C 言語による MPI プログラミング入門

片桐　孝洋

東京大学情報基盤センター

１．はじめに

本稿は、東京大学工学部・工学系研究科の共通科目として平成 19 年度から夏・冬学期に開講している「スパコンプログラミング（１）および（Ｉ）」の講義資料を基にして、C 言語により MPI のプログラムを行う並列プログラム初心者に対する参考資料として構成・執筆しました。基本的な概念、用語、使い方の説明を重点的に行っております。また、MPI を用いた並列プログラミングのコツをつかむための２、３の実例を載せてあります。スパコンを用いて並列化をおこなうユーザの参考資料になることを期待しています。

２．並列プログラミングとは

並列プログラミングは、高速化手段の一つです。この高速化は、処理対象を並列化し、並列処理を記述できる計算機言語でプログラミングをし、さらにその並列プログラムを複数の要素計算機 (Processing Element, PE) で構成された並列計算機上で実行することで達成されます。簡単にいうと、1 つの計算機で 1 時間かかる処理を、n 台の計算機を用いることで 1/n 時間で処理をすることを狙う技術です。

ところが実際は、単純に 1/n 時間となりません。これは処理対象の逐次部分の顕著化、利用する並列計算用ソフトウエア(たとえば、本稿で学習する通信ライブラリ MPI)のオーバヘッドなど、さまざまな並列化を阻害する要因が生じてしまうからです。なるべく 1/n 時間となるように、アルゴリズムや実装上の工夫をすることが効率の良い並列プログラム(並列アルゴリズム)を開発するために重要となります。効率の良い並列プログラムを開発するためには、処理すべき対象の性質、利用する計算機言語の性質、および実行する並列計算機環境(ソフトウエア、ハードウエアの両面)の知識が必要になります。

効率の良い並列プログラム開発のための技術は興味深いものですが、ここでは紙面の都合上、あまり述べることができません。そこで並列プログラミングの初心者が、最低限の並列プログラミングができるようになる程度の知識のみに限定し解説します。

２．１　Flynn の分類

並列計算機の分類として、スタンフォード大学の M.Flynn 教授が与えた、**Flynn の分類**という分類が知られています。Flynn の分類では、以下のように、**命令流(Instruction Stream)**と**データ流(Data Stream)**が、**単一(Single)**か**複数(Multiple)**かによって分類分けをしています。

- SISD (Single Instruction stream Single Data stream) : 逐次計算機
- SIMD (Single Instruction stream Multiple Data stream):ベクトル PE
- MISD (Multiple Instruction stream Single Data stream):例なし
- MIMD (Multiple Instruction stream Multiple Data stream):マルチ PE、PC クラスタ、データフローマシン

本稿で紹介する MPI による並列プログラムの特性における分類は、SIMD もしくは MIMD の分

類に当たります[1]。しかし MIMD の概念を用いた並列プログラミングは、複雑なため初心者に向きません。

本稿では以降、SIMD の分類(プログラミング・モデル)で並列プログラミングをすることを前提とします。図 1 に、4 台の計算機を使う場合の SIMD の概念図を載せます。

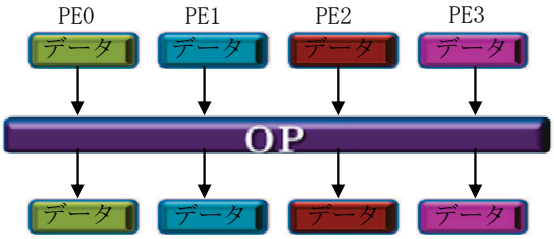

図 1 SIMD の概念

２．２　メモリ構成による分類

並列計算機アーキテクチャを、メモリ構成の違いにより分類する分類法が知られています。詳しくは、本特集号の「オープンスパコンの OS とアーキテクチャの基礎」(松葉浩也著) をご覧ください。

共有メモリ型並列計算機は、メモリを複数の PE が共有している並列計算機です。各 PE から共通のメモリ空間が参照できるため、プログラミングが容易となります。しかしメモリにアクセスするネットワークが複雑化する理由から、PE 数が多い並列計算機は構成できません(**スケーラビリティが低い**)。

分散メモリ型並列計算機は、各 PE にローカルなメモリを所有します。ローカルなメモリを所有するため、メモリ空間は PE 毎に分割され、それゆえ、他の PE が所有するデータをすぐには参照できません。したがって、プログラミングが困難となります。しかし PE 数が多い並列計算機が構成可能です(**スケーラビリティが高い**)。

最後に共有分散メモリ型並列計算機ですが、各 PE が共有分散メモリを所有するものです。共有分散メモリは、物理的には分散メモリなのですが、ハードウエアで論理的に共有メモリ化されているメモリといえます。そのためユーザの観点では、メモリ空間は単一のメモリ空間だとみなせます。それゆえプログラミングが容易となります。分散共有メモリ型並列計算機は、共有メモリ型並列計算機のプログラミングの容易さという長所、分散メモリ型並列計算機の高いスケーラビリティという長所を両方もつ並列計算機といえます。しかし、単一メモリを実現するハードウエア技術(メモリ内容の一貫性保証)が困難であること、またハードウエア実装においてハードウエア量の増加や金銭コストが増大するという問題が生じます[2]。

さて本稿で学習する MPI は、分散メモリ型並列計算機を対象に開発された通信ライブラリです。分散メモリ型並列計算機を対象としていることから、分散メモリ型並列計算機のみでしか

---

[1] 厳密には、MPI による並列プログラムの挙動は MIMD といえます。

[2] ハードウエア量増加の問題を解決するため、ソフトウエアで DSM を実現する研究がなされています(ソフトウエア DSM)。しかしソフトウエア DSM では、処理の遅さによる性能低下が常に問題となります。

実行できないと思うかもしれません。ところがMPIで記述された並列プログラムは、共有メモリ型並列計算機でも共有分散メモリ型並列計算機でも実行できるのです。この理由は、分散メモリ空間は単一メモリ空間を包含するので、一切変更をしなくても論理的に動作可能であるからです。さらにMPIで記述された並列プログラムは、データの参照が局所化されている理由から、共有分散メモリ型並列計算機用のプログラムより高速である事例が報告されています。

以上のことから、現在高速な並列プログラムを開発したい場合には、プログラミングは少々しにくいけれどもMPIを用いて開発するのがベストであるといえます。

３．MPIによる並列プログラミング入門

３．１　MPIとは

MPI（Message Passing Interface）とは、分散メモリ型並列処理の基本であるメッセージパッシングのライブラリの規格です。並列処理をするために初期は、PVM（Parallel Virtual Machine)、各社の並列計算機用の通信ライブラリなど、多数の通信ライブラリがありました。ところがここ十年あまりで、MPI が世界標準の通信ライブラリとなってしまいました。ものの本によると[1]、MPIが話題にのぼるようになってからPVMがMPIに置き換わるまで、1年ぐらいであったとされます。この理由は、MPI の設計方針のよさや、Linux とその上で動く mpich などのおかげで、広くユーザに支持されたことが大きいとのことです。現在、ほぼすべての並列計算機上で MPIライブラリが動きます。

もう1つの世の中の変化は、クラスタコンピューティング、分散メモリ型並列処理の広まりがあったとされます。共有メモリ型の並列処理には、物理上の限界（メモリバンド幅がとれない）があり、やはり大規模計算を行うには分散メモリ型の並列処理に最終的には到達するわけです。

一方でクラスタシステム上の大規模アプリケーションは広く利用されると考えられています。なぜならハードウエアの構築は（理論上は）際限なくできますし、プログラムの書き方によっては高い性能を引き出せるからです。

したがってアプリケーション開発者は、従来型のプログラミング技術（逐次型、メモリ階層型、もしくはベクトル型のプログラミング技術）に加えて、メッセージパッシングという新しい要素技術を勉強しなくてはならなくなっています。この流れは、時代の要請であるといえます。

３．２　MPIの背景

MPIの背景について、簡単に説明しておきます。現在，普通はMPIといえばMPI-1を指し、ここでの説明もMPI-1に基づいています。この拡張版にMPI-2(1997年7月）があります。MPI-2の規格は、一部MPI-1.2に実装されています。

MPI-2では、動的プロセス生成、単方向通信、外部インタフェース、拡張集団通信、並列I/O, C++およびFortran90インタフェースなど、非常に広範囲の拡張が行われました。特にMPI-1とMPI-2の大きな違いは、動的にプロセスが生成／消滅ができるので、そのような並列処理に向いているということです。例えば、並列の探索処理がその例です。

MPI-1.2については、それに準拠する実装がmpich 1.2などすでにいくつかあるのですが、MPI-2への動きはまだ十分でないといえます。ただ国産メーカのスーパコンピュータ（日立、

富士通、NEC など）には、MPI-2 が実装され利用できる状況にあります。

３．３　MPI が提供する主なインタフェース

MPI の全インタフェース（関数）について説明することは大変ですし、じつはあまり利用しない関数もあることから無駄といえます。したがってここでは、よく利用すると思われる、以下の関数についてのみ説明することとします。

- システム関数
  - MPI_Init 関数
  - MPI_Comm_rank 関数
  - MPI_Comm_size 関数
  - MPI_Finalize 関数
- 1 対 1 通信関数
  - ブロッキング型
    - MPI_Recv 関数
    - MPI_Send 関数
  - ノンブロッキング型
    - MPI_Isend 関数
    - MPI_Wait 関数
- 集団通信関数
  - MPI_Reduce 関数
  - MPI_Allreduce 関数
  - MPI_Barrier 関数
- 1 対全通信関数
  - MPI_Bcast 関数
- 時間計測関数
  - MPI_Wtime 関数
- 通信対象分割関数
  - MPI_Comm_split 関数

これらの関数の利用法を、次章に示すプログラム例を基に説明していきます。

４．プログラム例

４．１　初期設定など

それでは、具体的に MPI のプログラム例を示すことで、MPI プログラムに慣れていきましょう。

以下の例は、円周率を求めるプログラムを MPI で並列化したものです。

```
< 1>  #include <stdio.h>
< 2>  #include <math.h>
< 3>  #include <mpi.h>
< 4>  void main(int argc, char* argv[]) {
```

```
 5>      double PI25DT = 3.14159265358979238462643;
 6>      double mypi, pi, h, sum, x, f, a;
 7>      int    n, myid, numprocs, i, rc;
 8>      int    ierr;
 9>      ierr = MPI_Init(&argc, &argv);
10>      ierr = MPI_Comm_rank(MPI_COMM_WORLD, &myid);
11>      ierr = MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
12>      if ( myid == 0 ) {
13>         printf("Enter the number of intervals ¥n");
14>         scanf("%d",&n);
15>         printf("n=%d ¥n",n);
16>      }
17>      ierr = MPI_Bcast(&n, 1, MPI_INT,0,MPI_COMM_WORLD);
18>      h = 1.0 / n;
19>      sum  = 0.0;
20>      for (i = myid+1; i<=n; i+= numprocs) {
21>         x = h * (i - 0.5);
22>         sum = sum + 4.0 / (1.0 + x*x);
23>      }
24>      mypi = h * sum;
25>      MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE,
26>                 MPI_SUM, 0, MPI_COMM_WORLD);
27>      if (myid == 0) {
28>        printf("  pi is approximately: %18.16lf Error is: %18.16lf ¥n",
29>                pi, fabs(pi-PI25DT));
30>      }
31>      rc = MPI_Finalize();
32>   }
```

なおアルゴリズムの詳細はここでは説明しませんが、興味のある方はプログラムを追跡して調べてみることを勧めます。

まずはじめにMPIでは、上記のプログラムがすべてのPE上で実行されます。このような実行形態を、**SPMD (Single Program Multiple Data)**とよびます。

図2に、このプログラムの処理形態を載せます。図2の処理形態では、概述のSIMDの処理形態であることがわかります。

以降、このプログラムについて簡単な説明をします。

まずプログラム中で、<9>行にある関数でMPIの利用を初期化します。また<31>行の関数で、MPIを終了します。したがって全てのMPIプログラムは、MPI_Init関数で始まり、MPI_Finalize関数で終了します。

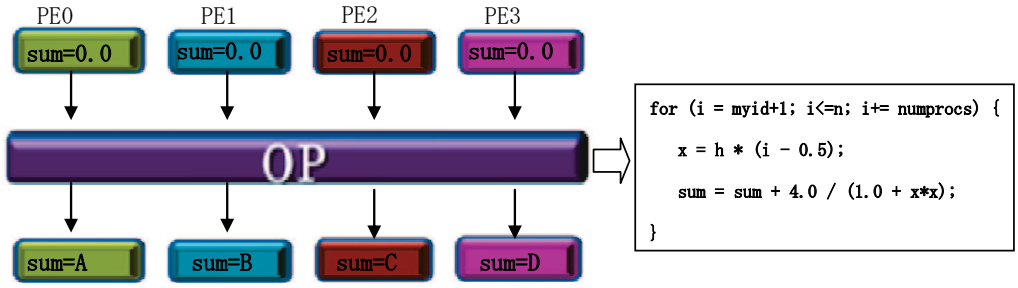

図 2　円周率プログラムの処理形態

<10>行の MPI_Comm_rank 関数では、各 PE に個別に割り付けられた認識番号を得ます。このプログラムでは、その番号は整数変数 myid に保存されます。この認識番号は 0 から始まり PE 台数より一つ少ない数で終ります。この認識番号のことを、MPI では**ランク**とよんでいます。

また MPI_COMM_WORLD は、**コミュニケータ**とよばれる MPI の制御変数です。MPI_COMM_WORLD に通信すべき対象の PE グループが代入されています。

<11>行の MPI_Comm_size 関数では、変数 numprocs に利用する全ての PE 数が代入されます。この値は、各 PE で同一の値となります。

とりあえずここでは、以上の関数の理解のみで終ることにします。

４．２　PE 間での総和演算

多くの並列処理プログラムでは、PE 間で所有している異なるデータに対して、加算演算（総和演算）をしたいことがあります。この PE 間での総和演算は、通信を必要としますが、通信方式が全体の性能に大きく影響します。

まずここでは、素朴な通信方式である各 PE が隣接 PE からのデータを受信した後、加算してその結果を隣接 PE に転送する実装法を示します。

```
<1 >  MPI_Status istatus;
<2 >  int  ierr;
<3 >  int  myid, nprocs;
<4 >  double  dsendbuf, drecvbuf;
<5 >  ierr = MPI_Init(&argc, &argv);
<6 >  ierr = MPI_Comm_rank( MPI_COMM_WORLD, &myid );
<7 >  ierr = MPI_Comm_size( MPI_COMM_WORLD, &nprocs );
<8 >  /* === 各 PE における総和演算したい値。ここでは myid とする。*/
<9 >  dsendbuf = myid;
<10>  printf ("myid:%d, dsendbuf=%4.2lf ¥n", myid, dsendbuf);
<11>  drecvbuf = 0.0;
<12>  if (myid != 0) {
<13>     ierr = MPI_Recv(&drecvbuf, 1, MPI_DOUBLE, myid-1, 0, MPI_COMM_WORLD, &istatus);
<14>  }
```

```
<15>  dsendbuf = dsendbuf + drecvbuf;
<16>  if (myid != nprocs-1) {
<17>    ierr = MPI_Send(&dsendbuf, 1, MPI_DOUBLE, myid+1, 0, MPI_COMM_WORLD);
<18>  }
<19>  if (myid == nprocs-1) printf ("Total = %4.2lf ¥n", dsendbuf);
<20>  ierr = MPI_Finalize();
```

このプログラムでは、行<12>-<14>で各 PE で計算済の部分和との加算を行うために隣接 PE である myid-1 番の PE からの受信を待ちます。MPI 関数 MPI_Recv の各引数の意味は以下の通りです。

```
int MPI_Recv(recvbuf, icount, idatatype, isource, itag, icomm, istatus);
```

- Recvbuf ： 受信領域の先頭番地を指定する。
- icount ： 整数型。受信領域のデータ要素数を指定する。
- idatatype ： 整数型。受信領域のデータの型を指定する。よく用いられるデータ型としては、以下である。
  - MPI_INT (整数型)
  - MPI_REAL (実数型)
  - MPI_DOUBLE (倍精度実数型)
- isource ： 整数型。受信したいメッセージを送信する PE のランクを指定する。
  - 任意の PE から受信したいときは、MPI_ANY_SOURCE を指定する。
- itag ： 整数型。受信したいメッセージに付いているタグの値を指定する。
  - 任意のタグ値のメッセージを受信したいときは、MPI_ANY_TAG を指定する。
- icomm ： 整数型。PE 集団を認識する番号であるコミュニケータを指定する。通常では MPI_COMM_WORLD を指定すればよい。
- istatus ： MPI_Status 型の配列の先頭番地を指定する。受信状況に関する情報が入る。
  - 例題のように MPI_Status 型の配列を宣言して、指定する。
  - 受信したメッセージの送信元のランクが整数フィールド istatus.MPI_SOURCE に、タグが整数フィールド istatus.MPI_TAG に代入される。
- 戻り値 ： 整数型。エラーコードが入る。

一方、行<16>-<18>で各 PE で計算済の部分和を隣接 PE である myid+1 番の PE に送信します。MPI 関数 MPI_Send の各引数の意味は以下の通りです。

```
int MPI_Send(sendbuf, icount, datatype, idest, itag, icomm);
```

- sendbuf ： 送信領域の先頭番地を指定する。
- icount ： 整数型。送信領域のデータ要素数を指定する。
- datatype ： 整数型。送信領域のデータの型を指定する。

- idest ： 整数型。送信したいPEの icomm内でのランクを指定する。
- itag ： 整数型。受信したいメッセージに付けられたタグの値を指定する。
- icomm ： 整数型。PE集団を認識する番号であるコミュニケータを指定する。
  - 通常ではMPI_COMM_WORLDを指定すればよい。
- 戻り値 ： 整数型。エラーコードが入る。

この例題では、最終的にPEnprocs-1番にのみ演算の結果が得られます。また結果が得られるまで、通信が（PE台数-1）回であるnprocs-1回必要となる点に注意してください。

高速プログラミングためのヒント：

さて次に、この素朴な通信方式を改良してみましょう。この改良された実装法は、データ構造とアルゴリズムを学ぶ上できわめて重要な概念である、二分木構造を利用するものです。先程の例題の行<12>-<18>を以下のプログラムに書き換えることで、二分木形態を用いた通信を実現できます。ここで説明を簡単にするためにPE台数は 2の冪数をとるものとします。

```
<12'>   inlogp = 3; /* PE台数 nprocs の 2 を底とする対数 log_2(nprocs) */
<13'>   k = 1;
<14'>   for(i=0; i<=inlogp-1; i++) {
<15'>     if ( (myid&k)  == k ) {
<16'>       ierr = MPI_Recv(&drecvbuf, 1, MPI_DOUBLE, myid-k, i, MPI_COMM_WORLD,
                        &istatus);
<17'>       dsendbuf = dsendbuf + drecvbuf;
<18'>       k *= 2;
<19'>     } else {
<20'>       ierr = MPI_Send(&dsendbuf, 1, MPI_DOUBLE, myid+k, i, MPI_COMM_WORLD);
<21'>       break;
<22'>     }
<23'>   }
```

8台のPEを用いる場合を考えましょう。この場合の通信手順を図 3に示します。

図 3では、各PEのランクを二進数で考えるとi反復時において、下位ビットから数えてi番目のビットが0のPEがデータを送信、ビットが1のPEがデータを受信することがわかります。この受信-送信の関係を整理して書き直すと、二分木の形状をなすことが見てとれます。このことから、この通信方式を、**二分木通信方式**とよびます。また、各種ゲームにおけるトーナメント形式(勝ち抜き戦)の対戦表にも類似していることから、**トーナメント方式の通信方式**ともよばれます。

次に、この二分木通信方式の通信回数を考えましょう。トーナメント方式であるので、試合数に相当する合計の通信回数はnprocs-1 回ですが、各log_2(nprocs)段のステージで同時に試合（通信）を行います。この同時通信について、通信衝突などの時間増加の要因がないと過程できれば、log_2(nprocs)回の通信回数で、結果を得ることができます。

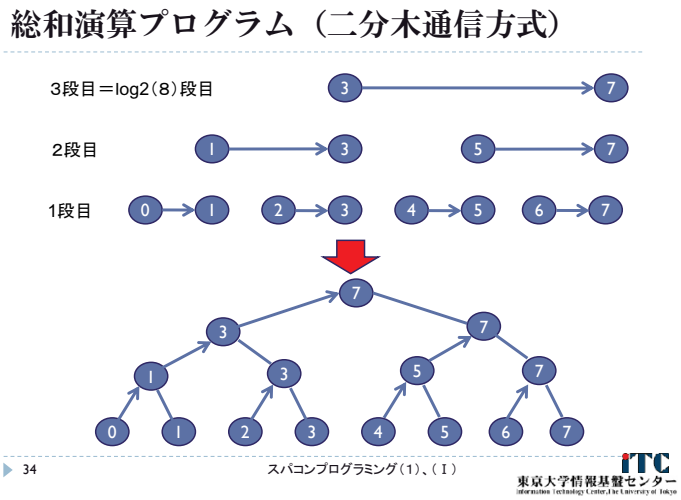

図 3　二分木通信方式の通信手順

この通信回数に関する改良は非常に有効であることに注意してください。たとえば 1024 台の PE を用いる場合、改良前の方法では 1023 回通信が必要なのに対して、二分木通信方式を用いた通信方式では log_2(1024)=10 なので、たった 10 回で済みます[3]。

４．３　リダクション演算

多くの並列プログラムでは、先程の計算例で示したように、通信と演算が必要な PE 間での演算処理が必要となります。このような演算を**縮約演算**または**リダクション演算**(**reduction operation**)、もしくは**集団通信演算**(**collective communication operation**) とよびます。

代表的なリダクション演算は、総和演算の他に最大値や最小値を求める演算などがあげられます。MPI ではこれらのリダクション演算を行う関数を、演算結果の持ち方の違いで以下の二つに分けています。

まず演算結果をある一つの PE に所有させる関数として、MPI_Reduce 関数があります。

```
int MPI_Reduce(sendbuf, recvbuf, icount, idatatype, iop, iroot, icomm);
```

- sendbuf ： 送信領域の先頭番地を指定する。
- recvbuf ： 受信領域の先頭番地を指定する。
  - iroot で指定した PE のみで書き込みがなされる。
  - 送信領域と受信領域は、同一であってはならない。すなわち、異なる配列を確保しなくてはならない。

[3] ただし、二分木通信方式が必ずしも高速でない点に注意してください。なぜなら、ネットワークの形状や性能に大きく依存するからです。たとえば PE 台数が少ない場合、1 つの PE にデータを収集する実装法が高速であることもあります。

- icount ： 整数型。送信領域のデータ要素数を指定する。
- idatatype ： 整数型。送信領域のデータの型を指定する。
  - 特に最小値や最大値と位置を返す演算を指定する場合は、以下を指定する。
    - MPI_2INT（整数型）
    - MPI_2FLOAT（単精度型）
    - MPI_2DOUBLE（倍精度型）
- iop ： 整数型。演算の種類を指定する。たとえば、以下を指定する。
  - MPI_SUM（総和）
  - MPI_PROD（積）
  - MPI_MAX（最大）
  - MPI_PROD（最小）
  - MPI_MAXLOC（最大と位置）
  - MPI_MINLOC（最小と位置）
- iroot ： 整数型。結果を受け取る PE の icomm 内でのランクを指定する。
  - 全ての icomm 内の PE で同じ値を指定する必要がある。
- icomm ： 整数型。PE 集団を認識する番号であるコミュニケータを指定する。
- 戻り値 ： 整数型。エラーコードが入る。

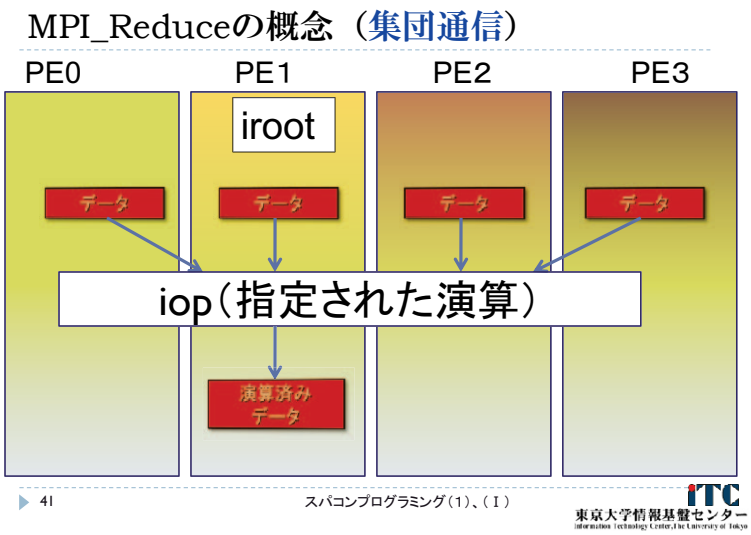

図 4　MPI_Reduce の概念

次に、演算結果を全ての PE に所有させる関数として MPI_Allreduce 関数があります。

```
int MPI_Allreduce(sendbuf, recvbuf, icount, idatatype, iop, icomm);
```

- sendbuf ： 送信領域の先頭番地を指定する。
- recvbuf ： 受信領域の先頭番地を指定する。
  - iroot で指定した PE のみで書き込みがなされる。

- なお、送信領域と受信領域は、同一であってはならない。すなわち、異なる配列を確保しなくてはならない。

- icount ： 整数型。送信領域のデータ要素数を指定する。
- idatatype ： 整数型。送信領域のデータの型を指定する。
  - 特に最小値や最大値と位置を返す演算を指定する場合は、
    - MPI_2INT（整数型）
    - MPI_2FLOAT（単精度型）
    - MPI_2DOUBLE（倍精度型）

  で指定する。
- iop ： 整数型。演算の種類を指定する。
  - MPI_SUM（総和）
  - MPI_PROD（積）
  - MPI_MAX（最大）
  - MPI_PROD（最小）
  - MPI_MAXLOC（最大と位置）
  - MPI_MINLOC（最小と位置）

  など。
- icomm ： 整数型。PE 集団を認識する番号であるコミュニケータを指定する。
- 戻り値 ： 整数型。エラーコードが入る。

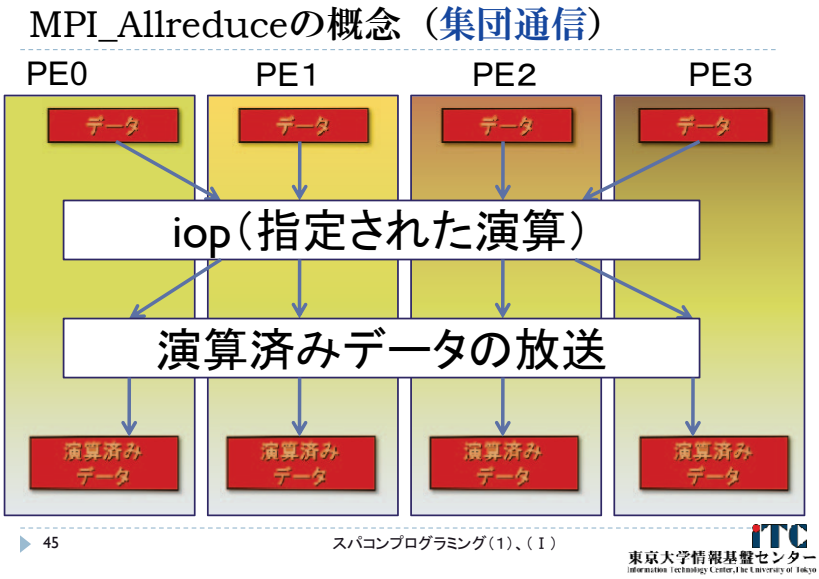

図 5　MPI_Allreduce の概念

高速プログラミングためのヒント：

MPI_Allreduce 関数は MPI_Reduce 関数に比べ、演算結果を全 PE に所有させるための放送処理が必要なことから、時間がかかることに注意してください。またこれらのリダクション演算

は、一般的に通信と演算を必要とするので単純な放送処理よりも時間がかかることを利用者は認識すべきです。すなわちリダクション演算を多用する実装方式は、性能の観点から好ましくないということを注意すべきです。

４．４　放送処理

放送処理を行う MPI 関数 MPI_Bcast について示します。

```
int MPI_Bcast(buf, icount, idatatype, iroot, icomm);
```

- buf ： 送受信領域の先頭番地を指定する。
- icount ： 整数型。送信領域のデータ要素数を指定する。
- idatatype ： 整数型。送信領域のデータの型を指定する。
  - よく用いられるデータ型としては、
    - MPI_INT（整数型）
    - MPI_REAL（実数型）
    - MPI_DOUBLE（倍精度実数型）

  などがある。
- iroot ： 整数型。結果を受け取る PE の icomm 内でのランクを指定する。
  - 全ての icomm 内の PE で同じ値を指定する必要がある。
- icomm ： 整数型。PE 集団を認識する番号であるコミュニケータを指定する。
- 戻り値 ： 整数型。エラーコードが入る。

MPI_Bcast 関数では、iroot で指定した PE が所有するデータを、すべての PE に変数 buf を通して放送することができます。

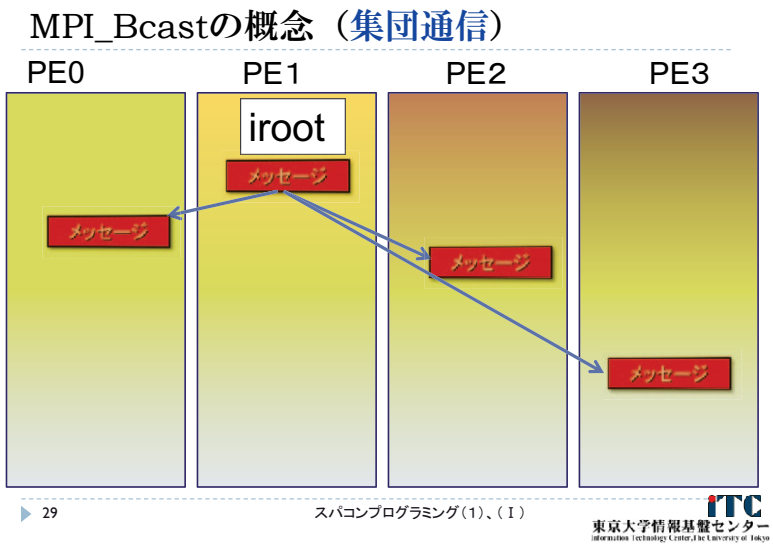

図 6　MPI_Bcast の概念

４．５　ノンブロッキング通信

いままでは、通信が**ブロッキング**である前提で説明をしてきました。通信がブロッキングとは、１対１通信の場合、MPI_Send が発行された PE は、どこかで MPI_Recv が発行されるまで処理が停止するということです。MPI では、ブロッキングではない通信（**ノンブロッキング**通信）についても、仕様を定めています。以降、ブロッキングとノンブロッキングの説明をします。

- **ブロッキング**
  - 送信／受信側のバッファ領域にメッセージが格納され、受信／送信側のバッファ領域が自由にアクセス・上書きできるまで、呼び出しが戻らない処理。
  - バッファ領域上のデータの一貫性を保障する。
- **ノンブロッキング**
  - 送信／受信側のバッファ領域のデータを保障せずすぐに呼び出しが戻る処理。
  - バッファ領域上のデータの一貫性を保障しない。
  - データ一貫性の保証はユーザの責任とする。

また、**ブロッキング**、**ノンブロッキング**が定義されると、各関数の挙動が定義できます。これは、ローカルとノンローカルという概念です。以下にそれを示します。

- **ローカル**
  - 手続きの完了が、それを実行しているプロセスのみに依存する。
  - ほかのユーザプロセスとの通信を必要としない処理。
- **ノンローカル**
  - 操作を完了するために、別のプロセスでの何らかの MPI 手続きの実行が必要かもしれない。
  - 別のユーザプロセスとの通信を必要とするかもしれない処理。

ノンブロッキングな通信を実現するには、通信するデータのコピーに関するとり扱い（バッファリング）が必要になります。MPI では、以下の４つのバッファリング方式が規定されています。

1. **標準通信モード**
   - デフォルトの方式
   - ノンローカル
   - 送出／受入メッセージのバッファリングは MPI に任せる。
     - バッファリングされるとき：　受信起動前に送信を完了できる
     - バッファリングされないとき：　送信が終了するまで待機する
2. **バッファ通信モード**
   - ローカル
   - 必ずバッファリングする。バッファ領域がないときはエラーとなる。
3. **同期通信モード**
   - ノンローカル

- バッファ領域が再利用でき、かつ、対応する受信／送信が開始されるまで待つ。

4. **レディ通信モード**
   - 処理自体はローカル
   - 対応する受信／送信が既に発行されている場合のみ実行できる。それ以外はエラーとなる。
     - ハンドシェーク処理を無くせるため、高い性能を発揮する。

いままで使ってきたMPI_Send関数は、以下のように説明できます。

- MPI_Send関数
  - ブロッキング
  - 標準通信モード（ノンローカル）
  - バッファ領域が安全な状態になるまで戻らない
    - バッファ領域がとれる場合： メッセージがバッファリングされる。対応する受信が起動する前に、送信を完了できる。
    - バッファ領域がとれない場合： 対応する受信が発行されて、かつ、メッセージが受信側に完全にコピーされるまで、送信処理を完了できない。

ノンブロッキングな送信を実現する、MPI_Isend 関数のインタフェースについて、以下に示します。

```
int MPI_Isend(sendbuf,  icount,  datatype, idest,  itag,  icomm,  irequest);
```

- sendbuf ： 送信領域の先頭番地を指定する。
- icount ： 整数型。送信領域のデータ要素数を指定する。
- datatype ： 整数型。送信領域のデータの型を指定する。
- idest ： 整数型。送信したいPEのicomm 内でのランクを指定する。
- itag ： 整数型。受信したいメッセージに付けられたタグの値を指定する。
- icomm ： 整数型。PE集団を認識する番号であるコミュニケータを指定する。
  - 通常ではMPI_COMM_WORLD を指定すればよい。
- irequest： MPI_Request型（整数型の配列)。送信を要求したメッセージにつけられた識別子が戻る。
- 戻り値 ： 整数型。エラーコードが入る。

MPI_Isend関数は、以下のように説明ができます。

- MPI_Isend関数
  - ノンブロッキング
  - 標準通信モード（ノンローカル）
    - 通信バッファ領域の状態にかかわらず戻る
      - バッファ領域がとれる場合は、メッセージがバッファリングされ、対応する

受信が起動する前に、送信処理が完了できる。

- バッファ領域がとれない場合は、対応する受信が発行され、メッセージが受信側に完全にコピーされるまで、送信処理が完了できない。
- 以上はシステムの挙動で、MPI_Wait 関数（後述）が呼ばれた場合の振舞いと理解すべき。

以下のように解釈してください。MPI_Send 関数は、この関数中に MPI_Wait 関数が入っている関数と見なせます。MPI_Isend 関数は、この関数中に MPI_Wait 関数が入っていない、かつ、すぐにユーザプログラム戻る関数であると見なせます。

MPI_Isend では、送信領域を書き換えても安全かどうかはチェックしません。そこで、送信領域を書き換えてもよい状況になるまで待つ関数が用意されています。これが、以下に説明する MPI_Wait 関数です。

```
int MPI_Wait(irequest,  istatus);
```

- irequest： MPI_Request 型（整数型配列)。送信を要求したメッセージにつけられた識別子。
- istatus ： MPI_Status 型（整数型配列)。受信状況に関する情報が入る。
  - 要素数が MPI_STATUS_SIZE の整数配列を宣言して指定する。
  - 受信したメッセージの送信元のランクが istatus[MPI_SOURCE] 、タグが istatus[MPI_TAG] に代入される。
- 戻り値： 整数型。エラーコードが入る。

高速プログラミングためのヒント：

MPI_Send を利用した高速実装法として、以下のように、**通信と計算をオーバラップ**させるように記述する実装方法があります。

```
/* PE 0 が持つデータ a をあらかじめ非同期通信で PE1 から PE nprocs-1 に、送っておく */
if (myid == 0)  {
    for (i=1;  i<numprocs;  i++)  {
        ierr = MPI_Isend( a,  N,  MPI_DOUBLE,  i,
            i_loop,  MPI_COMM_WORLD,  &irequest[i] );
    }
 }  else  {
    ierr = MPI_Recv( a,  N,  MPI_DOUBLE,  0,  i_loop,
                MPI_COMM_WORLD,  &istatus );
}

/* PE0 が 他の PE が受信中に、オーバラップして計算できる部分 */
```

```
/* PE0 の送信領域が書きかえれるか待つ */
if (myid == 0)   {
     for (i=1;  i<numprocs;  i++)  {
         ierr = MPI_Wait(&irequest[i],  &istatus);
     }
}
```

ここで注意は､通信中に計算が行えるハードウエアやＯＳが実装されていることが条件です。もしそうでない場合、MPI_Send を利用した場合や、通信と計算をオーバラップさせないように記述した方法と変わらかったり、むしろ遅くなることも珍しくありません。

４．６　コミュニケータの分割

今までの前提では、初期化した時に存在した PE すべてが、放送処理やリダクション演算に関連します。その関連する PE の集合情報は、コミュニケータとよばれる変数に格納され、通常は MPI_COMM_WORLD であることを説明しました。

放送処理やリダクション演算は、１対１通信よりも時間を必要とする通信であることを説明しました。この処理の時間は、PE 数に関連します。そこで PE 数を減らせば、処理時間の短縮が望めます。

高速プログラミングためのヒント：

リダクション演算などにおいて対象となる PE 数を減らすには､コミュニケータを分割する必要があります。このコミュニケータを分割する関数が、以下に示す MPI_Comm_split 関数です。

```
int MPI_Comm_split(icomm, icolor, ikey, inewcomm);
```

- icomm ：　整数型。分割するコミュニケータを指定する。
  - 通常では MPI_COMM_WORLD を指定すればよい。
- icolor：整数型。分割するコミュニケータにおける同一グループを決める制御変数である。
- ikey ：整数型。分割したコミュニケータ内におけるランクを指定する。
- inewcomm:　整数型。新しいコミュニケータを指定する。
- 戻り値 ：　整数型。エラーコードが入る。

ここで、分割されたコミュニケータは、inewcom になるのですが、この inewcom では、複数 PE の集合になっています(図 7)。

したがって、この新しい inewcom を用いて放送処理やリダクション演算を記述する場合、放送処理やリダクション処理が複数同時に実行される点に注意してください。このことで、それぞれの放送処理やリダクション演算時間が短縮されるだけでなく、並列に処理が実行されることで、さらに高速化されます。

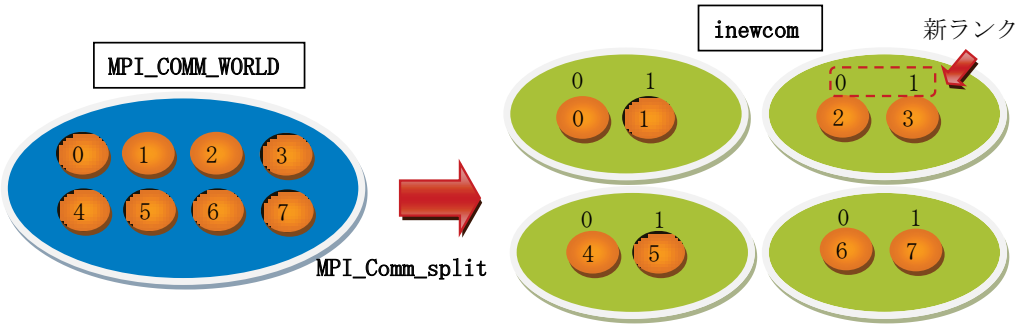

図 7　MPI_Comm_split でのコミュニケータの分割

以下に、プログラム例を示します。

```
int     myid_x, myid_y;
int     MPI_COMM_Y;
…
myid_x = myid / NPROCS_Y;
myid_y = myid % NPROCS_Y;
MPI_Comm_split(MPI_COMM_WORLD, myid_y, myid_x, &MPI_COMM_Y);
…
MPI_Reduce(&iflag, &iflag_t, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_Y);
…
```

以上の例では、0～nprocs-1 に割り当てられた元のランクは、NPROCS_Y 個づつ別のコミュニケータに分割されます。コミュニケータのグループは、myid_y (0～myid%NPROCS_Y-1) で決まります。それぞれのコミュニケータ内のランクは、myid_x となります。最後の MPI_Reduce は、NPROCS_Y 個並列に実行され、それぞれの MPI_Reduce に関連する PE 数は ceil(nprocs/NPROCS_Y) 個です。ここで、ceil(x)は、x の小数点切り上げの整数を返す関数です。

５．実行性能評価

実行性能を評価するにあたり、プログラム中の測定したい箇所の時間をどのように計るのか知る必要があります。ここではまず、その方法を説明します。

５．１　実行時間の測定方法について

MPI では、実行時間測定のための関数として MPI_Wtime があります。使用例を以下に示します。

```
<1 >     double t1, t2, t0, t_w;
<2 >
<3 >     ierr = MPI_Barrier(MPI_COMM_WORLD);
<4 >     t1 = MPI_Wtime();
```

```
<5 >      /*  測定したい部分の始まり */
<6 >            .....
<7 >
<8 >      /*  測定したい部分の終り */
<9 >      ierr = MPI_Barrier(MPI_COMM_WORLD);
<10>      t2 = MPI_Wtime();
<11>      t0 =  t2 - t1;
<12>      ierr = MPI_Reduce(&t0, &t_w, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
<13>
<14>      /* 実行時間は 0 番 PE が持つ */
<15>      if (myid == 0) printf(″  execution time = : %8.4lf  [sec.] ¥n″, t_w);
```

なお、MPI_Barrier 関数は、全 PE で同期をとる関数です。この方法では、各 PE で得られた実行時間の内、最も大きいもの（遅いもの）を全体の実行時間としています。

５．２　性能評価例

それでは最初に示した円周率プログラムを、東京大学情報基盤センターに設置されている HITACHI SR11000/J2 の 1 ノード（最大で 16PE ですが、実験環境では 8PE まで利用可能）を用いて性能評価した例を示します。図 8 に、実行時間をプロットしたものを載せます。

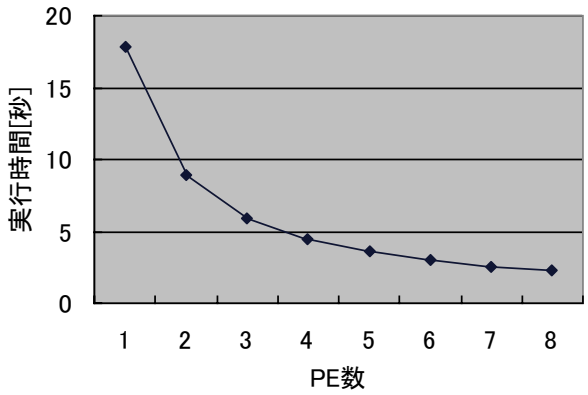

図 8　円周率計算プログラムの実行時間[秒]。n=1,000,000,000 を指定。

ここで図 8 の実行時間が、性能のよいものかどうか評価する尺度の１つとして、台数効果という指標があります。台数効果は以下のように定義されます。

p 台のときの台数効果　：＝ 1 台の実行時間 / p 台の実行時間　　…(1)

式(1)から p 台のときの台数効果が p に近い程、性能が良いということがいえます。
図 9 は円周率のプログラムに関して、台数効果を示したものです。

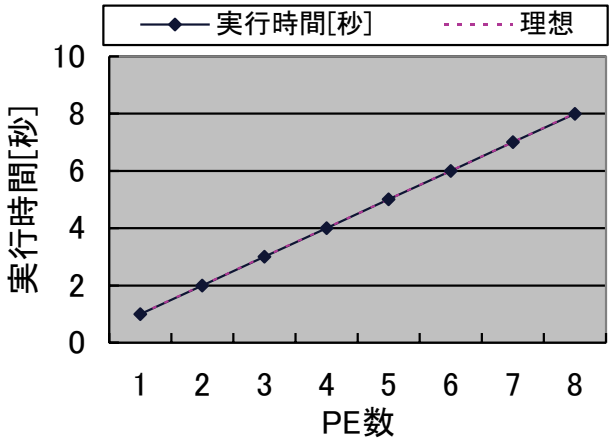

図 9　円周率計算プログラムの台数効果。n=1,000,000,000 を指定。

図 9 の結果から、円周率計算の並列プログラムは、きわめて性能が良いプログラムであるということがいえます[4]。

６．その他の高速プログラミングのためのヒント

数値計算処理において性能を向上させる技法として、以下のものが知られています。

- **最内ループ連続アクセス**：
  ネストされたループにおいて、最も内側のループに対する配列のアクセスが連続となるように、データの複製、ループネストの順番変更、およびデータ構造の変更などを行う技法。連続アクセスにより、データの読みだしに関するキャッシュミスによる速度低下を防いだり、データの書き込み処理の効率化をねらう。たとえば C 言語を用いる場合、2 次元配列は行方向にデータが収納[5]されるため、最内ループのアクセス方向を行方向にするようにループを構成する。
- **ループアンローリング**：
  ループを展開することで、(1)ループ自体のオーバヘッドの削減、(2)データの先行読みだし(プリロード)や書き込み(ポストストア)の機会の増加、(3)レジスタ割り当ての最適化可能性の増大、などコンパイラによるコード最適化を促すことによる高速化技法。
- **ブロック化**：
  頻繁にアクセスするデータをキャッシュにのせることで、キャッシュミスによる速度低下を防ぐ技法。ソースコードのみ変更する場合は、**タイリング**と呼ばれる。タイリングは、

---

[4]台数効果の指標では、1 台利用のときの性能が特に重要となります。すなわち 1 台利用のときの実行時間を効率の悪いものにすると、台数効果は非常によいものとなり、正当な評価にならないことに注意してください。一般的に公平な性能評価をするために 1 台の実行性能は、MPI コードなど並列処理に必要だが逐次処理に不必要なコードの除去をした上で、実行時間を計測すべきです。この性能評価では、MPI コードの除去はしていません。

[5] 行列 A=(a_{ij})とするとき、j 方向のこと。

コンパイラのコード最適化技法の１つとして知られている。

また現在のコンパイラによるコード最適化では実現できない、アルゴリズムレベルでのブロック化技法も知られている。これは、**ブロック化アルゴリズム**と呼ばれる(例：LU 分解における多段多列同時消去アルゴリズム)。

本稿では、以上の技法に関する詳細な説明は割愛します。ここでは、行列積におけるループアンローリングの一例を示すに留めます。

(a) オリジナルコード

```
for (i=0, i<n, i++)
  for (j=0, j<n, j++)
    for (k=0, k<n, k++)
      c[i][j] += a[i][k] * b[k][j];
```

(b) k-ループをアンローリング (n は２で割り切れるとする)

```
for (i=0, i<n, i++)
  for (j=0, j<n, j++)
    for (k=0, k<n, k+=2)
      c[i][j] += a[i][k  ] * b[k  ][j]
               + a[i][k+1] * b[k+1][j];
```

(c) i, j-ループをアンローリング (n は２で割り切れるとする)

```
 for (i=0; i<n;  i+=2)
   for (j=0; j<n; j+=2)
      for (k=0; k<n; k++)  {
         C[i   ][j   ] += A[i  ][k] *B[k][j  ];
         C[i   ][j+１] += A[i  ][k] *B[k][j+1];
         C[i+1 ][j   ] += A[i+1][k] *B[k][j  ];
         C[i+1 ][j+1 ] += A[i+1][k] *B[k][j+1];
      }
```

参 考 文 献

[1] P.パチェコ 著 ／ 秋葉 博 訳、MPI 並列プログラミング、培風館、2001

[2] 青山幸也 著、並列プログラミング虎の巻 MPI 版、理化学研究所情報基盤センター
(http://accc.riken.jp/HPC/training/text.html)

[3] Message Passing Interface Forum
(http://www.mpi-forum.org/)

[4] MPI-J メーリングリスト
(http://phase.hpcc.jp/phase/mpi-j/ml/)

[5] 富田眞治著、並列コンピュータ工学、昭晃堂、1996