

1995 年夏に Myrinet と Sun Sparc Workstation 20 を使用した小規模クラスタを構築し、idxPM 通信ライブラリ、ユーザレベルスレッドライブラリおよび **MPC++**、**SCore-D**を開発し、SCore クラスタシステムソフトウェアの原型を作った。1996 年初頭には、36 台の Sun Sparc Workstation 20 を使用したクラスタを構築した。この時、ベンチマークプログラムで Cray 社の Cray T3D に匹敵する性能が出ていることを確認している。

1996 年に 32 台の Intel 社製 Pentium プロセッサを使った RWC PC クラスタ 1 号機を開発し、1997、1998 年にかけて RWC PC クラスタ 2 号機を開発した。これら開発を通して、市販品のパソコンを用いても省スペースを実現出来ることを実証した。これら PC クラスタでは PICMG という工業規格ボードを用いている。

RWC PC クラスタ 2 号機では、Intel 社製 Pentium Pro 200MHz、256MBytes メモリ、4GBytes ディスクから構成されるコンピュータを使用し、全体で 128 台構成となっている。通信ハードウェアとして 100Mbps イーサネット以外に 1 ギガビット秒の通信性能をもつ Myricom 社 Myrinet を採用している。

1999 年には、Intel 社製 Penitum III を 2 台搭載した PC16 台および Compaq 社製 Alpha 21264 プロセッサを搭載した XP-1000 コンピュータ 16 台を、Myrinet、ギガビットイーサネットで接続したクラスタを制作した。

現在、Dual Pentium III 8000 MHz を使用した 128 プロセッサ数規模のクラスタである RWC PCCluster IV を構築中である。ネットワーク性能の違いによる並列アプリケーションの性能の違いを検証する目的で、Myrinet、Gigabit Ethernet および 3 系統の 100Mbps Ethernet が接続されている。

■ SCore Cluser System Software

SCore は、ワークステーションや PC 等で稼働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システムソフトウェアである。最新版は、 **TurboLinux** , **Redhat** , **SuSE** 上で稼動する SCore Version 3.1 である。現在、Version 3.2 の配布準備を進めており、9 月下旬には Version 3.2 を正式リリースする予定でいる。

図 10.1 に示すとおり、SCore クラスタシステムソフトウェア Version 3 は、

1. 高性能通信機能を実現した PM 2 通信ライブラリ³⁾ ,
2. PM 2 上に構築した MPI 通信ライブラリである MPICH-SCore⁴⁾ ,
3. コンピュータ資源を管理する SCore-D グローバルオペレーティングシステム²⁾ ,
4. 分散共有メモリシステム SCASH¹⁾ ,
5. 並列プログラミング言語 MPC++¹⁾ ,
6. Omni OpenMP コンパイラ⁶⁾ ,

から構成され、さらに、以下のツールを提供している。

- EIT イージーインストールツール
- SCOOP クラスタ監視ツール

以下、SCore の特長を紹介する。

□ 高性能通信機構

通信ハードウェアの性能を引き出すために **PM 2 通信ライブラリ** を開発した。PM2 は複数のネットワークハードウェアを支援できるよう、PM2 API レイヤ、PM2 デバイスレイヤの 2 つのレイヤから構成されている。現在までに、Myrinet および Ethernet 用ドライバ、共有メモリ型コンピュータ用に OS の共有メモリ機構を利用したドライバ、UDP プロトコルを利用したドライバ、が開発されている。これにより、次の 2 つの利用方法が可能となっている⁵⁾。

- 異なるネットワークが混在したクラスタを構築できる。たとえば、Myrinet を使った 2 つのクラスタをギガビットイーサネットをつなげたクラスタオブクラスタを構築することができる。
- PM2 上に構築された MPI 通信ライブラリ等のアプリケーションは、再コンパイルすることなく、シームレスにこれらネットワークハードウェアを利用できる。たとえば、100Mbps イーサネットを利用した小規模クラスタ上で稼動しているアプリケーションを、そのまま Myrinet を利用した大規模クラスタ上でも稼動する。

Myrinet 用 PM 2 では、Myrinet ネットワークインターフェイスカードのファームウェアを変更して、ユーザレベル通信およびゼロコピー通信機能を実現することにより、Pentium III(500MHz) を使った PC でラウンドトリップ 16 マイクロ秒(4 バイトメッセージ) , 116Mbytes/sec(8 キロバイトメッセージ) のバンド幅を達成している。図 10.2 に MPI 通信ライブラリにおける 1 対 1 通信バンド幅の比較を示す。

SCore Version 3.2 からは、複数のネットワークを一つに束ねて通信する、 **ネットワークランキング** 機能が実現されている。図 10.3 に、100Mbps Ethernet カード 5 枚挿しまでの性能を示す。440BX および 450MHz Pentium III と Tulip NIC の組合せでは、NIC を 3 台搭載するとほぼ 3 倍の性能が出ていることが分かる。

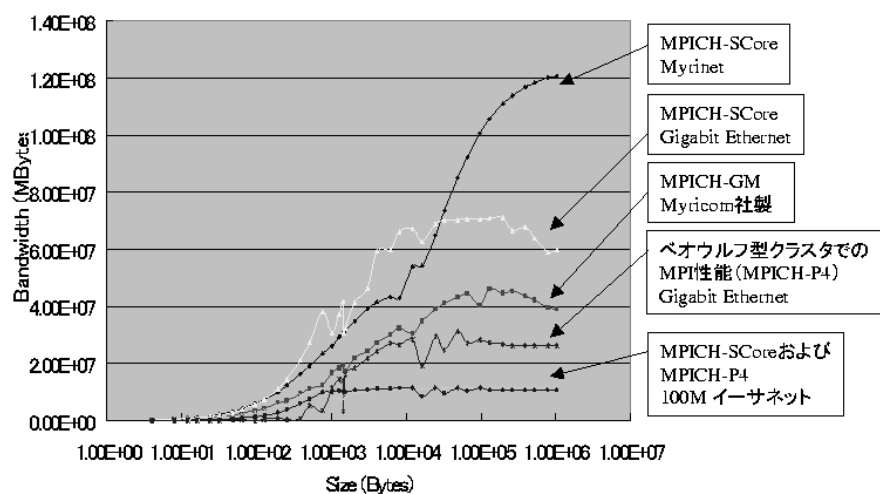
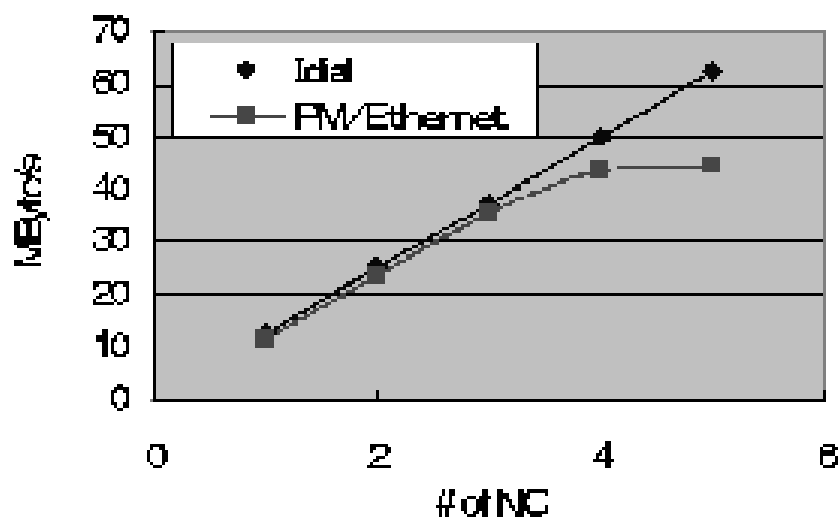


図 10.2: MPI 通信ライブラリにおける 1 対 1 通信バンド幅の比較



Pentium III 450 MHz 440BX , 5 つの Tulip NIC による一対一通信性能

図 10.3: PM/Ethernet ネットワークランキング性能

□ 単一システムイメージ

商用並列コンピュータが提供しているシングルユーザモード，時分割・空間分割スケジューリングモード，バッチモードを提供している．アドバンスドランタイムその他，Score では，デッドロック検出機構，プログラムが異常終了した時に自動的にデバッガを立ち上げる機能，チェックポイント・リスタート機能を提供している．

SCore Version 3.2 から PBS バッチジョブシステムもサポートしている．

□ 豊富なプログラミング支援環境

SCore では，並列アプリケーション記述で広く利用されている MPI 通信ライブラリだけでなく，マルチスレッド，リモートオブジェクト呼出し機能，グローバル演算を C++ で実現した MPC++，分散共有メモリによるプログラミングを支援する SCASH が提供されてきている．

さらに，SCore 3.2 では，クラスタ用 OpenMP として Omni/SCASH が提供されている．OpenMP は，SMP 用システムとして規格化され，多くのベンダがシステムを提供している．計算ノードが SMP 構成の SMP クラスタでは，ノード内の並列性記述には OpenMP を，ノード間の並列記述には，MPI 通信ライブラリを使用すると言うのが，一般的であった．

Omni/SCASH OpenMP システムにより，OpenMP で記述されたプログラムは，SMP の上でもクラスタの上でも稼働することが出来るようになった．

□ 利用可能なコンパイラ

現在使用可能なコンパイラは次の通りである．

- Gnu コンパイラ
- Fujitsu コンパイラ
- Absoft コンパイラ
- KAI コンパイラ
- PGI コンパイラ

■ アプリケーションの紹介

□ pKIVA

pKIVA は，米ロスアラモス国立研究所で開発されたディーゼルエンジンシミュレーションプログラムである KIVA を，RWC プロジェクトで並列化したソフトウェアである．自動車工業界で実際に使われている KIVA の並列版である pKIVA により，今まで PC 一台で十数時間かかっていた処理が数十分で処理できるようになる．

□ AMBER

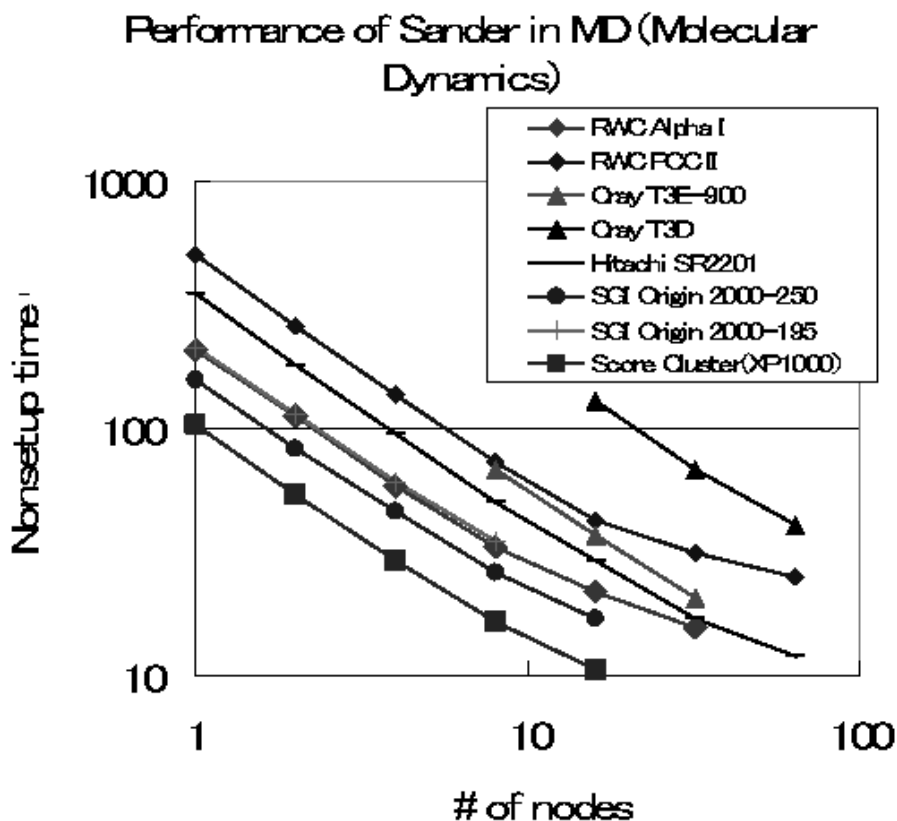


図 10.4: Amber の性能

分子動力学シミュレータで世界的に有名な AMBER というソフトウェアもクラスタ上で稼働している。図 10.4 に示す通り、我々のクラスタでは、今までのスーパーコンピュータである Cray T3E や日立 SR2201 以上の性能が出ている。

□ PAPIA

PC クラスタの持つローカルディスクはデータベース検索エンジンにも最適である。その応用例として PAPIA システムを紹介する。PAPIA システムは、遺伝子の配列やタンパク質の立体構造からその意味を探りだすために、データベース検索を必要とするシステムである。PAPIA システムは 1998 年初頭より、世界中の研究者が利用できるように公開している (<http://www.rwcp.or.jp/papia/>)。

■ ユーザ

1998 年秋より 300 サイト以上がインターネット経由でソフトウェアを入手している。国内では、三菱産業システム研究所が実時間電力シミュレータのシステムとして SCore が利用されている。米国では、ロスアラモス国立研究所で使用されている。ヨーロッパでもイギリス Oxford 大学計算センタ、ドイツ Bonn 大学応用数学科、フランスパリ南大学等で使われている。たとえば、ボン大学では、Parnass2 と呼ばれるクラスタを製作した。Parnass2 は、Myrinet ネットワークでつながれた Intel 社製 PentiumII 400 MHz をノードコンピュータに持つ 144 台構成のクラスタである。

■ 終わりに

新情報処理開発機構では、クラスタおよび並列分散処理市場の拡大を支援すべく、昨年 10 月より、開発したソフトウェアをオープンソースとして提供している。これにより、現在、SuSE, Turbo-Labs/TurboLinux, Kondra は、SCore を使用したクラスタパッケージを開発中である。ソフトウェア配布に関しては <http://www.rwcp.or.jp/lab/pdslab/> を参照してほしい。

本年 10 月 3 日には、第一回 SCore Users Meeting がイギリス Oxford 大学で開催される。また、クラスタの普及を支援すべく、大学、研究機関、企業の研究所に新情報処理開発機構のクラスタを利用してもらおう機会を設けている。ご興味のある方は、score-info@rwcp.or.jp 宛てにメールしていただきたい。

Linux Distributor による SCore のサポート、ユーザーズミーティングによる know how の蓄積および共有を通して、クラスタ市場ならびに SCore 型クラスタが発展していくことを期待している。

■ 参考文献

- 1) Y. Ishikawa, H. Tezuka, A. Hori, S. Sumimoto, T. Takahashi, F. O'Carroll, and H. Harada, "RWC PC Cluster II and SCore Cluster System Software - High Performance Linux Cluster," in Proc. of the 5th Annual Linux Expo, pp. 55-62, 1999.
- 2) A. Hori, H. Tezuka, and Y. Ishikawa, "Highly Efficient Gang Scheduling Implementation," in SC'98, 1998.
- 3) H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa, "Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication," in IPPS/SPDP'98, pp. 308-314, IEEE, 1998.
- 4) T. Takahashi, F. O'Carroll, H. Tezuka, A. Hori, S. Sumimoto, H. Harada, Y. Ishikawa, and P. H. Beckman, "Implementation and Evaluation of MPI on an SMP Cluster," in Parallel and Distributed Processing - IPPS/SPDP'99 Workshops, Vol. 1586 of Lecture Notes in Computer Science, pp. 1178-1192, Springer-Verlag, 1999. 1
- 5) Toshiyuki Takahashi, Shinji Sumimoto, Atsushi Hori, Hiroshi Harada, Yutaka Ishikawa, "PM2: A High Performance Communication Middleware for Heterogeneous Network Environments," To appear in SC2000, 2000.
- 6) OpenMP, <http://www.rwcp.or.jp/lab/pdperf/> .