◇◇◇◇◇◇
特　集
◇◇◇◇◇◇

PVM を使った並列プログラミング

山之上　卓[1]

1　はじめに

並列プログラミングとは，複数の仕事を複数のコンピュータ上で協調しながら同時に動作するプログラム，すなわち並列プログラム，を開発することである．

並列プログラミングは，プログラムの計算時間を短くし，大規模な計算を行なうのに必要なメモリを多く確保するのに最も有効な手段の１つであり，計算高速化の最後の切札と呼ばれている．

情報科学センターの研究教育両システムのハードウェアは，高速 EWS を，高速 Switch ネットワークで接続した分散システムであり，並列プログラムを実行させるのに適した形態になっている．

PVM (Parallel Virtual Machine) は分散システムを１台の大型並列計算機に見立て並列計算を行なうためのパッケージであり，米国テネシー大学，オークリッジ国立研究所，エモリー大学で開発された．PVM は分散システム上で並列プログラミングを行なうための，業界標準のパッケージであり，様々な分野の大規模計算で実績を持っている．

2　PVM の概要

並列プログラミングのために，様々なハードウェア，様々な方法論，プログラミング言語，ツールなどがある．

PVM が対象にするハードウェアは，複数のコンピュータをネットワークで接続したものや，MIMD 型並列マシンと呼ばれる CPU とメモリーの組みを，内部バスなどで接続したものなどである．

PVM のプログラミングは，メッセージパッシングモデルと呼ばれる方法論を用いる (図 1)．メッセージパッシングとは，複数のプログラム (オブジェクトと呼ぶ場合もある．PVM ではタスクと呼ぶ) 間で，情報交換を行なうことである．

プログラミング言語は C や FORTRAN などを利用する．メッセージパッシングは，PVM が備えているライブラリの関数やサブルーチンを呼び出すことによって行なう．

[1]情報科学センター戸畑キャンパス

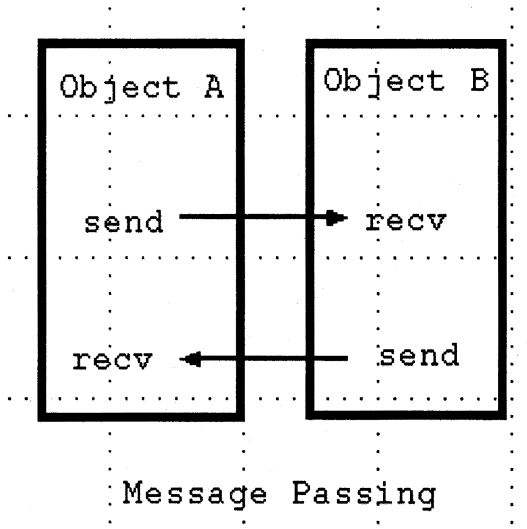

図 1: メッセージパッシング

PVM は様々な種類 (ヘテロジニアス) のコンピュータ間でメッセージパッシングを行なわせる機能を持っている．また，1種類 (ホモジニアス) のコンピュータ間では，より高速にメッセージパッシングを行なう事ができる．

PVM にはデバッグのための様々なツールが用意されている．また，並列プログラムの実行時の様子を視覚的に表示する XPVM というソフトも存在する．

3 PVM を使った数値積分プログラムの例

数値積分プログラムは並列化が最も簡単で，その効果が最も顕著なものの1つである．基本的には，

$$\int_a^b f(x)dx \simeq d\sum_{i=0}^{n-1} f(x_i)$$

$$where, d = \frac{b-a}{n}, x_i = a + d \cdot (i + 0.5)$$

のような近似式において，Σの計算を分割して，これを複数のコンピュータで同時に行なわせる．このとき，分割した各部分の計算は別の部分の値に依存しない．このため，通信は，初期値を与える時と，最後に各部分の値を合計するときに必要なだけである．

並列計算を行なう時，一部のコンピュータに偏って負荷がかかってしまうと，その一部のコンピュータの計算時間が長くなり，遊んでいるコンピュータが増えて，結果的に実行時間 (実 CPU 時間ではない) が

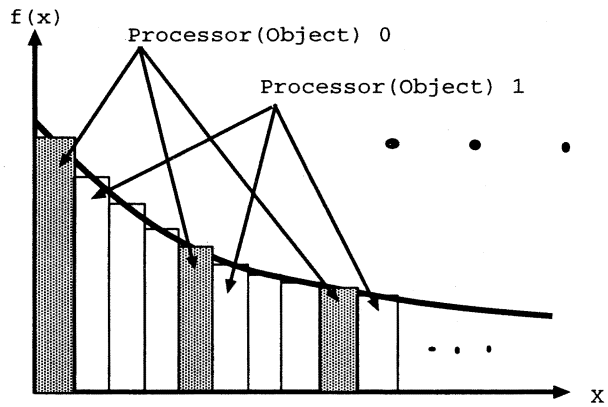

図 2: 数値積分の負荷分散

長くなってしまう．nproc 個のコンピュータで数値積分の計算をさせる場合，$k(0 \leq k \leq nproc-1)$ 番目のコンピュータが

$$d \sum_{j=0}^{\lfloor n/nproc \rfloor} f(x_i)$$

$$where, i = j \cdot nproc + k, d = \frac{b-1}{n}, x_i = a + d \cdot (i + 0.5)$$

のように nproc 個飛ばしの部分を計算させるようにすると (図 2)，負荷の偏りが小さくなることが期待できる．負荷分散が均一に行なわれた場合，nproc 個のコンピュータで計算を行なうと，従来のシーケンシャルプログラム (並列化していないプログラム) のほぼ 1/nproc の時間で計算が終了する．

図 3 (**pai_master**) と図 4 (**pai_worker**) は，数値積分によって π の近似値を求める PVM を使った並列プログラムの例である．図 3 は全体を統括しているマスタープログラムであり，図 4 は，実際の計算の大部分を行なうワーカ (スレーブと呼ぶこともある) プログラムである．図のワーカが複数のコンピュータで同時に実行され，割り当て部分の計算を行なう．なお PVM ではコンピュータ上で実際に動作しているプログラムをタスク (task) と呼ぶ．

マスタータスクは，

1. **pvm_spawn** 関数によって，ワーカタスクを nproc 個起動する．
2. **pvm_send** 関数によって，初期値やパラメータをすべてのワーカタスクに送る．
3. **pvm_recv** 関数によって，各ワーカタスクによって計算された Σ f(x) の結果を受け取り，その和を求める．

```
/*
*    calculate pai (3.191592)
*     pai_master <No.of processor> <No.of rectangle...precision>
*     by t.yamanoue, Nov.1995
*/
#include <stdio.h>
#include <sys/types.h>
#include "pvm3.h"

main(argc,argv)
int argc; char *argv[];
{
    int mytid;                  /* my task id */
    int me;                     /* my process number */
    int nproc;                  /* no. of processes */
    int n;                      /* no. of devision */
    int tids[32];               /* array of task id */
    int i,status,numt,msgtype;
    double d,s,x;

    nproc=atoi(argv[1]); n=atoi(argv[2]); printf("nproc=%d, n=%d\n",nproc,n);

    /* enroll in pvm */
    mytid = pvm_mytid();

    /* start up worker tasks */
    numt=pvm_spawn("pai_worker", (char **)0, 0, "", nproc, tids);
    if( numt < nproc) {
         printf("error..\n");
         for(i=numt ; i<nproc ; i++) printf("tid %d %d\n",i,tids[i]);
         for(i=0; i<numt; i++) pvm_kill(tids[i]);
         pvm_exit();   exit();
       }

    d=1.0/n;
    /* distribute initial data to workers */
    for(i=0;i<nproc;i++){
        pvm_initsend(PvmDataDefault);
        pvm_pkint(&nproc, 1, 1); pvm_pkint(&i, 1, 1);
        pvm_pkint(&n, 1, 1);     pvm_pkdouble(&d,1,1);
        status=pvm_send(tids[i],  0);
     }

    s=0.0;
    /* Wait for results from workers */
    msgtype = 5;
    for( i=0 ; i<nproc ; i++ ){
       pvm_recv( -1, msgtype ); pvm_upkdouble( &x, 1, 1 );  s+=x;    }
    printf("pai=%3.12lf\n",s*d);
    pvm_exit();
  }
```

図 3: プログラム 1，π の値を並列計算するプログラム (マスター)

4. 受け取った結果の総和に d を掛けて π を求める.

を行なう.

各ワーカタスクは,

1. pvm_recv 関数によって，マスタータスクから初期値やパラメータを受け取る.

2. 自分の担当部分の Σ f(x) を計算する.

3. pvm_send 関数によって，結果をマスタータスクに送る.

を行なう. 図 5 に計算の概要を示す.

```
/*
*  pai_worker
*    calculate pai (3.191592) ... worker
*     by t.yamanoue, Nov.1995
*/

#include <stdio.h>
#include <sys/types.h>
#include "pvm3.h"

main()
{
    int mytid;                  /* my task id */
    int me;                     /* my process id */
    int i;
    int nproc;                  /* no. of processes */
    int n;                      /* no. of division */

    double s,x,d;
    int msgtype,master;

    /* enroll in pvm */
    mytid = pvm_mytid();

    msgtype=0;
    pvm_recv(-1,msgtype);
    pvm_upkint(&nproc,1,1);  pvm_upkint(&me,1,1);
    pvm_upkint(&n, 1,1);     pvm_upkdouble(&d,1,1);

    s=0.0;

    for(i=me; i<n; i+=nproc) {x=(i+0.5)*d; s+=4.0/(1.0+x*x);}

    pvm_initsend(PvmDataDefault);
      pvm_pkdouble(&s,1,1);
      msgtype=5;
      master = pvm_parent();
      pvm_send(master, msgtype);
    pvm_exit();
}
```

図 4: プログラム 2，π の値を並列計算するプログラム (ワーカ)

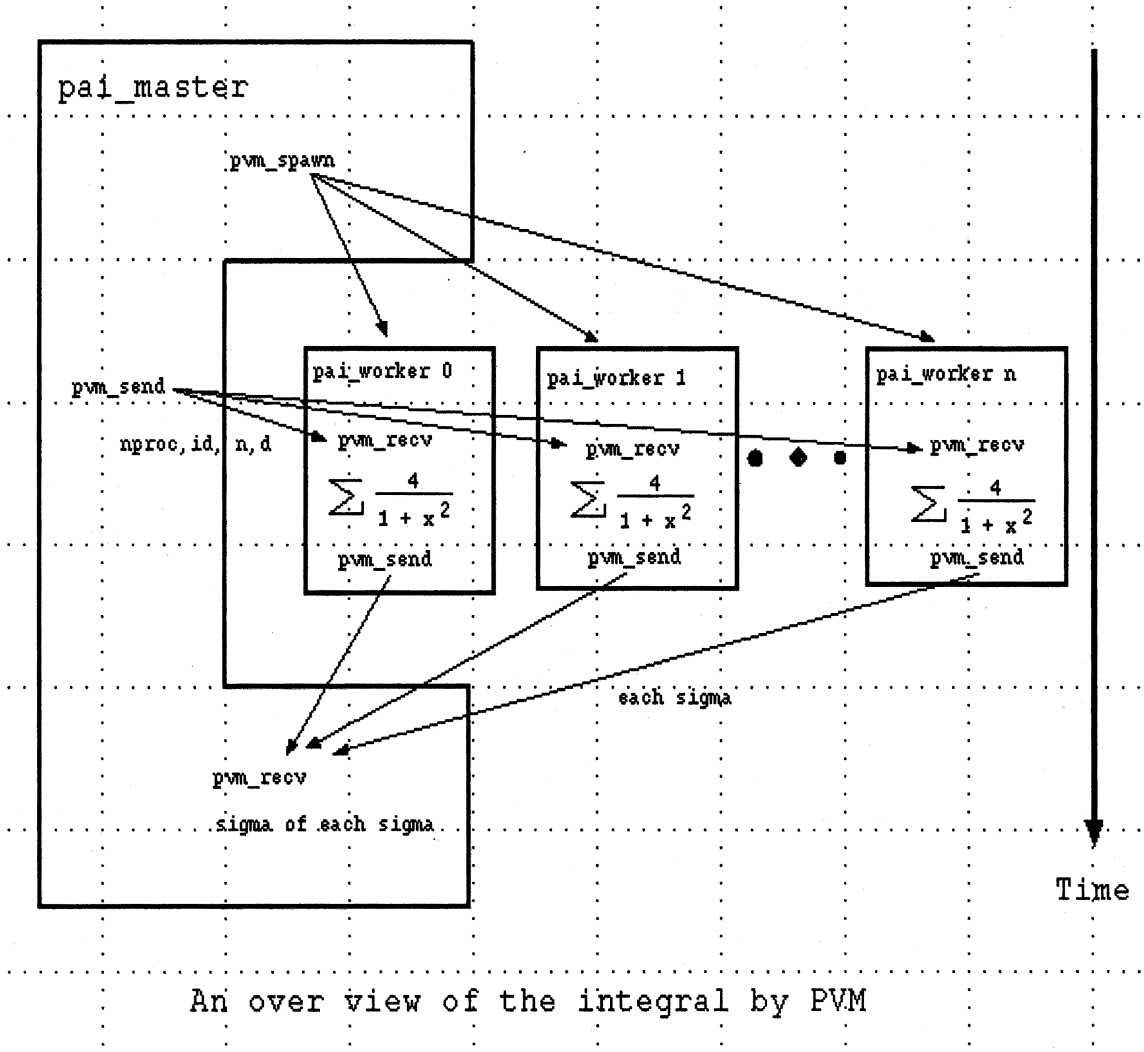

図 5: π を計算する並列プログラムの概要

4 PVM で良く使う関数 (サブルーチン)

以下は C でPVM を利用する場合に良く使う関数の一部である. C で PVM を利用する場合, プログラムの先頭に `#include "pvm3.h"` が必要である. FORTRAN の場合は `http://www.tobata.isc.kyutech.ac.jp/res-tebiki/pvm` などを参照していただきたい.

- 起動終了などを行なう関数
  - タスク識別子

    ```
    int pvm_mytid()
    ```

    そのプログラムが PVM に参加して, PVM 上での識別子を得るための関数である. PVM を使うためには必ず必要な関数である. mytid は my task id の略である.
  - 子タスクの起動

    ```
    int pvm_spawn(char * 起動するタスク (プログラム) 名 ,
                  char ** 起動するタスクの引数 (argv) のリスト,
                  int オプション (0 の場合はタスクをどこで起動しても
                      良いことを示す),
                  char * 起動場所 (ホスト名, 上のオプションが 0 の場合は
                      無視される),
                  int 起動するタスクの個数,
                  int * 起動したタスクの識別子の配列
                 )
    ```

    マスタータスクがワーカ (スレーブ) タスクを起動する時などに用いる. 帰り値は, 実際に起動されたタスクの数を示す. 3章図 3の

    ```
    numt=pvm_spawn(''pai_worker'',(char **)0,0, '''',nproc,tids)
    ```

    は, `pai_worker` を nproc 個, 任意のホストで起動し, その nproc 個のタスク識別子を配列 `tids[]` に格納することを表す.
  - 親タスクの識別子

    ```
    int pvm_parent()
    ```

    ワーカタスクでこの関数が呼び出された場合, このタスクを起動したマスタータスクの識別子を返す.

– PVM 終了

```
int pvm_exit()
```

これを実行したタスクが PVM を終了するときに用いる.

- メッセージの送信を行なう関数

– 送信開始

```
int pvm_initsend(int コード化の方法)
```

メッセージの送信手続き開始を伝える関数である. 帰り値は送信バッファの識別子である. コード化の方法には以下のものがある.

* PvmDataDefault
  XDR フォーマットで送信データをコード化する. 複数の種類のコンピュータで PVM を利用する場合に安全な方法である.
* PvmDataRaw
  コード化しないでデータを送る. 利用するコンピュータの内部フォーマットが分かっていて, それが同じ場合, 高速化できる.
* PvmDataInPlace
  送信バッファにデータを貯めずに, メモリ上のデータをそのまま送る時に使う. 高速化の効果が最も大きいが, CONVEX などのシェアードメモリマシンでは利用できないなどの制限がある.

– パック

```
int pvm_pkint(int *整数型変数や配列へのポインタ,
              int 個数(変数の場合は1)
              int ストライド(1 の場合は配列や変数の内容をすべて送る)
             )

int pvm_pkdouble(double *倍精度型変数や配列へのポインタ,
              int 個数(変数の場合は1)
              int ストライド(1 の場合は配列や変数の内容をすべて送る)
             )
```

送信するデータを包む (pack). 帰り値が負の場合はエラーを表す. ストライドは, 配列データを送信するとき, ここで指定した値をまたいでデータを包むことを示す.

- 送信

```
int pvm_send( int 受信先タスク識別子,
              int 名札
            )
```

受信先タスク識別子を持つタスクに, 名札をつけて, 包んだ (pack した) データを送信する.

3章, 図3 の

```
pvm_initsend(PvmDataDefault);
pvm_pkint(&nproc, 1, 1);
pvm_pkint(&i, 1, 1);
pvm_pkint(&n, 1, 1);
pvm_pkdouble(&d,1,1);
status=pvm_send(tids[i], 0);
```

は, タスク tids[i] に, 整数 nproc, 整数 n, 倍精度浮動小数 d を, 名札 0 を付けて送ることを表す. このとき, データはデフォルト (XDR) の変換が行なわれる.

- メッセージの受信を行なう関数

  - 同期受信

```
int pvm_recv( int 送信元タスク識別子 (-1 の場合は任意)
              int 名札
            )
```

送信元タスク識別子で指定したタスクから送られて来た, 指定した名札のついたメッセージを受信する. 帰り値は, 受信バッファの id である. このとき, 指定したメッセージが受信されるまで待つ.

  - 非同期受信

```
int pvm_nrecv( int 送信元タスク識別子 (-1 の場合は任意)
               int 名札
             )
```

送信元タスク識別子で指定したタスクから送られて来た，指定した名札のついたメッセージを受信する．帰り値は，受信バッファの id である．このとき，帰り値が 0 の場合は指定したメッセージがまだ受信されていない事を表し，受信を待たない．

– アンパック

```
int pvm_upkint(int *整数型変数や配列へのポインタ,
               int 個数 (変数の場合は 1)
               int ストライド (1 の場合は配列や変数の内容をすべて送る)
              )

int pvm_upkdouble(double *倍精度型変数や配列へのポインタ,
               int 個数 (変数の場合は 1)
               int ストライド (1 の場合は配列や変数の内容をすべて送る)
              )
```

受信したデータの包みを解く (unpack)．帰り値が負の場合はエラーを表す．ストライドは，配列にデータを受信するとき，ここで指定した値をまたいでデータを格納することを示す．

3章，図 4の

```
pvm_recv(-1, 0);
pvm_upkint(&nproc,1,1);
pvm_upkint(&me,1,1);
pvm_upkint(&n, 1,1);
pvm_upkdouble(&d,1,1);
```

は，任意のタスク (この場合はマスター) から届いた名札 0 の付いたメッセージを受けとり，包みを解いて (デコードして) nproc， me， n， d に入れることを表す．

5 PVM プログラムのコンパイルと実行

PVM を使った並列プログラム (以下 PVM プログラム) のコンパイルと実行は，情報科学センター研究システムの場合，以下のように行なう．

- 準備

1. ユーザのホームディレクトリ $HOME の下に $HOME/pvm3/ と$HOME/pvm3/bin を作成する.
2. $HOME/. rhosts に PVM で利用するコンピュータのホスト名を列挙する.

   例

   ```
   res000% cat ~/.rhosts

   res0000.res.isc.kyutech.ac.jp
   res0001.res.isc.kyutech.ac.jp
   res0002.res.isc.kyutech.ac.jp
   res0003.res.isc.kyutech.ac.jp
   res0004.res.isc.kyutech.ac.jp
   res0005.res.isc.kyutech.ac.jp
   res0006.res.isc.kyutech.ac.jp
   res0008.res.isc.kyutech.ac.jp
   res0009.res.isc.kyutech.ac.jp
   ```

3. PATH , MANPATH, PVM_ROOT, PVM_DPATH, PVM_EXPORT などの環境変数を以下のように設定する. これらの環境変数は, 研究システムの標準環境では設定済みである. 教育システムでは, 各自ホームディレクトリの.cshrc に

   ```
   source /usr/ext/pvm3.3.10/pvmsetenv
   ```

   を追加すれば良い. 以下は PVM が利用できる場合の環境変数の値である.

   ```
   PATH=...:/usr/ext/pvm3.3.10/lib: ホームディレクトリ /pvm3/bin/NEWS
   ...

   MANPATH=...:/usr/ext/pvm3.3.10/man
   ...

   PVM_ROOT=/usr/ext/pvm3.3.10
   PVM_DPATH=/usr/ext/pvm3.3.10/lib/pvmd
   PVM_EXPORT= ホームディレクトリ /pvm3
   ```

- PVM プログラムと aimk ファイルの作成

  準備の 1 で作成した pvm3 ディレクトリの下に適当な名前のサブディレクトリを作り, その下で mule 等のエディタを使って PVM のソースプログラムを作成する. 次に同じディレクトリで, PVM の Makefile である Makefile.aimk ファイルを作成する. 図 6 に 3章のプログラムをコンパイルするための Makefile.aimk ファイルの例を示す.

```
#
# Makefile.aimk for PVM example programs.
#
# Set PVM_ROOT to the path where PVM includes and libraries are installed.
# Set PVM_ARCH to your architecture type (SUN4, HP9K, RS6K, SGI, etc.)
# Set ARCHLIB to any special libs needed on PVM_ARCH (-lrpc, -lsocket, etc.)
# otherwise leave ARCHLIB blank
#
# PVM_ARCH and ARCHLIB are set for you if you use "$PVM_ROOT/lib/aimk"
# instead of "make".
#
# aimk also creates a $PVM_ARCH directory below this one and will cd to it
# before invoking make - this allows building in parallel on different arches.
#

SDIR       =       ..
BDIR       =       $(HOME)/pvm3/bin
XDIR       =       $(BDIR)/$(PVM_ARCH)

CC         =       cc
OPTIONS    =       -O3 -mips2 -non_shared
CFLAGS     =       $(OPTIONS) -I$(PVM_ROOT)/include $(ARCHCFLAGS)
LIBS       =       -lpvm3 $(ARCHLIB)
GLIBS      =       -lgpvm3

LFLAGS     =       -L$(PVM_ROOT)/lib/$(PVM_ARCH)

$(XDIR):
        - mkdir $(BDIR)
        - mkdir $(XDIR)

pai:   pai_master pai_worker
pai_master:       $(SDIR)/pai_master.c $(XDIR)
       $(CC) $(CFLAGS) -o pai_master $(SDIR)/pai_master.c \
       $(LFLAGS) $(GLIBS) $(LIBS)
       mv pai_master $(XDIR)
pai_worker:       $(SDIR)/pai_worker.c $(XDIR)
       $(CC) $(CFLAGS) -o pai_worker $(SDIR)/pai_worker.c \
       $(LFLAGS) $(GLIBS) $(LIBS)
       mv pai_worker $(XDIR)
```

図 6: π を計算する並列プログラムをコンパイルする aimk ファイルの例

- コンパイル

  Makefile.aimk ファイルが存在するディレクトリで，PVM の aimk コマンドを実行する．図6 の Makefile.aimk の例では，

  ```
  aimk pai
  ```

  を実行することによって，必要なプログラムのコンパイルが行なわれる．

- 実行

  – 直接実行させる場合

  1. pvm コマンドを実行することにより，PVM デーモンを起動する．この時，PVM モニターのプロンプト

     が表示される．PVM デーモンとは，PVM タスクの識別子の管理やメッセージパッシングの管理などを行なうプログラムである．PVM モニタで利用できるサブコマンドは help を入力することによって，表示される．

  2. PVM モニタ上で，使用するコンピュータを

     add ホスト名 ... ホスト名

     サブコマンドによって追加する．ホスト名の間は空白を入れる．ここで追加できるコンピュータは，PVM モニタを動かしているコンピュータから rsh を実行できるものでなければならない．現在使用できるホストを確かめるには，conf サブコマンドを実行する．
     なお研究システムで直接実行させる場合は，ログインしているホストしか利用できない．

  3. PVM モニタ上で，quit サブコマンドを入力することによって，PVM を抜ける．このとき PVM デーモンは動いたままである．

  4. PVM のマスタープログラムを UNIX のプロンプトで実行する．

  5. pvm コマンドを実行して PVM モニタに入り，halt コマンドを入力することによって，PVM デーモンを終了させる．

以下に 3章のプログラムの実行例を示す．

```
yamanoue% pvm
pvm> conf
1 host, 1 data format
```

```
                    HOST     DTID     ARCH   SPEED
                    yuri    40000     NEWS    1000
pvm> add kikyo renge ayame sumire
4 successful
                    HOST     DTID
                   kikyo    80000
                   renge    c0000
                   ayame   100000
                  sumire   140000
pvm> conf
5 hosts, 1 data format
                    HOST     DTID     ARCH   SPEED
                    yuri    40000     NEWS    1000
                   kikyo    80000     NEWS    1000
                   renge    c0000     NEWS    1000
                   ayame   100000     NEWS    1000
                  sumire   140000     NEWS    1000
pvm> quit

pvmd still running.
yamanoue% pai_master 4 10000000
nproc=4, n=10000000
pai=3.141592653590
yamanoue% pvm
pvmd already running.
pvm> halt
yamanoue%
```

– バッチシステムで実行させる場合

研究システムのバッチシステムを使用する場合は，バッチファイル (シェルスクリプト) を作成し，これを submit コマンドに，並列実行オプションを付けて，バッチキューに投入する．このとき，submit コマンドの並列実行オプションは，使用する CPU の数を指定する必要がある．この時 submit コマンドは以下のようにして実行する．

```
submit <並列実行オプション> <使用する CPU の数> <バッチファイル名>
```

また，バッチシステムを使用する場合は，PVM デーモンを起動したり，PVM モニターの中でホストを追加したりする必要はない．

3章のプログラムの場合，たとえば

```
#/bin/sh
rm output
pai_master 4 100000 > output
```

のようなバッチファイルを paibatch という名前で保存し，

```
chmod 755 paibatch
```

を行なって，実行可能にし，

```
submit -pa 5 paibatch
```

でバッチキューに投入する．

以下に実行例を示す．

```
yamanoue% cat paibatch
#!/bin/sh
rm output
pai_master 4 100000000 >output
yamanoue% submit -pa 5 paibatch
Job <18213> is submitted to queue <class-pa>.
yamanoue% bjobs
JOBID USER     STAT  QUEUE      FROM_HOST   EXEC_HOST   JOB_NAME   SUBMIT_TIME
18213 yamanoue RUN   class-pa   res000      res002.res. *ob paibat Nov 14 11:07
                                            res003.res.isc.kyutech.ac.jp
                                            res005.res.isc.kyutech.ac.jp
                                            res008.res.isc.kyutech.ac.jp
                                            res009.res.isc.kyutech.ac.jp
yamanoue% bjobs
No unfinished job found
yamanoue% cat output
nproc=5, n=100000000
pai=3.141592653590
yamanoue%
```

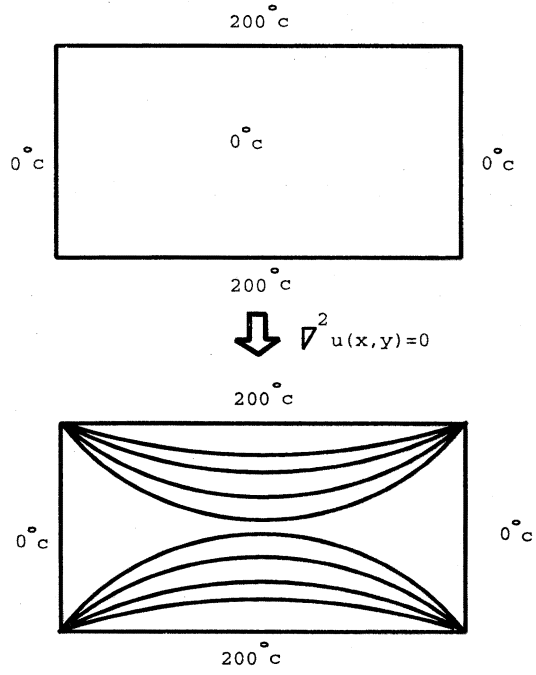

図 7: ラプラスの方程式を解くことによって温度分布を求める

6 差分法で偏微分方程式の近似解を求める並列プログラム

差分法で偏微分方程式の近似解を求めるプログラムも，並列プログラミングによって高速化可能である．

例として，ラプラスの方程式

$$\nabla^2 f(x,y) = 0$$

をガウスザイデル法で解く事によって，矩形面の温度分布の定常状態の近似解を求める場合を考える (図 7)．面の温度は，離散的なメッシュデータで近似する．計算を並列化するために，矩形面を何等分かして，それぞれの面の温度分布を，別々のコンピュータ (タスク) に計算させる．各面の計算を行なうためには，隣の面の端の計算値が必要になる．この時，コンピュータ間 (タスク間) の通信回数や通信量を少くする程，並列化の効果が大きくなる．このため，必要な通信が分割した面の左右の端の部分だけになるよう，短冊状に分割する．分割した面の境界が格子状になるよう分割した場合は，上下左右の端で通信が必要になり，通信回数が増加してしまう．

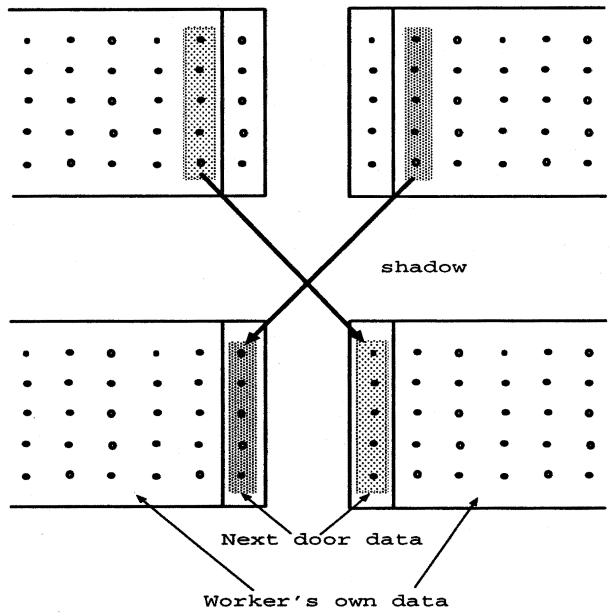

図 8: 隣接メッシュデータを得る shadow 手続き

3章の場合と同じように，プログラムは，計算全体の制御を行なうマスタープログラム (laplace_m) と，計算の大部分を行なうワーカプログラム (laplace_w) の 2 種類を使う．

マスタータスクは，

1. pvm_spawn 関数によって，ワーカタスクを複数個起動する．
2. pvm_send 関数によって，初期値やパラメータをすべてのワーカタスクに送る．
3. pvm_recv 関数によって，各ワーカタスクの残留誤差を受け取り，最大のものを選ぶ．
4. 3の結果が許容誤差以内に納まっていれば，全ワーカタスクに pvm_send 関数によって，計算終了のメッセージを送り 5 のステップに移る．そうでなければ，計算未終了のメッセージを送り，3 に戻る．
5. pvm_recv 関数によって，各ワーカタスクの計算結果を受け取り，出力する．

を行なう． 各ワーカタスクは，

1. pvm_recv 関数によって，マスタータスクから初期値やパラメータを受け取る．
2. 隣接面の端のデータを得る為， shadow という手続きによって，隣接面を担当するワーカタスクと情報交換を行なう (図 8)．

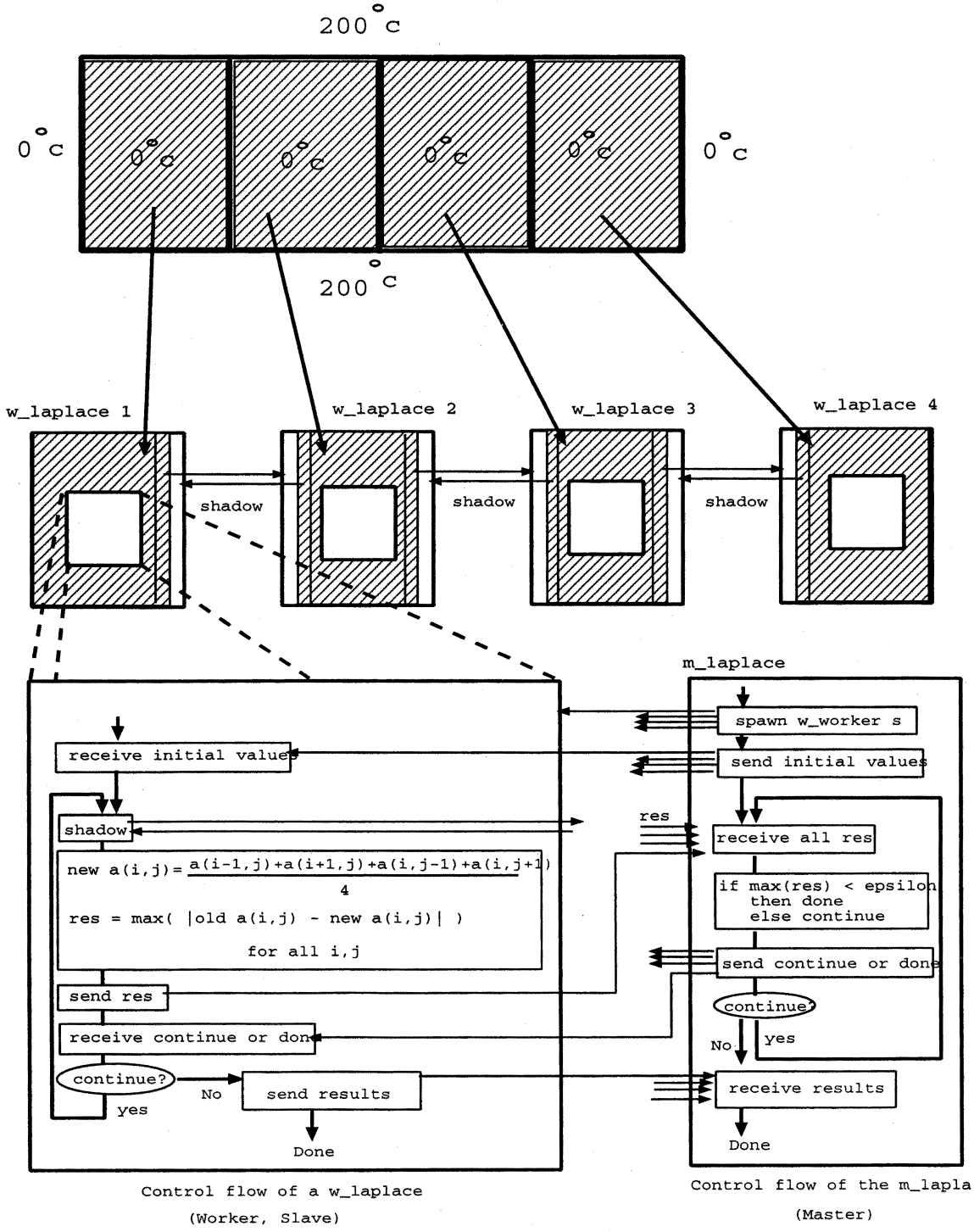

図 9: ガウスザイデル法によってラプラス方程式を解く並列プログラムの概要

3. 自分が担当する面のメッシュデータについて

$$new\ a_{i,j} = \frac{a_{i-1,j} + a_{i+1,j} + a_{i,j-1} + a_{i,j+1}}{4}$$

$$res = \max(|old\ a_{i,j} - new\ a_{i,j}|)$$

$$for\ all\ i,j$$

を計算する．ここで res は残留誤差である．

4. 残留誤差 res を `pvm_send` 関数によって，マスタータスクに送る．

5. 計算を継続するか，終了するかのメッセージを `pvm_recv` 関数によってマスタータスクから受け取る．継続であれば2 に戻り，そうでなければ次のステップに進む．

6. 計算結果 (配列の値) を `pvm_send` 関数によって，マスタータスクに送る．

を行なう．図9に計算の概要を示す．

実際のプログラム (FORTRAN) は

URL `http://www.tobaba.isc.kyutech.ac.jp/res-tebiki/pvm/laplace/laplace.html`

に掲載しているので参照されたい．

7 おわりに

PVM を使った数値積分や差分法の計算を行う並列プログラムの解説を行った．この他に，連立一次方程式の解を求める場合などでも並列プログラミングは計算の高速化に効果がある．今回取り上げた数値計算以外でも，データベース検索の高速化や人工知能プログラミングなどでも並列計算が行われている．

参考文献

[1] S. ラグスデイル著，吉川正孝訳，"並列処理とオブジェクト指向プログラミング"マグロウヒル出版株式会社 1993 年，定価 2000 円

[2] 星野 力編著，"PAX コンピュータ - 高並列処理と科学計算 -"株式会社オーム社 1985 年