LAM/MPI マニュアル

谷 合 由 章[1]

平成 29 年 5 月 24 日

[1]mail:yosi@bcl.sci.yamaguchi-u.ac.jp

目 次

第I部

LAMの説明書

第1章　LAM/MPIについて

MPI(Message Passing Interface) は, メッセージ通信によって並列化を行う標準規格です. その規格に準拠して作成されたライブラリがLAM/MPI です.MPI は規格であるので, 実際に並列化を行うときLAM の機能が中心になります.MPI という大前提のものがどういう感じであるかを説明し, その後メインのLAM/MPI に関する実戦的なところを説明します.

1.1　MPIについて

1.1.1　MPIの現在までの流れ

1992 年 4 月 29-30 日に「分散メモリ環境におけるメッセージ通信のための標準に関するワークショップ」においてMPI の標準化が始まった. このワークショップはバージニア州ウィリアムズバーグで「並列コンピューティング研究センタ」の主催によって開催された.1992 年 11 月に,Dongarra,Hempel,Hey,Walker により,MPI1 として知られる予備的な草案が提示された. それは並列化の論議を促進するためのたたき台として役割りを果たした. 論議には 40 を越える団体が参加して, メッセージ通信のためのライブラリインターフェースの標準に関する議論と定義が度々行われてきた. 現在ではMPI2 という標準が設計されるに至っている.

1.1.2　どういう目標があったのか

Message Passing Interface の目標は, メッセージ通信を行うプログラムを書くための, 広く一般に使われる標準を作り出すことのもとに設計された. この目標を実現するには,MPI は実用的で, ポータブルで, 効率がよくかつ融通の効くメッセージ通信の標準を確立しなければならない. 以下に全ての目標のリストを以下に示す.

- アプリケーションプログラムから呼ぶためのインタフェースを設計する.
- 効率のよい通信を可能にすること. メモリ間コピーを避け, 計算と通信を並行して進めることができ, 通信プロセッサがあれば処理の一部をそれに任せるようなもの.
- 異機種環境でも使用できる処理系に備える.
- C 言語やFortran 言語のための使い易い呼び出し形式を可能にする.
- 信頼できる通信インタフェースを想定する. ユーザーは通信障害に対処する必要がない. そのような障害は下位の通信サブシステムが処理する.
- PVM,NX,Express,p4 などの既存のインタフェースとそれほど違わないインタフェースで, なおかつより融通の効く拡張機能を可能にするインタフェースを定義する.
- 多くのベンダのプラットフォーム上に実現でき, その際, 下位の通信およびソフトウェアに大きな変更を加えずに済むインタフェースを定義する.
- このインタフェースの意味は言語に依存するべきではない.
- このインタフェースは, マルチスレッド環境への対応が可能なように設計するべきである.

1.1.3 概要

MPI(Message Passing Interface) は得に分散メモリを備える並列マシンで広く用いられている. その種類にはいろいろあるが, プロセスがメッセージによって通信を行うということが基本概念になる.

設計においては, 既存のメッセージ通信システムが持つもっとも魅力的な機能の数々を採り入れるように努められた.

メッセージ通信の標準を確立する主な利点は, ポータビリティと使いやすさににある. 分散メモリ環境の中でも得に, 高いレイヤ層でのルーチンやアブストラクションが低いレイヤのメッセージ通信ルーチンの上に構築されているものでは, 標準化の利点は明白である. さらに, 定義に沿ったルーチンを提供している. このようなルーチン群があれば, ベンダは, それが効率よく実現したり, あるいは場合によってはそれ専用のハードウェアを用意したりすることができ, それによりスケーラビリティを高めることができる.

MPI2 標準に含まれるもの

- 1 対 1 通信 (MPI1)
- 集団操作 (MPI1)
- プロセスグループ (MPI1)
- 通信コンテキスト (MPI1)
- プロセストポロジー (MPI1)
- Fortran77 言語と C 言語の呼び出し形式 (MPI1)
- 環境管理と問い合わせ (MPI1)
- プロファイリングインタフェース (MPI1)
- 入出力 (I/O)(MPI2)
- Fortran90,C++言語向けの枠組み (MPI2)
- 動的プロセス制御 (MPI2)
- 片側通信 (MPI2)
- グラフィックス (MPI2)
- 実時間対応 (MPI2)

1.2 LAM/MPI

実際にプログラムするときに MPI の機能を提供するのは LAM/MPI である. 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 Nortre Dame) が作成したフリーの MPI ライブラリである. 現在はインディアナ大学 (Indiana university) によって引き継がれ, 大幅にバージョンアップが行われている. 機能的にも高いものになってきており, 今後も発展が見込まれる.

1.2.1 LAM/MPI のサポート

LAM ユーザガイドの 29 ページ
LAM7.0.3 では全ての MPI-1 関数をサポートしている.MPI-2 は大体サポートされているが一部未サポートがある.

1.2.2 LAM/MPI について (実践)

LAM/MPI のプログラムが書けるとしても, それだけではプログラムは動かない.LAM/MPI を構築することに関する管理者向けのものは付録 2 に, コンパイルや実行についてのユーザ向けのものは付録 3 を参照してください.

第2章　並列計算環境の構築(管理者向け)

以下の設定環境は VINE LINUX 2.6 CR で行いました. LAM パッケージを利用する場合, パスワードなしでログインできる設定を行っておく必要があります.ssh か rsh のいずれか 1 つを使う環境に合わせて選択し, その設定を行ってください.

1000 Base の Giga Bit Ethernet の設定と Intel 製のコンパイラの使用はオプションとしてあります. これらを採りいれなくても, MPI パッケージは使用できるのですが,MPI による並列計算のパフォーマンスをより上げる意味で行いました.

以上のことが終われば, LAM を利用できることになります.

2.1　パスワードなしでのログイン

2.1.1　rsh, rlogin 編

【はじめに】

rsh, rlogin は他のホストにネットワークを経由してログインを可能にします. ですが, ネットワークの経路上は暗号化されないため, セキュリティ上非常に危険にさらされることになります.

しかし, 経路がインターネットなどの不特定多数の経路 (人) がいないローカルネットワークではそうではありません. 暗号化することに比べて, 非常にめんどうな鍵の管理を考えなくてすみます. さらに, 暗号化と復号化にかかるコストもなくなります (しかし, 通信が遅い場合や通信内容が重い場合, ssh には圧縮機能があるため, 有効利用できます).

並列計算環境を作る上で rsh, rlogin の有用性を持っていますので, ここに文章としてまとめています.

【パスワードなしでログインできるまで】

ここでは, 見通しとして概要を見ます.

1. rsh, login のサービスを有効にする.

2. pam 認証において, パスワードを必要なしにする.

3. 設定を反映させるために, inet を再起動する.

4. /etc/hosts.equiv やユーザのホームディレクトリ~/.rhosts にログインを許可するホスト, ユーザを書いておく.

【詳細な設定】

[**pam 認証でホスト間認証を設定する**]

pam 認証は, さまざまなアプリケーションの認証を共通に扱うものです. プログラマは, パスワードの要求, 判定などを, pam の共有ライブラリにまかせることができます. そして, 柔軟な認証設定ができ, 強力な認証機能をもっています.

具体的に rsh, rlogin での認証をパスワードなしにする設定を以下行っていきます.

1. rsh での設定

/etc/pam.d/rsh に rsh の設定ファイルがおいてあります. 以下にその中身を載せました.

```
#%PAM-1.0
auth       required /lib/security/pam_nologin.so
#auth       required /lib/security/pam_securetty.so
auth       required     /lib/security/pam_env.so
auth        required    /lib/security/pam_rhosts_auth.so \
  hosts_equiv_rootok promiscuous
account    required /lib/security/pam_stack.so service=system-auth
session    required /lib/security/pam_stack.so service=system-auth
```

pam_rhosts_auth.so の hosts_equiv_rootok promiscuous は root でのログインを可能にするオプションです. これを設定するのはセキュリティ上, 非常に危険です.

2. rlogin での設定

/etc/pam.d/rlogin に rlogin の設定ファイルがおいてあります. 以下にその中身を載せました.

```
#%PAM-1.0
auth       required /lib/security/pam_nologin.so
#auth       required /lib/security/pam_securetty.so
auth       required     /lib/security/pam_env.so
#auth       sufficient /lib/security/pam_rhosts_auth.so
#auth       required /lib/security/pam_stack.so service=system-auth
auth       required     /lib/security/pam_rhosts_auth.so\
 hosts_equiv_rootok promiscuous
account    required /lib/security/pam_stack.so service=system-auth
password   required /lib/security/pam_stack.so service=system-auth
session    required /lib/security/pam_stack.so service=system-auth
```

pam_rhosts_auth.so の hosts_equiv_rootok promiscuous は root でのログインを可能にするオプションです. これを設定するのはセキュリティ上, 非常に危険です.

[パスワードなしでログインできるホストを決める]

rsh, rlogin のサービスを提供して, パスワードなしのログインができるようになったのですが, それでめでたしめでたしとはいきません. 誰でもパスワードなしで, どこからでもログインできることは危険きわまりないです. そこで, ここでは特定のホストだけにパスワードなしでのログインを許可するように設定を行います.

設定には２通りあり, 1 つはユーザ個人で許可を許せるユーザレベルでの設定, もう 1 つは管理者によって許可するホストレベルでの設定です. 以下, この２つについて, それぞれの設定を行います.

1. ユーザレベルでの許可をする

設定ファイルはホームディレクトリの~/.rhosts におきます. このファイルに書かれているホストに対して許可を出します. 書いていないホストはログインすることはできません.

書式は, 箇条書で, ホスト名, FQDN, IP address のいずれでも OK です. ただし, それによって名前解決ができることが必須条件です.

なお, ホストの横に 1 つスペースをおいて, ユーザ名を書いておくとそのホストのそのユーザのみの許可だけを許すことができます.

以下, その例を載せます.

```
[例]
node1.sample.ac.jp user1
node2.sample.ac.jp
```

上の例では, node1 からはユーザ user1 だけがログイン可能なことがわかります. 1 つ下の node2 からは, どのユーザでも (root は pam 認証の設定次第で) ログインできることになります.

ファイルがない場合は作成してください. 作成した場合, ファイルアクセスのパーミッションを「chmod 600 .rhosts」を実行し, 変えておく必要があります. 万が一, ファイルのパーミッションの設定で, 他人がそのファイルに任意のコードを書き込めた場合, ログインなどの際に.rhosts が読み込まれ意図しない動作を行ったり, システムを乗っ取られたりする可能性があります.

2. ホストレベルでの許可

/etc/hosts.equiv が設定ファイルです. ホストレベルでの認証の場合にこれを記述する. 書式は「 /.rhosts」と同様である.

2.1.2 ssh 編

【設定の概要】

毎回ログインする際にパスワードの入力がありますが, それを行わないでログインすることもできます. SSH1 プロトコルでは RhostsRSA 認証, SSH2 ではホストベース認証がそれに当たります. その他 ssh-agent を利用する方法もありますが, ここでは SSH2 のホストベース認証を扱います. 以下の作業は [11] や LAM の公式ページの FAQ を参考にしています.

【ホストベース認証】

ホストベース認証は, パスワードの入力なしにログインすることができることを前述しました. これには 2 つのレベル, クライアントレベルとホストレベルでの管理ができます.

クライアントレベルでの認証は管理者でなくても, 自由にパスワードを入力なしでログインすることを設定することができます. ホストレベルでは, 管理者のみがパスワードの入力なしにするホストを決めることができます.

【サポート】

Hostbased 認証がサポートされているのは OpenSSH-2.9 以降のバージョンみたいです. 詳しくは公式ホームページなどを参照してください.

【バグフィックス】

「openssh-3.5p1」に不都合があるようです. これによってホストベース認証ができないことがわかっています. そのため, Vine2.6 の初期のバグを含んだ ssh パッケージを「openssh-3.5p2」にする必要があります. なお, 本研究室では, apt でアップグレード可能なようにしています.

[訂正]
実際のところバグはないような気がします. 詳しくはわかりませんので, 御本人でお確かめください.

【手順】

Hostbased 認証を行う上で, おおまかな作業手順です.

1. サーバでホストベース認証ができるようにする.(ファイル/etc/ssh/sshd_conf の設定)
2. クライアントのホスト鍵をサーバのファイル /etc/ssh/ssh_known_hosts2 に登録する.
3. サーバのファイル /etc/ssh/shosts.equiv を正しく設定しておく.

【設定が必要なファイル】

見通しが良くなるように, どのファイルを変更することになるのかを上げました.

1. /etc/shosts.equiv

   パスワードなしでログインを許すホストを決める設定ファイルです. 書くときはホストを特定する名前 (IP, FQDN, ホスト名) を使います. 箇条書をした各々のホストの横 1 つスペースをおいてユーザ名を書くことで, ユーザレベルでのログインを制限できます.

2. /etc/ssh/sshd_config

   SSH サーバの設定ファイルです. ホストベース認証を行うように設定変更します.

3. /etc/ssh/ssh_known_hosts2

   /etc/shosts.equiv で箇条書したホストの公開鍵を登録しておくファイルです. ファイル名の最後に「2」がついていることで, SSH2 の場合に参照されることになります. SSH1 の場合はファイル /etc/ssh/ssh_known_hosts になります.

4. /etc/ssh/ssh_host_dsa_key と /etc/ssh/ssh_host_dsa_key.pub

   そのホスト自身の公開鍵と秘密鍵を置いておくファイルです. 二つのファイルの名前の違いは最後の「.pub」の拡張子で, ついているものが公開 (public) 鍵です. ほとんどの場合, インストールされたときに作成されています. この二つのファイル (鍵) の作成は, コマンドラインで行います.

【aris サーバでの具体的な設定】

aris 並列計算機サーバから, 実際に計算するホストにホストベース認証できるように設定します (クライアントの設定). 逆に, 計算ホストからサーバへの認証には, パスワード認証, DSA 認証を使い, パスワードの入力を必要にします.

[クライアントの設定]

ファイル ~/.ssh/config を作成しておく.

```
Host *
  ForwardAgent no
  ForwardX11 no
  PreferredAuthentications hostbased,pubkey,password # 優先する認証の順番
  PubkeyAuthentication yes # 公開鍵認証 (SSH2)
  RhostsAuthentication no # 普通は no
  RhostsRSAAuthentication no # 普通は no
  RSAAuthentication yes # 公開鍵認証 (SSH1)
  PasswordAuthentication yes # パスワード認証
  HostbasedAuthentication yes # ホストベース認証
  BatchMode no
  CheckHostIP yes
  StrictHostKeyChecking ask
  IdentityFile ~/.ssh/identity
  IdentityFile ~/.ssh/id_rsa
  IdentityFile ~/.ssh/id_dsa
  Port 22
  Protocol 2,1 # 利用するプロトコルバージョン
  Cipher 3des # ssh1 で使う暗号化方式 (blowfish も可)
  Ciphers aes128-cbc,3des-cbc,blowfish-cbc,cast128-cbc,arcfour,aes192-cbc,aes256-cbc # SSH2
  EscapeChar ~
Host *
ForwardX11 yes
```

PrefferedAuthentication の項目は, 認証の際に優先して行う認証方法をスペースをいれずに書きます. 上では, ホストベース認証, 公開鍵認証, パスワード認証の順になります.

[サーバの設定]

1. 公開鍵, 秘密鍵を作成する

   ```
   $ssh-keygen -t dsa -f /etc/ssh/ssh_host_dsa_key -N ""
   ```

   これで公開鍵, 秘密鍵が作成されます. 作成されている場合は, 作成しなくてもよいです. ユーザレベルでの公開鍵, 秘密鍵はパスフレーズで暗号化されておかれるのですが, このホスト鍵は暗号化してはいけません. 暗号化しないために指定したオプションが「$-N$ ””」になります.

2. /etc/ssh/sshd_config

```
Port 22
Protocol 2 # SSH2 のみでの認証
#ListenAddress 0.0.0.0
#ListenAddress ::

# HostKey for protocol version 1
#HostKey /etc/ssh/ssh_host_key
# HostKeys for protocol version 2
#HostKey /etc/ssh/ssh_host_rsa_key
HostKey /etc/ssh/ssh_host_dsa_key # ホスト鍵に dsa を使う

# Lifetime and size of ephemeral version 1 server key
#KeyRegenerationInterval 3600
#ServerKeyBits 768

# Logging
#obsoletes QuietMode and FascistLogging
#SyslogFacility AUTH
SyslogFacility AUTHPRIV
#LogLevel INFO

# Authentication:

#LoginGraceTime 120
#PermitRootLogin yes
PermitRootLogin no
#StrictModes yes

#RSAAuthentication yes
PubkeyAuthentication yes # SSH2 で公開鍵認証を使う
#AuthorizedKeysFile .ssh/authorized_keys

# rhosts authentication should not be used
RhostsAuthentication no # 普通 no
# Don't read the user's ~/.rhosts and ~/.shosts files
IgnoreRhosts no # ユーザレベルでのホストベース認証を禁止
# For this to work you will also need host keys in /etc/ssh/ssh_known_hosts
RhostsRSAAuthentication no # 普通 no
# similar for protocol version 2
HostbasedAuthentication no # ホストベース認証を使わない
# Change to yes if you don't trust ~/.ssh/known_hosts for
# RhostsRSAAuthentication and HostbasedAuthentication
IgnoreUserKnownHosts yes

# To disable tunneled clear text passwords, change to no here!
PasswordAuthentication yes # パスワード認証を使う
PermitEmptyPasswords no

# Change to no to disable s/key passwords
#ChallengeResponseAuthentication yes
```

```
# 前ページの続き
#
# Kerberos options
#KerberosAuthentication no
#KerberosOrLocalPasswd yes
#KerberosTicketCleanup yes

#AFSTokenPassing no

# Kerberos TGT Passing only works with the AFS kaserver
#KerberosTgtPassing no

# Set this to 'yes' to enable PAM keyboard-interactive authentication
# Warning: enabling this may bypass the setting of 'PasswordAuthentication'
#PAMAuthenticationViaKbdInt no

#X11Forwarding no
X11Forwarding yes
#X11DisplayOffset 10
#X11UseLocalhost yes
#PrintMotd yes
#PrintLastLog yes
#KeepAlive yes
#UseLogin no
UsePrivilegeSeparation no # ホストベース認証を使う場合，yes
#PermitUserEnvironment no
#Compression yes

#MaxStartups 10
# no default banner path
#Banner /some/path
#VerifyReverseMapping no

# override default of no subsystems
Subsystem sftp /usr/libexec/openssh/sftp-server
```

【並列計算におけるクライアントの設定】

[**公開鍵, 秘密鍵を作成する**]

```
$ ssh-keygen -t dsa -f /etc/ssh/ssh_host_dsa_key -N ""
```

[**/etc/shosts.equiv**]

一行ごとにホストを書きます.

```
node1.sample.ac.jp
111.111.111.111 user1
node2
```

上の例では

1. node1.sample.ac.jp からの接続を許可.
2. 111.111.111.111 からユーザ user1 のみのログインを許可.
3. ホスト node2 からの接続を許可.

という設定になります.

[/etc/ssh/ssh_known_hosts2]

ホストベース認証を許可するホストの公開鍵を置くファイルです.

```
$ cat <公開鍵ファイル> >> /etc/ssh/ssh_known_hosts2
```

を実行して追加してください. cp などによってファイルを上書きしないように注意してください.

```
ssh-dss AAAAB3NzaC1kc3MAAACBAJ4gcPqz2A4X4yDUfnlgFNgCMrbzALlTd9l4doGYJrDCYD
oOwgUkRrmf14yeQKct9SSXPGVsc88N6a6eRwKBWGrm7MGaTNMwllY+w0eDIaFXgbFXGqx1pnz4
esiffS5f9ap3/pVXXaoCTSGXg/wN8hqaG58OY8bySTX6HK3JWTZ1AAAAFQCzcweCvbIyLJqidT
inGYradMb/CwAAAIBMF2k+acHAAn+md51WUWru2p+e4jV7+SwyS7wdh2lwhfB81qOdGvP8BgY9
DceK5P3srrnC6ZlUt67IyuKD6VxsUce8L8XELsHlRYoE0+w+WaPnaXAzxMisPFZGpClGnzsutN
RSUMNq15xszztqHQAhXX2dE/iAAf94W8qi5BDqbAAAAIAveEucknapYS2VUtuck7qyT21DuVdZ
jgEqVXlWHFxr06mrdWq7Hp3lRXF6++WYmK/QWUR8BjngXQ1PFCYW9JAiyQIWFBrWfWtbElmR6g
r/qXZOsNXNjoDxdDG5SwXMyJbhJaf3aXK7YGvq9FKSCEoNjadQ19VITmw6C8dYFNh8yQ=
```

追加した公開鍵が上のように ssh-dss の前にホスト名などが記入されていない場合, editor でホスト名, IP address, FQDN を以下のように記入してください.

```
node2, node2.sample.ac.jp,222.222.222.222 ssh-dss AAAAB3NzaC1kc3
MAAACBAJ4gcPqz2A4X4yDUfnlgFNgCMrbzALlTd9l4doGYJrDCYDoOwgUkRrmf14yeQKct9SSX
PGVsc88N6a6eRwKBWGrm7MGaTNMwllY+w0eDIaFXgbFXGqx1pnz4esiffS5f9ap3/pVXXaoCTS
GXg/wN8hqaG58OY8bySTX6HK3JWTZ1AAAAFQCzcweCvbIyLJqidT inGYradMb/CwAAAIBMF2k
+acHAAn+md51WUWru2p+e4jV7+SwyS7wdh2lwhfB81qOdGvP8BgY9DceK5P3srrnC6ZlUt67Iy
uKD6VxsUce8L8XELsHlRYoE0+w+WaPnaXAzxMisPFZGpClGnzsutNRSUMNq15xszztqHQAhXX2
dE/iAAf94W8qi5BDqbAAAAIAveEucknapYS2VUtuck7qyT21DuVdZjgEqVXlWHFxr06mrdWq7H
p3lRXF6++WYmK/QWUR8BjngXQ1PFCYW9JAiyQIWFBrWfWtbElmR6gr/qXZOsNXNjoDxdDG5SwX
MyJbhJaf3aXK7YGvq9FKSCEoNjadQ19VITmw6C8dYFNh8yQ=
```

[/etc/ssh/sshd_config]

ホストベース認証を許すように設定します.

```
Port 22
Protocol 2,1
#ListenAddress 0.0.0.0
#ListenAddress ::

# HostKey for protocol version 1
HostKey /etc/ssh/ssh_host_key
# HostKeys for protocol version 2
HostKey /etc/ssh/ssh_host_rsa_key
HostKey /etc/ssh/ssh_host_dsa_key

# Lifetime and size of ephemeral version 1 server key
#KeyRegenerationInterval 3600
#ServerKeyBits 768

# Logging
#obsoletes QuietMode and FascistLogging
#SyslogFacility AUTH
SyslogFacility AUTHPRIV
#LogLevel INFO

# Authentication:

#LoginGraceTime 600
#PermitRootLogin yes
PermitRootLogin no
#StrictModes yes

RSAAuthentication yes
PubkeyAuthentication yes
#AuthorizedKeysFile .ssh/authorized_keys

# rhosts authentication should not be used
RhostsAuthentication no
# Don't read the user's ~/.rhosts and ~/.shosts files
IgnoreRhosts yes # ユーザレベルでのホストベース認証をさせない
# For this to work you will also need host keys in /etc/ssh/ssh_known_hosts
RhostsRSAAuthentication no
# similar for protocol version 2
HostbasedAuthentication yes # ホストベース認証を許可する
# Change to yes if you don't trust ~/.ssh/known_hosts for
# RhostsRSAAuthentication and HostbasedAuthentication
IgnoreUserKnownHosts yes # ユーザレベルでのホストベース認証をさせない
```

```
# To disable tunneled clear text passwords, change to no here!
PasswordAuthentication yes
PermitEmptyPasswords no

# Change to no to disable s/key passwords
#ChallengeResponseAuthentication yes

# Kerberos options
#KerberosAuthentication no
#KerberosOrLocalPasswd yes
#KerberosTicketCleanup yes

#AFSTokenPassing no

# Kerberos TGT Passing only works with the AFS kaserver
#KerberosTgtPassing no

# Set this to 'yes' to enable PAM keyboard-interactive authentication
# Warning: enabling this may bypass the setting of 'PasswordAuthentication'
#PAMAuthenticationViaKbdInt yes

#X11Forwarding no
X11Forwarding yes
#X11DisplayOffset 10
#X11UseLocalhost yes
#PrintMotd yes
#PrintLastLog yes
#KeepAlive yes
#UseLogin no
UsePrivilegeSeparation yes # ホストベース認証をするときは必要
#Compression yes

#MaxStartups 10
# no default banner path
#Banner /some/path
#VerifyReverseMapping no

# override default of no subsystems
Subsystem sftp /usr/libexec/openssh/sftp-server
```

【再起動】

サーバ, クライアントの sshd サーバを再起動します. 以上で設定完了です.

【管理者向け】

初期設定ファイル ~/.bashrc においては, あらかじめ /etc/skel に対して反映させておくと, ユーザを追加する度に毎回設定をしなくてすみます.

【トラブルシューティング】

以上のことを行っても上手くいかない場合, コマンド /usr/bin/ssh のユーザ権限が「s」になっているか確かめてください. なっていない場合, 以下のコマンドを実行してください.

```
$ chmod u+s /usr/bin/ssh
```

2.2 LAM のインストール

2.3 Gigabit Network の設定

2.3.1 はじめに

研究室に GigaBit 環境を取りいれるための設定です。最低今までの 3 倍 (?) の速度がでて、よりよい環境でコンピュータを扱えると思います.

1000MB での通信をする場合、それに対応した NIC(LAM カード) が必要になります。そして、NIC を使うためのドライバーをインストールする必要があります。

2.3.2 作業手順

以下は作業手順と方法です。行う作業としては 4 つの項目があります。

【kernel のインストール】

1. root になる.
2. ディレクトリ /home/public/Kernel_Gigabit/ に移動する.
3. kernel をインストールする.

【LILO の設定】

1. root になる.
2. エディタでファイル /etc/lilo.conf を開く.
3. エディタで開いたら、下のような感じの内容がでてきます。以下のように変更してください。

「変更 1」

```
prompt
timeout=50
default=linux
boot=/dev/hda
map=/boot/map
install=/boot/boot.b
message=/boot/message
linear

image=/boot/vmlinuz-2.4.19-0vl11    <------ コピー (この行から)
        label=linux
        read-only
        root=/dev/hda2              <------ コピー (この行まで)
```

「変更 2」

```
prompt
timeout=50
default=linux
boot=/dev/hda
map=/boot/map
install=/boot/boot.b
message=/boot/message
linear

image=/boot/vmlinuz-2.4.19-0vl11
        label=linux
        read-only
        root=/dev/hda2

image=/boot/vmlinuz-2.4.19-0vl11    <---- 張り付けた (始
        label=linux
        read-only
        root=/dev/hda2              <---- 張り付けた (終
```

「変更 3」

```
prompt
timeout=50
default=linux
boot=/dev/hda
map=/boot/map
install=/boot/boot.b
message=/boot/message
linear

image=/boot/vmlinuz-2.4.19-0vl11
        label=linux-old             <---- 変更した。
        read-only
        root=/dev/hda2

image=/boot/vmlinuz                 <---- 変更した。
        label=linux
        read-only
        root=/dev/hda2
```

コマンド /sbin/lilo を実行してください. もしファイル /etc/lilo.conf の設定がおかしい場合, error メッセージがでますので, もう一度確認してみてください.

【NIC(LAM ボード) をつける】

1. まずコンピュータを停止する.
2. システムが停止した状態を確認して電源コンセントを抜く。
3. コンピュータに NIC をつける。(静電気に注意。作業をする前になにか金属に触れておくなどする)
4. 電源コンセントを入れ、コンピュータを起動する。

【netcfg で設定】

1. 電源コンセントを入れ, コンピュータを起動する.
2. 新しいカーネルを選択する.
3. root でログインする.
4. コマンド dmesg を実行して、eth1 か eth0 かを調べる.
5. コマンド netcfg を実行し、インタフェースのタブを押す.
6. 先ほど調べた eth に関して無い場合は追加, ある場合はそれの変更をする.
7. 追加や変更内容は以前使っていた設定と同じにする.
8. 設定を保存して、終了する。
9. コマンド「/etc/init.d/network restart」を実行する.

2.3.3 終わりに

以上で設定は終了となります. なにかおかしなことになったり、よくわからない点があれば聞いてください。あらかじめ言ってもらえれば、この作業をしますので、連絡してください。おかしい点があれば、教えてください。

2.4 Intel Compiler の設定

2.4.1 まえがき

LAM は, デフォルトで gcc コンパイラを使います.gcc より実行速度の早くなるコンパイラに Intel 製のコンパイラをインストールします.

2.4.2 変更するメリット

Intel 製のコンパイラは gcc に比べて２〜４割の実行時間の短縮を見込めます.

2.4.3 gcc から Intel 製のコンパイラへ

インストール

1. ディレクトリ /home/public/MPI/Intel_Compiler/ にあるファイル intel_ compiler.tgz をひろってくる.
2. コマンド $ su - で root になる.
3. ファイル intel_compiler.tgz を展開して ./install を実行。
4. is32 のコンパイラを選ぶ (ライセンスに関しては、全て「accept」で良い)
5. インストールディレクトリは default(/opt) で良い。(/usr/local なども可)

2.4.4 管理者向け

初期設定ファイル ~/.bashrc においては, あらかじめ /etc/skel に対して反映させておくと, ユーザを追加する度に毎回設定をしなくてすみます.

第3章　操作方法(ユーザ向け)

3.1　研究室内編

3.1.1　最初にしておく設定

LAM で SSH, Intel Compiler を利用するための設定を行っておく必要があります. LAM は, インストールした時点で rsh と gcc を使って並列計算することができます. rsh はセキュリティを考えるとできるだけ使用したくありません. gcc より実行速度の早くなるコンパイラを利用したい. これらのことから rsh を ssh に, gcc を Intel 製のコンパイラに変更します.

【変更するメリット】

- SSH の利用

  ssh は秘密鍵, 公開鍵を使って通信内容の暗号化, なりすまし対策を行えます.

- Intel 製コンパイラの利用

  Intel 製のコンパイラは gcc に比べて２〜４割の実行時間の短縮を見込めます.

【設定の変更】

[rsh から ssh へ]

ファイル ~/.bashrc に以下の行をを追加してください.

```
LAMRSH="ssh -x"
export LAMRSH
```

コマンド $ source ~/.bashrc を実行して, 設定を反映させてください.

[gcc から Intel 製のコンパイラへ]

PATH の設定

インストールしたら, ディレクトリ /opt 下に配置されているので, PATH を追加します. ファイル ~/.bashrc に以下の項目を追加してください.

```
PATH=$PATH:/opt/intel/compiler70/ia32/bin
export PATH
```

これで Intel 製のコンパイラを利用できるようになりました. 次に LAM で Intel 製のコンパイラを使えるようにします. ホームディレクトリ~/.bashrc に同様に以下の行を加えてください.

```
LAMHCC=icc
export LAMHCC
```

コマンド$ source ~/.bashrc を実行して, 設定を反映させてください.

3.1.2 手順

流れとしては以下のようになります.

1. 並列計算させるコンピュータを決める.
2. プログラムを実行できるようにする (LAM の起動).
3. プログラムを実行する.
4. 並列計算を終える.

あらかじめプログラムをコンパイルし, 実行ファイルが必要になります (参照)

3.1.3 設定, 操作方法

【並列計算させるコンピュータを決める】

計算させるホストを箇条書にします. 以下のようにファイル ~/lamhosts を作成してください.

```
node1.sample.ac.jp
node2.sample.ac.jp
node3.sample.ac.jp
node4.sample.ac.jp
```

このようにファイル ~/lamhosts に載せたホストは並列計算する候補でしかありません. というのは, プログラムを実行するとき, オプションを指定することで, 候補のうちから選択して計算を行うことができ, 資源 (CPU, メモリなど) を全て使用しなくてよいです.

【プログラムを実行できるようにする (LAM の起動)】

次にコマンド $ lamboot によって LAM の起動を行います. オプション「-v」は, スタート開始時のプロセスの報告をさせます.

```
$ lamboot -v lamhosts

LAM 7.0.3/MPI 2 C++/ROMIO - Indiana University

n0<23808> ssi:boot:base:linear: booting n0 (node1.sample.ac.jp)
n0<23808> ssi:boot:base:linear: booting n1 (node2.sample.ac.jp)
n0<23808> ssi:boot:base:linear: booting n2 (node3.sample.ac.jp)
n0<23808> ssi:boot:base:linear: booting n3 (node4.sample.ac.jp)
n0<23808> ssi:boot:base:linear: finished
$
```

上の例の最後に「finished」という出力があることで成功したことがわかります.

【プログラムを実行する】

実行するにはコマンド $ mpirun を使います. 以下は, ファイル~/lamhosts で指定したホスト全てで計算を行います.

```
$ mpirun N <object file>
```

オプション「N」 は全てのホストを示しています.

以下のように特定のホスト (n0 と n1 と n4) を選択することも可能です.

```
$ mpirun n0,1,4 <object file>
```

並列計算を行わせるホスト全てに実行ファイルをおいておく必要があります. あらかじめ, scp などでホームディレクトリにおいておいてください. しかし, 以下のようにオプション「-s h」を指定することで, 実行ファイルを並列計算するホストに転送して実行することができます.

```
$ mpirun n1,3 -s h <object file>
```

【並列計算を終える】

並列計算 (LAM) を終えるとき, 以下のコマンドを実行してください.

```
$ lamhalt
```

コマンド lamhalt で正常に終了できなかった (エラーがでた) 場合, 以下のコマンドで終了させてください.

```
$ wipe -v ~/lamhosts
```

3.2 情報演習室編

3.2.1 最初にしておく設定

以下の作業は www.info.sci.yamaguchi-u.ac.jp, alpha.info.sci.yama guchi-u.ac.jp, is00.sci.yamaguchi-u.ac.jp のいずれかで行ってください.

【ディレクトリ bin の作成と PATH の設定】

自分のホームディレクトリに bin というディレクトリを作成してください.

```
$ mkdir ~/bin
```

作成したディレクトリに PATH を通します. やり方はファイル ~/.bashrc に以下の文を記入してください.

```
PATH=$PATH:~/bin
export PATH
```

最後に以下の文を実行して終りです.

```
$ source ~/.bashrc
```

【lam のためのコマンドをインストール】

http://bcl.sci.yamaguchi-u.ac.jp/lab/texts/info_mpi.tgz をダウンロードしてください (研究室内からしかアクセスできないことに注意).

```
$ wget bcl.sci.yamaguchi-u.ac.jp/~yosi/internal/Data/info_mpi.tgz
```

次にそのファイルをディレクトリ ~/bin で展開します.

```
$ tar zxvf info_mpi.tgz
```

最後に, 以下のファイルがあることを確認してください.

1. BOOT.sh
2. all_node_reaction_ping.sh
3. mpihosts.dat
4. wakeupPC.pl
5. config(ssh 用)

【ssh を使うために】

ファイル ~/.bashrc に以下の文を記入してください.

```
LAMRSH="ssh -x"
export LAMRSH
```

設定を反映させるために次の文を実行してください.

```
$ source ~/.bashrc
```

ディレクトリ ~/.ssh に /bin/config を移動させてください.

```
$ mv ~/bin/config ~/.ssh/
```

ディレクトリがない場合は以下のようにして作成してください.

```
$ mkdir ~/.ssh
$ chmod 700 .ssh
```

【並列計算させるホストの決定】

ファイル ~/bin/mpihosts.dat には, 並列計算させる候補を記述しておきます. あらかじめホスト is00 から is10 までの 11 台を載せています.

3.2.2 使い方

【並列計算を行うまでの操作】

以下の作業は, Lam を使用する前に行うことです.

1. ホスト is00.sci.yamaguchi-u.ac.jp(IP:133.62.213.130) にログインする.

   –注意–

   ホスト is00 が動作していないときは, www.info.sci.yam aguchi-u.ac.jp, もしくは alpha.info.sci.yamaguchi-u. ac.jp にログインして, 以下のコマンドで起動してください.

   ```
   $ wakeupPC.pl is00
   ```

2. コマンド BOOT.sh を実行する.(ホストの起動に数分かかる)

3. 数分後にコマンド all_node_reaction_ping.sh を実行する.

4. コマンド lamboot -v ~/active.dat を実行する.

【並列計算を終えるときの操作】

Lam(並列計算) を終了するとき以下のコマンドを実行してください.

```
$ lamhalt
```

コマンド lamhalt でエラーが生じた場合, 以下のコマンドで終了してください.

```
$ wipe -v ~/active.dat
```

3.2.3 コマンドの説明

【wakeupPC.pl】

ホストの電源が落ちている場合は, コマンド wakeupPC.pl を使って起動させてください.

```
$ wakeupPC.pl <hostname>
```

【あとがき】

コマンド all_node_reaction_ping.sh や BOOT.sh はシェルスクリプトなので実際に中を見て, 動作を確認した上で使用することをお薦めします.

3.3 コンパイル

LAM/MPI はラッパコンパイラを用意してくれています. これによって,MPI に関するヘッダのリンクなどを考えずにコンパイルができます. 各ラッパコンパイラは以下のようになります.

- mpicc - C プログラム
- mpiCC,mpic++ - C++プログラム
- mpif77 - Fortran プログラム

ラッパコンパイラは以下のように gcc とほぼ同様に扱えます.

```
% mpicc -o sample sample.c
```

オプション (-showme) を指定することで, 実際に実行されている中身を見ることができます.

```
$ mpicc -O sample sample.c -showme
gcc -pthread -O sample sample.c -llammpio -llamf77mpi -lmpi -llam -lutil -pthread
```

3.3.1 C のサンプルプログラム

以下のプログラムは,Hello World です.

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

int main(int argc, char *argv[])
{
   int rank, size;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   printf("Hello Workd! I am %d of %d\n", rank, size);

   MPI_Finalize();

   return 0;
}
```

上のプログラムをコンパイルするには以下のコマンドラインになります.

```
$ mpicc -o hello hello.c
```

3.3.2 C++のサンプルプログラム

```
#include <iostream>
#include <mpi.h>

using namespace std;

int main(int argc, char *argv[])
{
   int rank, size;

   MPI::Init(argc, argv);
   rank = MPI::COMM_WORLD.Get_rank();
   size = MPI::COMM_WORLD.Get_size();

   cout << "Hello Workd! I am " << rank << " of " << size << endl;

   MPI::Finalize();

   return 0;
}
```

上のプログラムをコンパイルするには以下のコマンドラインになります.

```
$ mpiCC -o hello hello.cc
```

もしくは

```
$ mpic++ -o hello hello.cc
```

3.4 MPI プログラムの実行

3.4.1 mpirun

mpirun コマンドは並列プログラムを実行するのにさまざまなオプションを持っています. 以下少数ですが, とりあげていきます.

ブートスキーマの全ての CPU で並列計算を行う場合, 以下のようになります.

```
$ mpirun C hello
```

全てのノード上で並列計算を行う場合は以下のようになります.

```
$ mpirun N hello
```

これはマルチスレッド MPI プログラムに便利でしょう.

最後に, プログラムを走らせたいプロセスの数を決める方法です. これは LAM ユニバースにおいて CPU やノードの数がどれだけあるかは関わりありません.

```
$ mpirun -np 4 hello
```

-ssi は MPI プロセスにモジュールを指定するのに使用される.

```
$ mpirun -ssi rpi usysv C hello
```

上の例では RPI モジュールの usysv を指定して実行しています.

コマンドラインからプログラムに引数を渡すには以下のようにします.

```
$ mpirun -ssi rpi usysv C hello arg1 arg2 arg3
もしくは
$ mpirun -ssi rpi usysv C hello -- arg1 arg2 arg3
```

コマンドラインの最後に指定することに注意してください.

あと LAM ユーザガイドの 51 ページの編集!

ここでは取り上げていませんが,mpirun も MPMD プログラムをサポートしています. 詳しくは man mpirun(1) を見て下さい.

3.4.2 mpiexec

MPI-2 では mpiexec を推奨しています.mpiexec は mpirun に似ていますが,mpirun に有効なオプションだけど,mp iexec には無効なものがいくつかあります. 以下大雑把に mpiexec の都合の良いことを上げます.

- MPMD プログラムを実行するとき.
- ヘテロジーニャス (異機種間) で並列化するとき.
- lamboot, 実行,lamhalt の 3 つの処理を一括りにして MPI プログラムを実行するとき.

一般的な mpiexec の文法は以下のようになります.

```
$ mpiexec < global_options > < cmd1 > : < cmd2 > : ...
```

「:」の前後にスペースがいることに注意してください.

MPMD プログラムの実行

サーバ/クライアント並列化プログラム (server,client) を実行するとき,2 つの実行ファイルが必要になります.

```
$ mpiexec -n 1 server : client
```

LAM ユニバースにおいて, 全ての CPU に「client」プログラムがコピー・実行され, 1 つの「server」プログラムがコピーされます.

ヘテロジーニャス (異機種間) での実行

詳細はマニュアル参照

一括り処理

LAM ユニバースの起動, プログラムの実行,LAM ユニバースの終了の 3 つの処理をバッチ処理によって行えます.

```
$ mpiexec -machinefile hostfile hello
```

上記の hostfile はブートスキーマ (boot schema) です. このコマンドによって, 全ての CPU で hello プログラムが実行されます.

3.5 MPI コマンドリファレンス

3.5.1 lamnodes

コマンド lamboot で Lam[deamon] を起動したホスト, ID を調べるには, 以下のコマンドを実行することで確認できます.

```
$ lamnodes
n0 node1.lam.com:1:origin,this_node
n1 node2.lam.com:1:
n2 node3.lam.com:2:
n3 node4.lam.com:2:
```

上の一番左の列は LAM のノード番号です.n3 は node4 を示しています. 第 3 列目の数字はロックした CPU の数を示しています. 合計 6CPU になっています.

origin というのは, どこのノードが lamboot を起動したかを示しています. 上の場合では,node1 で lamboot が実行されたことがわかります.

this_node というのは, どこで lamnodes を実行したかがわかります. 上の例では,node1 で lamnodes を実行していることになります.

3.5.2 mpirun

MPI アプリケーションは mpirun によって実行される. 実行の方法には 2 通りある.

1. SPMD のアプリケーション

   簡単な SPMD プログラムは mpirun のコマンドラインから起動することができる.

2. アプリケーションスキーマ

   アプリケーションスキーマは, アプリケーションを構成しているそれぞれのプログラムごとに一行ずつ記述する. それぞれのプログラムのために,mpirun コマンドラインと同じ構文のオプションによって示される 3 つの指定を与えれる.

アプリケーションスキーマを用いる場合の文法は以下になる.

```
$ mpirun -v <my_app_schema>
```

以下アプリケーションスキーマについての構文のオプション 3 つを示した.

- -s< ノード識別子 >

  実行プログラムがあるノードを指定する. これが指定されないとき, LAM はプログラムの実行ノードにおいてプログラムを探す.

- < ノード識別子 >

  プログラムを実行するノードを指定する. これが指定されないとき, 全てのノードで実行する.

- -c < # >

  与えられたノードで生成するプロセスの数を指定する. これが指定されないとき,LAM は与えられたノードに 1 つずつプロセスを生成する.

【直接通信】

-c2c　　MPI 通信において LAM デーモンをバイパスする.

MPI のライブラリの「クライアント間直接通信」の機能は, 監視と制御の犠牲にすることによって, ハードウェアのもっとも速いスピードをださせる. 送られても受信できないメッセージは受信側のバッファに置かれる.
アプリケーションは最初デーモンを使う通信を使いデバッグし, その後で直接通信を利用すると良い.
アプリケーションは最初デーモンを使う通信を使いデバッグし, その後で直接通信を利用すると良い.

3.5.3 lamclean

全てのホストで実行されているプロセスを殺す場合, lamclean を使います. コマンド kill では全てのホスト上のプロセスを殺すことはできません.
以下のように実行してください.

```
$ lamclean -v
```

3.5.4 wipe

コマンド $ wipe -v lamhosts を実行することによって, LAM を強制終了させます. このコマンドは lamhalt が失敗した場合など, LAM を終了する際の最後の砦として考えてください.

```
$ wipe -v lamhosts

LAM 7.0/MPI 2 C$++$/ROMIO - Indiana University

n0<9167> ssi:boot:base:liner:booting n0 (node1)
n0<9167> ssi:boot:base:liner:booting n1 (node2)
n0<9167> ssi:boot:base:liner:booting n2 (node3)
n0<9167> ssi:boot:base:liner:booting n3 (node4)
```

3.5.5 tping

コマンド $ tping は, 通信する経路があること, メッセージが送受信できることを確かめるものです. オプション N はブロードキャストを指定します. もし, tping が失敗した場合は, 「Control-Z」をおして, wipe コマンドで終了し, 最初 (セッションのリスタート) からやり直してください.

3.5.6 recon

ファイル ~/lamhosts に記載した並列計算候補に LAM がインストールされていること, アクセス可能なことをコマンド $ recon によってチェックする必要はありません. 以下は, aris と titan を候補としてコマンドを実行した結果です.「Woo hoo!」以下の文が出力され, 成功したことがわかります.

```
$ recon -v lamhosts
n0<23821> ssi:boot:base:linear: booting n0 (node1.sample.ac.jp)
n0<23821> ssi:boot:base:linear: booting n1 (node2.sample.ac.jp)
n0<23821> ssi:boot:base:linear: booting n2 (node3.sample.ac.jp)
n0<23821> ssi:boot:base:linear: booting n3 (node4.sample.ac.jp)
n0<23821> ssi:boot:base:linear: finished
-----------------------------------------------------------------------------
Woo hoo!

recon has completed successfully.  This means that you will most likely
be able to boot LAM successfully with the "lamboot" command (but this
is not a guarantee).  See the lamboot(1) manual page for more
information on the lamboot command.

If you have problems booting LAM (with lamboot) even though recon
worked successfully, enable the "-d" option to lamboot to examine each
step of lamboot and see what fails.  Most situations where recon
succeeds and lamboot fails have to do with the hboot(1) command (that
lamboot invokes on each host in the hostfile).
-----------------------------------------------------------------------------
```

3.5.7 lamboot

lamboot は LAM セッションを開始する. ホストファイルの形式で記述されたブートスキーマファイルが lamboot の第一の引数になる.

```
$ lamboot -v < boot schema >
```

LAM には耐故障機能がついており, 障害のあるノードを検出し, 取り去ることができる. 耐故障機能の指定は,lamboot でセッションを開始するときに, オプション -x をつける.

```
$ lamboot -v < boot schema > -x
```

この耐故障機能は, 全てのノードの間で周期的な「心拍」メッセージを行うことで実現している. 任意のノードの停止が確認されると LAM は自動でそのノードを取り去る. 停止したノードのノード識別子は無効になり, その ID は空いたままになる. デフォルトでは, 停止したノードに対して対応せずに, 無限に再送信を繰り返すことになる.

なお, オプション d で詳細なデバッグ情報が得れる.

```
$ lamboot -d < boot schema >
```

3.5.8 mpitask

mpitask コマンドは,UNIX/LINUX の ps コマンドに似ています. mpitask は LAM ユニバーサルで実行されている MPI プログラムの現在の状態をみれ, それぞれの MPI 関数が現在実行されていることについての素朴な情報を見れます. 普通

の実行において,mpimsg コマンドだけが MPI プロセス間のデータ転送の一部始終を寸見することができます.

本当にメッセージトラフィックをデバッグしたいなら, メッセージパッシングアナライザ (XMPI など) や並列デバッガ (TotalView:有償) を使うと良い.

3.5.9 mpimsg

mpimsg コマンドはあまり有用なものではない. いずれ LAM/MPI ではなくなるらしい. 詳しくは man で.

3.5.10 lamexec

lamexec は mpirun と似ている. 違うのは, mpirun が MPI プログラムを実行するものであるのに対し, lamexec は非 MPI プログラムを実行する. 例えば,

```
$ lamexec N uptime
 午後 05 時 32 分   稼働  13 日間，19 時 53 分，  2  ユーザ，  負荷平均率：0.00，0.00，0.00
 午後 05 時 32 分   稼働  14 日間，  2 時 36 分，12  ユーザ，  負荷平均率：0.00，0.00，0.00
 午後 05 時 32 分   稼働  21 日間，  9 時 49 分，  7  ユーザ，  負荷平均率：0.00，0.01，0.00
```

とすることができる. mpirun に有効なほとんどのパラメタやオプションは lamexec にも使用できる.

3.5.11 lamgrow

lamgrow コマンドは LAM ユニバースに 1 つのノードを新規に加える. 注意点として以下がある.

- 最初に LAM ユニバースを起動するのに使った boot モジュールを使うこと.
- lamgrow は, もうすでにある LAM ユニバースでのノードから実行すること.

例として, 既に boot モジュール rsh で起動されている LAM ユニバースに newnode を加えるには以下のコマンドになる.

```
$ lamgrow -ssi boot rsh newnode.sample.ac.jp
```

以下に部分的なパラメタを示す.

- -v : 詳しく内容を表示する
- -d : デバッグモード
- -n $<$ *nodeid* $>$: 新しいノードに $<$ *nodeid* $>$ を指定する. $<$ *nodeid* $>$ は未使用のものを指定しなければならない. もし,-n が指定されなかった場合, 未使用の一番小さな node id になる.
- -no-schedule : boot スキーマにおいて,「no_schedule=yes」と同じ.mpirun や mpiexec において使われる C,N でこのノードは含めないことになる.
- -ssi $<$ *key* $><$ *value* $>$: SSI パラメタ $<$ *key* $>$ に値 $<$ *value* $>$ を通す.
- $<$ *hostname* $>$: 既に存在する LAM ユニバースに $<$ *hostname* $>$ のノードを加えるのに指定する.

3.5.12 lamshrink

lamshrink コマンドは, LAM ユニバースからノードを削除するのに使用される. 例として,

```
$ lamshrink n2
```

上のように実行した場合,LAM ユニバースの n3 ノードを削除する. 削除されたノード ID より大きいノードの ID はそのままである, つまりその部分は空になったままである. この状態で mpirun や mpiexec のノード指定「N」,「C」を実行しても正常に動く (空になったノードは無視される).

3.5.13 lamhalt

並列計算 (LAM) を終えるときに使用します.

```
$ lamhalt
```

コマンド lamhalt で正常に終了できなかった (エラーがでた) 場合, 以下のコマンドで終了させてください.

```
$ wipe -v ~/lamhosts
```

3.5.14 laminfo

laminfo コマンドは, 現在インストールされている LAM のバージョン情報, lam のインストールディレクトリ, コンパイラ使用可否, SSI モージュールの利用不可を知ることができます.

```
$ laminfo
```

3.6 X の転送

ssh などでリモートホストにログインした場合, 基本的な操作は CUI(character user interface) で行うことになります. GUI(graphical user interface) を使いたい場合は以下を読んでください. ただし, 居場所の特定 (IP もしくは ホスト名) が可能なことが必要です. もちろん,MPI でも可能です.

3.6.1 環境変数 DISPLAY の設定

あなたは 「here」 というホストから 「there」 というホストにログインしていると仮定します. リモート上で次のコマンドを実行します.

```
$ DISPLAY=here:0.0
$ export DISPLAY
```

これで下準備は完了です. LAM で行う場合実行時にオプション「-x DISPLAY」を付けることで X の転送ができます.

```
$ mpirun -x DISPLAY N -s h program
```

もし上手くいかないなら, 実際にディスプレイに表示するホスト (この場合 here) で次のコマンドを実行してください.

```
$ xhost +there
```

ファイル /.bashrc などに環境変数の設定をするようにしておくのは便利かもしれません.

3.6.2 詳しくは

man ページの 「xauth(1)」などを参照してください.

第II部

MPI-GA

第4章　目的と過程

本文の主な目的は GA において並列化を行うことです. 並列化は MPI で行います. この有益性は以下の通りです.

- 個体数をより多く扱えること.
- GA における MPI のプログラム設計が容易であること.
- GA という形式を取ったものなら, さまざまな問題を扱える.

MPI のプログラムを作成する手順は, まず目的とした逐次的なプログラムの設計をして, それから MPI の並列計算するための設計を行うことになります. ベクトルやマトリックス計算などの場合, 1 つずつの計算過程の分割とそれにおけるデータの分割の 2 つを考える必要があります. しかし, GA の場合, 共通した評価式を利用することができ, 基本的に個体数を増やすだけで済みます. 分散型なのでメモリも安価に多く所有していることが他の並列計算方法と比べても良い点です. その 1 つの形式性を把握することでさまざまな問題にも取り組むことができます.

基本的な作業の過程を示します.

1. 任意の問題を GA を扱って解く判断をする.
2. 逐次的な GA のプログラムの設計および作成する.
3. 作成した逐次的なプログラムを, 単純な形式によって並列化 (個体数を増やす) を行う.

GA と MPI の融合によるものであるので, まずそれぞれについて解説します. その解説した内容の上で GA における MPI を利用したプログラムの説明を行います.

第5章　メッセージ通信

5.1　メッセージ通信の初見

並列計算を行う目的はただ1つ, 計算能力の獲得です. その獲得の仕方でも何種類かあります. 一台のマシンでメモリを共有し, 複数個の CPU によって並列計算を行うものや, メッセージ通信のようにフォンノイマン形式のパソコンをネットワークによって複数個接続したものなどがそうです. メッセージ通信の強さも弱さも, ethern et などのネットワークに接続することに起因しています. そんな中でメッセージ通信はメモリを共有し, 複数個の CPU をもった1つのマシンさえも取り入れることができる, その計算能力を統合できるものとして強力に感じます.

5.2　MPI プログラムのおまじない

C 言語によるプログラミングを習ったとき, まず最初におまじないとして習ったことがあると思います. 似たように MPI にもおまじないがあります.

```
#include<stdio.h>
#include "mpi.h"    // MPI に必要なヘッダ

main(int argc, char *argv[]) {
   int my_rank;        // それぞれのパソコンを特定する ID(0 から順に)
   int p_sum;          // プロセスの数
   int source;         // 送信プロセスのランク
   int dest;           // 受信プロセスのランク
   int tag = 0;        // メッセージのタグ
   MPI_Status status;  // 受信の戻りステータス

   MPI_Init(&argc, &argv);                      // MPI の始まり

   // 自分のランクを取得
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   // プロセスの数を求める
   MPI_Comm_size(MPI_COMM_WORLD, &p_sum);

   //
   // 実行する処理部分
   //

   MPI_Finalize();                              // MPI の終わり

   return 0;
}
```

C 言語のおまじない+MPI のおまじみたいなものになっています. MPI_Comm_rank によってそれぞれのパソコンに ID がふられ, MPI_Comm_size によって計算するパソコンの総数がわかります. この ID や総数に対して実行処理部分で if 文などで制御を行います. このように 1 つのプログラムで複数の処理を行うことを SPMD(Single Program Multiple Data) といいます. 以降,SPMD を前提に並列化を行います.

5.3 メッセージ通信をやってみる

それでは少し触れてみましょう. 二つのプロセスによるメッセージ通信をします. ランク 1 のノードからランク 0 のノードへ int 型の整数を送信して, 出力するプログラムを以下に示します (ランクとはそれぞれのノードの整数で表される識別子です.0 から割り当てられていくことに注意).

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

main(int argc, char *argv[]) {
   // 基本的な MPI の宣言
   int my_rank;        // 自分のランク
   int p_sum;          // プロセスの総数
   int source;         // 送信プロセスのランク
   int dest;           // 受信プロセスのランク
   int tag = 0;        // メッセージのタグ
   MPI_Status status;  // 受信の戻りアドレス

   // 計算上必要な宣言 (送受信するデータ)
   int num = 0;

   // MPI の開始
   MPI_Init(&argc, &argv);

   // それぞれのランクを決定
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   // プロセスの数を求める
   MPI_Comm_size(MPI_COMM_WORLD, &p_sum);

   // 目的の処理開始 ----------------------------------------------------

   if(my_rank != 0) {
      // 以下 ランク 1 のノードの処理

      // ランク 0 に向けて 送信先 (destination) を指定
      dest = 0;   // to rank 0

      // 送信データを 10 とする.
      num = 10;

      // 送信操作 (num を送信)
      MPI_Send(&num, 1, MPI_INT, dest, tag, MPI_COMM_WORLD);
```

```
    } else {
        // 以下 ランク 0 のノードの処理

        // ランク 1 から 受信先 (source) を指定
        source = 1;   // from rank 1

        // 受信前の num の値を 1 とする .
        num = 1;

        // 受信前の num を表示
        printf("(通信終了前) my_rank : %d , num = %d\n", my_rank, num);

        // 受信操作 (num を送信)
        MPI_Recv(&num, 1, MPI_INT, source, tag, MPI_COMM_WORLD, &stats);

        // 受信後の num を表示
        printf("(通信終了後) my_rank : %d , num = %d\n", my_rank, num);
    }

    // MPI の終了
    MPI_Finalize();

    return 0;
}
```

5.4 配列の送受信のやり方

5.5 構造体の送受信のやり方

ここで構造体の送受信についての話しをします. 構造体を使うことによって, 1 つ 1 つの個体とその要素の扱いが把握しやすくなると思います. しかし, MPI で構造体の送受信を扱う場合, 1 つの落し穴が存在します. それは, 構造体のそれぞれのメンバのデータの格納が非連続的であることに起因しています.

MPI の送受信関数は, 送受信するデータの格納されている開始アドレス, データの型, データの要素数の 3 つによって目的のデータを把握します. これは連続的に格納されているデータを送受信することが前提となっているからです. よって, 配列は連続性が保証されているので大丈夫なのですが, 構造体のそれぞれのメンバの連続性は保証されていないため, 目的に沿ったデータの送受信は保証されません.

問題は連続性でないということなので, 解決策の 1 つとしてひとまず構造体のデータを連続的な領域に格納し, それを送受信関数を使って送信, その後逆の手順で構造体に格納するという回りくどい方法があります. もう 1 つの解決策として, MPI の関数の中には非連続なデータ形式を 1 つの型として定義することによって, データがあたかも連続しているように扱うことができる関数をサポートしたものがあるので, それを扱えば, さほどのことではありません.

では実際に, 送受信プログラムを見てみましょう. これは, 書籍「MPI 並列プログラミング (P. パチェコ著) に載っていた例題プログラムです.

```
// 派生データ型を定義する関数
void Build_derived_type(double* a_ptr, double* b_ptr,
                        int* n_ptr, MPI_Datatype* mesg_mpi_t_ptr)
{
```

```
int block_length[3] = {1};   // 新しい型の各ブロックの要素数.
                             // a_ptr は double 型の要素で要素数 1.
                             // b_ptr も同様に要素数 1.
                             // n_ptr は int 型の要素で要素数 1.

// 新しい型の, 先頭からの各要素の変位.
// MPI_Aint(address int) は MPI で定義する C の型
MPI_Aint displacements[3];

// それぞれの型情報を示す配列
MPI_Datatype typelist[3];

// 変位を計算するのに用いる.( 変位 = address - start_address )
MPI_Aint start_address;   // a_ptr(start_address) が先頭
MPI_Aint address;         // 先頭の a_ptr を基準に変位を取る.( b_ptr, n_ptr )

// 2 つの double と 1 つの int からなる派生データ型を構築する.
typelist[0] = MPI_DOUBLE;   // a_ptr ( double 型 )
typelist[1] = MPI_DOUBLE;   // b_ptr ( double 型 )
typelist[2] = MPI_INT;      // n_ptr ( int    型 )


// 最初の要素 a は変位が 0
displacements[0] = 0;

// 他の要素の a との相対変位を計算するために a_ptr の先頭のアドレスを取る.
MPI_Address(a_ptr, &start_address);

// b のアドレスを取り, a から b の変位を求める.
MPI_Address(b_ptr, &address);
displacements[1] = address - start_address;

// n のアドレスを取り, a から n の変位を求める.
MPI_Address(n_ptr, &address);
displacements[2] = address - start_address;

// 派生データ型の構築 (決まり文句 1)
//
// 関数 MPI_Type_struct(
//          int count;                 ※ 派生データ型の要素のブロック数
//          int block_lengths[];       ※ 各要素のブロックサイズ
//          MPI_Aint displacements[];  ※ メッセージの先頭からの各要素の変位
//          MPI_Datatype typelist[];   ※ 各エントリの MPI データ型
//          MPI_Datatype* new_mpi_t;   ※ 生成された MPI データ型へのポインタを返す
//      )
//
MPI_Type_struct(3, block_lengths, displacements, typelist, mesg_mpi_t_ptr);
```

```
   // 派生データ型をコミットする (決まり文句 2)
   MPI_Type_commit(mesg_mpi_t_ptr);
}

void Get_data(double* a_ptr, double* b_ptr, int* n_ptr, int my_rank)
{
   MPI_Datatype mesg_mpi_t;

   if(my_rank == 0){
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_ptr, b_ptr, n_ptr);
   }

   Build_derived_type(a_ptr, b_ptr, n_ptr, &mesg_mpi_t);
   MPI_Bcast(a_ptr, 1, mesg_mpi_t, 0, MPI_COMM_WORLD);
}
```

MPI では派生データ型を任意に作成することができます. 派生データ型は, C 言語などでもお馴染の int, double などの型を組み合わせて, 任意のデータ配置の型を作成することができます. これを利用することにより, 配列やベクトル計算において, 要素ごとの分割を行い, データを局所的なものに分割し, 並列計算することができます. C 言語では, データを細部まで扱う必要はさほどありませんでしたが, MPI においては非常に重要になってきます. こういう点から MPI は「並列計算のアセンブリ言語」という名も持っています. 共有メモリ型の並列計算などでは, コンパイラが自動的に並列化を行うので, データを細部まで扱う必要はありません.

第6章　GA

遺伝的アルゴリズム (Genetic Algorithm;GA) は生物の進化の過程をもとにした最適化・探索アルゴリズムです.GA では遺伝子を持つ仮想的な生物の集団を計算機内に設定し, あらかじめ定めた環境に適応している個体が, 子孫を残す確率が高くなるよう世代交替シュミレーションを実行し, 遺伝子および生物集団を進化させる. GA は, 実際のプログラムの詳細を規定しない枠組みであるため, 各種の規則やパラメータの設定方法など, 不確定な要素が多い手法のため, 欠陥の指摘が多い. しかし, この不確定さながら, 応用範囲としては広いとも言える.

6.1　単純GA

単純 GA においてあらかじめ設定する必要のあるものがあります.

- 染色体と遺伝子型の決定
- 表現型の設定
- 環境との適応度の計算方法の決定

以上の設定が済んで, 進化の処理を行う. 図 6.1 に処理手順を表した.

(1) 手順 1：初期生物集団の発生

(2) 手順 2：各個体の適応度の計算

(3) 手順 3：淘汰および増殖の実行

(4) 手順 4：遺伝子型の交差の実行

(5) 手順 5：突然変異の実行

(6) 手順 6：生物集団の評価

以上の単純 GA をもとに $x^2 + y^2$ の最小値を求めるプログラムを作成しました.

6.2　最大・最小におけるGA

```
#include<stdio.h>
#include<math.h>
#include<time.h>
#include<stdlib.h>

#define N      10000    // 個体数
#define CRSS   0.3      // 交差率 (Cross over rate)
#define MTTN   0.03     // 突然変異率 (Mutation rate)
#define LAST_N 10       // 評価枠 (個体数) last number

enum {SLCT, MLTPL};  // SLCT  = SELECT(絶滅)
```

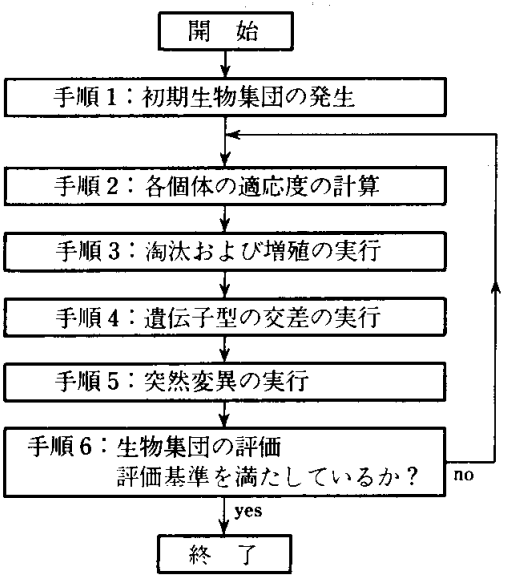

図 6.1: 単純 GA の処理手順

```
                    // MLTPL = MULTIPLICATION(増殖)

enum {RANKING, OUT};  // 最終的な評価における項目
                      //
                      // RANKING : 評価基準満たす
                      // OUT     : 基準を満たさない

typedef struct prm  // (x, y, z) parameters define
{
   double x;
   double y;
   double z;
   int dd_alv;
} OBJ;

double f(double x, double y)
{
   return ( x*x + y*y );
}

void srt(OBJ a[], int first, int last) // QUICK_SORT プログラム (リトルエイディ)
{
   int i , j;
   double cntr;   // cntr = center
   OBJ tmp_obj;

   cntr = a[ (first + last) / 2].z;
```

```
    i = first;
    j = last;

    while(1)
    {
        while(a[i].z < cntr)
            i++;

        while(cntr < a[j].z)
            j--;

        if(i >= j)
            break;

        tmp_obj = a[i];
        a[i]    = a[j];
        a[j]    = tmp_obj;

        i++;
        j--;
    }

    if(first < i-1)
       srt(a, first, i-1);

    if(last > j+1)
       srt(a, j+1, last);
}

void pri(OBJ b[])
{
    int i;

    srt(b, 1, N-1);

    // 評価を満たしたものを表示
    for(i = 1; i <= LAST_N; i++)
    {
          printf("%lf %lf %lf\n\n", b[i].x, b[i].y, b[i].z);
    }
}

void init_xy(OBJ a[])
{
    int i;
```

```
   for(i = 1; i < N; i++)
   {
      a[i].x =  drand48();
      a[i].y =  drand48();
      a[i].z = f( a[i].x, a[i].y);
      a[i].dd_alv = MLTPL;  // 生存することを前提にフラグを初期化
   }
}

// 適用度による生存, 絶滅
int slt_mlt(OBJ a[])
{
   int i;
   int dist_num = 0;    // 滅んだ数
   double P[N];         // 確率
   double avrg = 0.0;   // 平均値 (z)

   for(i = 1; i < N; i++)
   {
      avrg += a[i].z;
   }
   avrg /= N;

   for(i = 1; i < N; i++)
   {
      P[i] = a[i].z / avrg;

      if(P[i] > avrg)   // 平均値より大きかったら, 絶滅
      {
         a[i].dd_alv = SLCT;
         dist_num++;
      }
   }

   return dist_num;
}

// 交差 (crossover)
void crsscvr(OBJ a[])
{
   int i = 1, cnt;
   int fthr, mthr;   // fthr = father, mthr = mother
   double tmp_exch;   // 一時的なデータの格納 (temporary exchange)

   cnt = (int) N * CRSS;
```

```
   while(i <= cnt)
   {
      fthr = rand() % 71 + 1;
      mthr = rand() % 71 + 1;

      if( MLTPL == a[fthr].dd_alv && MLTPL == a[mthr].dd_alv)
      {
         tmp_exch = a[fthr].x;
         a[fthr].x = a[mthr].x;
         a[mthr].x = tmp_exch;
      }

      i++;
   }
}

// 突然変異 (mutation)
void mttn(OBJ a[])
{
   int i = 1, cnt;
   int tgt;        // 突然変異する要素 (tgt = target)

   cnt = (int) MTTN * N;

   while(i <= cnt)
   {
      tgt = rand() % 71 + 1;

      if( a[tgt].dd_alv == MLTPL)
      {
         a[tgt].x -= 0.1;
         a[tgt].y -= 0.1;
      }

      i++;
   }
}

// 失った obeject の分を補う
void add(OBJ a[])
{
   int i;

   for(i = 1; i < N; i++)
```

```
   {
      if(a[i].dd_alv == SLCT)
      {
         a[i].x = drand48();
         a[i].y = drand48();
         a[i].z = f( a[i].x, a[i].y );
         a[i].dd_alv == MLTPL;
      }
   }
}

int main(int argc, char *argv[])
{
   int dist_num;                    // 各々の世代の絶滅数
   long ct, gnr = 0;                // 世代交替の回数 (gnr = generation)
   OBJ obj[N];                      // オブジェクトの数
   double strt_tm, end_tm;          // 処理計測

   if(argc == 2)
   {
      // 世代数による評価のループ回数
      ct = atoi(argv[1]);
   } else{
      ct = 100;
   }

   // 種の初期化
   srand48(time(NULL)); srand(time(NULL));

   // (x, y) を初期化
   init_xy(obj);


   strt_tm = clock();
   // メイン処理 (開始)
   while(1)
   {
      // 1： selection or multiplication
      dist_num = slt_mlt(obj);


      // 2： 遺伝子の交差 (crosscover)
      if(dist_num != 0)
         crsscvr(obj);

      // 3： 突然変異 (mutation)
      if(gnr % 10 == 10)
```

```
            mttn(obj);

        // 4： 評価
        if(gnr >= ct)
            break;


        // 5： 補足
        add(obj);

        gnr++;
    }
    /* メイン処理 (終了) */
    end_tm = clock();

    /*結果の出力 */
    pri(obj);

    // 所要時間の出力
    printf("所要時間 %f [s]\n", (double) (end_tm - strt_tm) / CLOCKS_PER_SEC);

    return 0;
}
```

次にこの GA プログラムに MPI で計算するように改良します.

第7章　MPIによるGA

1つ前の章の逐次的なプログラムをMPIを使ったプログラムに変更します. 今回用いる並列化はN個のプロセスにおいて, それぞれ逐次的なGAを実行し, 後にそれぞれの個体をrank 0のプロセスに集め, 評価を行うというものです. この並列化は, 任意のプロセス数で扱うことができます.

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

#define N    10000    // 個体数
#define LAST_N 10     // 評価枠(個体数) last number
                      // ただし, N > LAST_N

#define CRSS 0.3      // 交差率(Cross over rate)
#define MTTN 0.03     // 突然変異率(Mutation rate)

#define ST_N 4        // 構造体の要素のブロック数

enum {SLCT, MLTPL};   // SLCT  = SELECT(絶滅)
                      // MLTPL = MULTIPLICATION(増殖)

enum {RANKING, OUT};  // 最終的な評価における項目
                      //
                      // RANKING : 評価基準満たす
                      // OUT     : 基準を満たさない

typedef struct prm    // (x, y, z) parameters define
{
   double x;
   double y;
   double z;
   int dd_alv;
} OBJ;

double f(double x, double y)
{
   return ( x*x + y*y );
}

void srt(OBJ a[], int first, int last) // QUICK_SORT プログラム(リトルエイディ)
```

```
{
   int i , j;
   double cntr;   // cntr = center
   OBJ tmp_obj;

   cntr = a[ (first + last) / 2].z;

   i = first;
   j = last;

   while(1)
   {
       while(a[i].z < cntr)
          i++;

       while(cntr < a[j].z)
          j--;

       if(i >= j)
          break;

       tmp_obj = a[i];
       a[i]    = a[j];
       a[j]    = tmp_obj;

       i++;
       j--;
   }

   if(first < i-1)
      srt(a, first, i-1);

   if(last > j+1)
      srt(a, j+1, last);
}

void pri(OBJ b[])
{
   int i;

   // 評価を満たしたものを表示
   for(i = 1; i <= LAST_N; i++)
   {
         printf("%lf %lf %lf\n\n", b[i].x, b[i].y, b[i].z);
   }
}
```

```
void init_xy(OBJ a[])
{
   int i;

   for(i = 1; i < N; i++)
   {
      a[i].x =  drand48();
      a[i].y =  drand48();
      a[i].z = f( a[i].x, a[i].y);
      a[i].dd_alv = MLTPL;  // 生存することを前提にフラグを初期化
   }
}

// 適用度による生存, 絶滅
int slt_mlt(OBJ a[])
{
   int i;
   int dist_num = 0;    // 滅んだ数
   double P[N];         // 確率
   double avrg = 0.0;   // 平均値 (z)

   for(i = 1; i < N; i++)
   {
      avrg += a[i].z;
   }
   avrg /= N;

   for(i = 1; i < N; i++)
   {
      P[i] = a[i].z / avrg;

      if(P[i] > avrg)   // 平均値より大きかったら, 絶滅
      {
         a[i].dd_alv = SLCT;
         dist_num++;
      }
   }

   return dist_num;
}

// 交差 (crossover)
void crsscvr(OBJ a[])
{
   int i = 1, cnt;
   int fthr, mthr;   // fthr = father, mthr = mother
   double tmp_exch;   // 一時的なデータの格納 (temporary exchange)
```

```
   cnt = (int) N * CRSS;

   while(i <= cnt)
   {
      fthr = rand() % 71 + 1;
      mthr = rand() % 71 + 1;

      if( MLTPL == a[fthr].dd_alv && MLTPL == a[mthr].dd_alv)
      {
         tmp_exch = a[fthr].x;
         a[fthr].x = a[mthr].x;
         a[mthr].x = tmp_exch;
      }

      i++;
   }
}

// 突然変異 (mutation)
void mttn(OBJ a[])
{
   int i = 1, cnt;
   int tgt;   // 突然変異する要素 (tgt = target)

   cnt = (int) MTTN * N;

   while(i <= cnt)
   {
      tgt = rand() % 71 + 1;

      if( a[tgt].dd_alv == MLTPL)
      {
         a[tgt].x -= 0.1;
         a[tgt].y -= 0.1;
      }

      i++;
   }
}

// 失った obeject の分を補う
void add(OBJ a[])
{
   int i;

   for(i = 1; i < N; i++)
```

```
   {
      if(a[i].dd_alv == SLCT)
      {
         a[i].x = drand48();
         a[i].y = drand48();
         a[i].z = f( a[i].x, a[i].y );
         a[i].dd_alv == MLTPL;
      }
   }
}

int main(int argc, char *argv[])
{
   // For Loop
   int i;

   // MPI Declare
   int myid;                                  // myid = 自分の ID
   int all_nds;                               // all_nds = all nodes (全てのノードの数)
   int msgt = 99;
   MPI_Status stat;

   int nmln;                                  // プロセッサの名前の長さ
   char prcssr_nm[MPI_MAX_PROCESSOR_NAME];    // プロセッサの名前を格納する
   double strt_tm, end_tm;                    // 処理の時間計測

   int dist_num;                              // 各々の世代の絶滅数
   long ct, gnr = 0;                          // 世代交替の回数 (gnr = generation)

   OBJ s_obj[N], r_obj[N], end_obj[LAST_N];   // s_obj    : send オブジェクト
                                              // r_obj    : receive オブジェクト
                                              // end_obj : 最終的なオブジェクト

   char *rna_recv, *rna_send;       // rna_recv：構造体受信文 rna_send：構造体送信文
   int sz;                                   // size：構造体の送信サイズ
   int pstn;                                  // pstn = position はバッファ中での現在のバイト位置
   int prtnr;                                 // 送信相手

   // MPI データ型の定義 (構造体送受信における)
   int blckln[ST_N];                          // blckln = blocklength エントリ数
   MPI_Aint disp[ST_N];                       // disp = displacement はメッセージの先頭からの位置
   MPI_Datatype STR;                          // STR = structure
   MPI_Datatype type[ST_N];                   // type = typelist は各エントリの MPI データ型

   // MPI Initializations
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &all_nds);
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
MPI_Get_processor_name(prcssr_nm, &nmln);
//fprintf(stderr, "Process %d on %s\n", myid, prcssr_nm);

// 送信構造体のタイプの宣言
type[0] = MPI_DOUBLE;
type[1] = MPI_DOUBLE;
type[2] = MPI_DOUBLE;
type[3] = MPI_INT;
blckln[0] = 1;
blckln[1] = 1;
blckln[2] = 1;
blckln[3] = 1;
disp[0] = 0;
disp[1] = sizeof(double) * 1;
disp[2] = sizeof(double) * 2;
disp[3] = sizeof(double) * 3;
MPI_Type_struct(ST_N, blckln, disp, type, &STR);
MPI_Type_commit(&STR);

// 世代数による評価のループ回数
ct = atoi(argv[1]);

// 種の初期化
srand48(time(NULL) + myid); srand(time(NULL) + myid);

// (x, y) を初期化
init_xy(s_obj);

// 時間計測開始
if(myid == 0)
   strt_tm = MPI_Wtime();

// メイン処理 (開始)
while(1)
{
   // 1： selection or multiplication
   dist_num = slt_mlt(s_obj);

   // 2： 遺伝子の交差 (crosscover)
   if(dist_num != 0)
      crsscvr(s_obj);

   // 3： 突然変異 (mutation)
   if(gnr % 10 == 10)
      mttn(s_obj);
```

```
      // 4： 評価
      if(gnr >= ct)
         break;

      // 5： 補足
      add(s_obj);

      gnr++;
   }

   // メイン処理 (終了)

   // ソート
   srt(s_obj, 1, N - 1);

   // 構造体の送受信 start
   if(myid != 0){
   // 送受信する構造体のサイズを sz に求め代入
   MPI_Pack_size(N, STR, MPI_COMM_WORLD, &sz);

   rna_send = (char *) malloc(sz);

   // 構造体のパック化
   pstn = 0;
   for(i = 0; i < N; i++)
      MPI_Pack(&s_obj[i], 1, STR, rna_send, sz, &pstn, MPI_COMM_WORLD);
   }

#if 1
   if(myid != 0)
   {
      MPI_Send(rna_send, pstn, MPI_PACKED, 0, msgt, MPI_COMM_WORLD);
   } else {
      MPI_Recv(r_obj, N, STR, 1, msgt, MPI_COMM_WORLD, &stat);
      pstn = 0;
   }
#endif

#if 0
   MPI_Sendrecv(rna_send, pstn, MPI_PACKED, 0,
                msgt, r_obj, N, STR, 1,
                msgt, MPI_COMM_WORLD, &stat);
#endif

   // myid 0 での処理 それ以外のノードはここで終了
   if(myid == 0)
   {
```

```
        // ソート
        //srt(s_obj, 1, 2 * N - 1);

        // MPI 時間計測終了
        end_tm = MPI_Wtime();

        // 結果の出力
        pri(s_obj);
        printf("\n");
        pri(r_obj);

        // 実行時間出力
        printf("time spended  %f [s]\n", end_tm - strt_tm);
    }

    MPI_Finalize();
}
```

参考文献

[1] P. パチェコ 訳 秋葉博, MPI 並列プログラミング, 培風館，2002

[2] MPI フォーラム 訳 MPI 日本語プロジェクト, MPI:メッセージ通信インターフェース標準, http://phase.hpcc.jp/phase/mpi-j/ml/, 1996

[3] MPI フォーラム 訳 MPI 日本語プロジェクト, MPI:メッセージ通信インターフェースに対する拡張機能, http://phase.hpcc.jp/p hase/mpi-j/ml/, 1997

[4] Message Possing Interface Forum, MPI-2: Extensions to the Message-Passing Interface, http://www.mpi-forum.org/docs/docs.html, , 1997

[5] The LAM/MPI Team Open Systems Lab, LAM/MPI Installation Guide http://www.lam-mpi.org/, 2003

[6] The LAM/MPI Team Open Systems Lab, LAM/MPI User's Guide http://www.lam-mpi.org/, 2003

[7] Ohio Supercomputer Center, MPI Primer Developing With LAM, http://www.osc.edu/lam.html, 1996

[8] 渡邉真也, MPI による並列プログラミングの基礎, http://mikilab.doshisha.ac.jp/dia/smpp/cluster2000/PDF/chapter02.pdf, 1997

[9] Rob Davies, An introduction to MPI and Parallel Genetic Algorithm(Cource Notes, Cardiff HPC Training & Education Centre

[10] 安居院猛, 長尾智晴, ジェネティックアルゴリズム, 昭晃堂, 1993

[11] 新山祐介, 春山征吾, OpenSSH セキュリティ管理ガイド, 秀和システム, 2001