

卒業研究論文

並列コンピュータによる素数計算

2002年2月7日

高知工科大学

知能機械システム工学科

磯村研究室

山口 泰三

目 次

	ページ
まえがき	3
1．並列コンピュータ・システムの構成	4
2．ハードウェア構成	6
3．並列計算テストプログラム	7
4．テスト結果	9
結言	11
謝辞	12
参考文献	12
付録 A メッセージパッシング・インターフェイス (MPI)	13

並列コンピュータ・ソフトウェアの性能評価

まえがき

コンピュータによる計算はより速くすることが常に求められてきた。高速計算が必要とされる科学や工学におけるシミュレーションの分野では、しばしば有効な結果を得るために大量のデータに対して多数回の繰返し計算が必要なことが多い。現在の製造現場では、工学的計算やシミュレーションは秒あるいは分単位の時間でできなければならない。設計者が効率的に作業するための時間は短いので、解に到達するのに1ヶ月もかかるシミュレーションは設計環境では通常受け入れられない。

システムが複雑になればなるほど、それをシミュレートする時間はますます増加するし、計算に特定の締め切り時間があることもある。翌日の株価を正確に予測するのに2日かかったら予測は役に立たない。大きな分子構造のモデリングや広域の天気予報のような問題は、現在、コンピュータでは適切な時間内には解けない問題で、実行に10年を要するようでは話にならない。

スーパーコンピュータはそのために開発された。しかしこれは非常に高価である。一つの解決策は安価なパソコン(PC)やワークステーション(W/S)をローカルエリア・ネットワーク(LAN)で接続した並列コンピュータである。

ちなみにUSAの国防省では原爆のシミュレーション用として、256CPU×32システム=8192CPU 並列コンピュータを予算2億ドルで計画中である。

当研究室では脳の記憶・学習機能のシミュレーションを目的として

UNIX マシン 8 CPU

Windows NT マシン 16 CPU

計24CPUの異機種間接続並列コンピュータシステムを試作、並列処理ソフトウェアを開発した。

そこで今回は、前回のデータを元により多くの並列コンピュータで素数および個数の探索を行い、前回のデータと比較し性能の向上を目的して

UNIX マシン 8 CPU

Windows NT マシン 22 CPU

計30CPUの異機種巻接続並列コンピュータシステムを試作、並列処理ソフトウェアを開発した。

テストプログラムの例として

1. 素数およびその個数の探索
2. マルチシリンダーエンジン動特性シミュレーション
3. 3次元波動方程式の求解とCG表示
4. クレーン制御系のシミュレーション

を取り上げ、このソフトウェアの評価テストを行っている。

本稿では、22台の並列CPUを使用し、オイラーの関数すなわち素数の個数計算を例に

とり、並列計算ソフトウェアの性能評価を行った結果について述べている。素数は、現在インターネット上を行き交う情報の暗号化に重要な役割を演ずる数である。

CPU 間の通信時間が CPU の処理時間に比べ無視できるときには、2 台の並列計算が計画通り機能すれば、2 2 台でも可能であり、2 2 台が可能ならば 1 0 0 台でも 8 1 9 2 台でも可能である。ただし、無限の台数とはいかない。LAN の処理時間が CPU 1 台の処理時間にせまるような大量のデータのやりとりがあると、当然システムのスループットは低下する。

1 並列コンピュータ・システムの構成

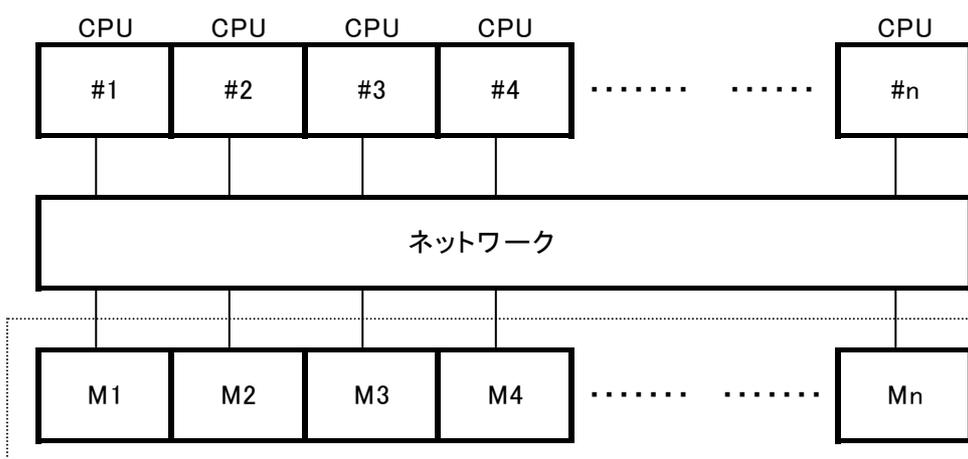
システム構成として次の三つが提案された。

- (1) 共有記憶マルチプロセッサシステム
- (2) 分散共有記憶マルチプロセッサシステム
- (3) メッセージ通信マルチプロセッサシステム

1 - 1 共有記憶マルチプロセッサシステム

単一プロセッサモデルを自然に拡張すると複数のプロセッサを複数のメモリに結合し、各プロセッサがどのメモリモジュールもアクセスできるようにする図 1 に示すような仮想共有記憶構成が得られる。

プロセッサとメモリ間の結合は何らかの形の結合網によっている。共有記憶マルチプロセッサシステムは単一アドレス空間を用いていて、それは主記憶のすべての位置が一意的なアドレスをもち、プロセッサは各位置のアクセスにそのアドレスを使うことを意味する。



仮想共有記憶

図 1 仮想共有記憶マルチプロセッサ構成

この方式では仮想共有記憶が一つのメモリとして機能し、全ての CPU からアクセス可能となる。

(2) 分散共有記憶マルチプロセッサシステム

各プロセッサが単一のアドレス空間を用いて全メモリをアクセスする分散共有記憶システムである。

この場合、プロセッサが自分のローカルメモリにない場所をアクセスするときは、データを他のプロセッサから受け取るのに、メモリが分散しているという事実を隠すために何らかの形で自動的にメッセージ通信を発生させなければならない。(図 2)

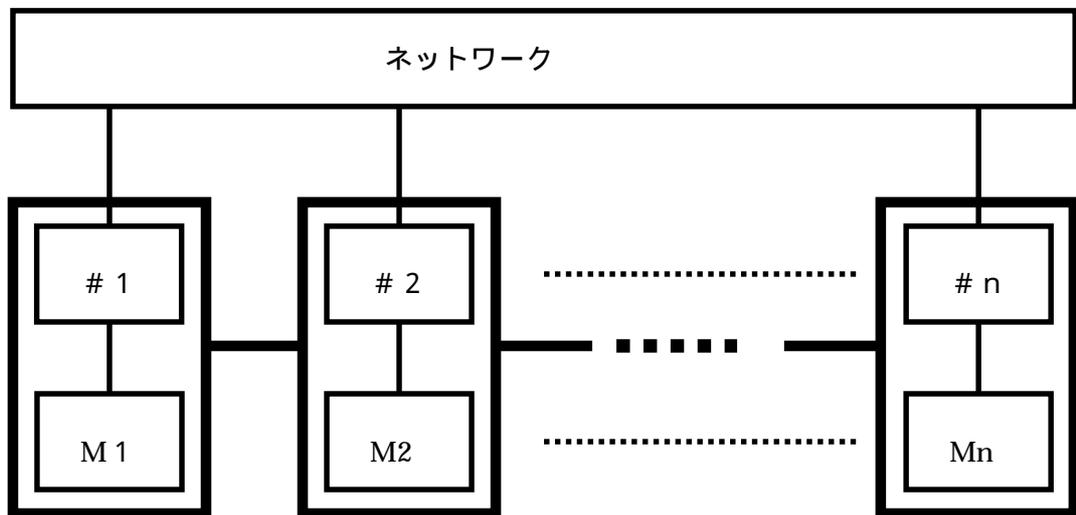


図 2 分散共有記憶マルチプロセッサ構成

したがって、この構成では見かけ上、各メモリ M1 , M2 , M3 , . . . Mn が全ての CPU からアクセスできるようにネットワークが機能しなくてはならない。

(3) メッセージ通信マルチプロセッサシステム

マルチプロセッサシステムのもう一つの形は、図 3 に示すように、完全なコンピュータを結合することによってつくれる。各コンピュータはプロセッサと他のプロセッサからはアクセスできないローカルメモリで構成される。すなわち、マルチコンピュータではメモリはコンピュータ間に分散され、各コンピュータは自分のアドレス空間を有し、プロセッサは自分のメモリのロケーションのみをアクセスできる。プロセッサが他のメモリにメッセージを送信できるように結合網が用意されている。このメッセージには他のプロセッサの計算に必要なデータが含まれる。この種のマルチプロセッサシステムは通常メッセージ

通信マルチプロセッサとよばれ、特に独立して動作できるそれだけで完結したコンピュータから構成されている場合には単にマルチコンピュータともよばれる。このメッセージ通信マルチコンピュータにおけるメッセージは一つのプロセッサから他のプロセッサへプログラムの指定に従ってデータを送る。

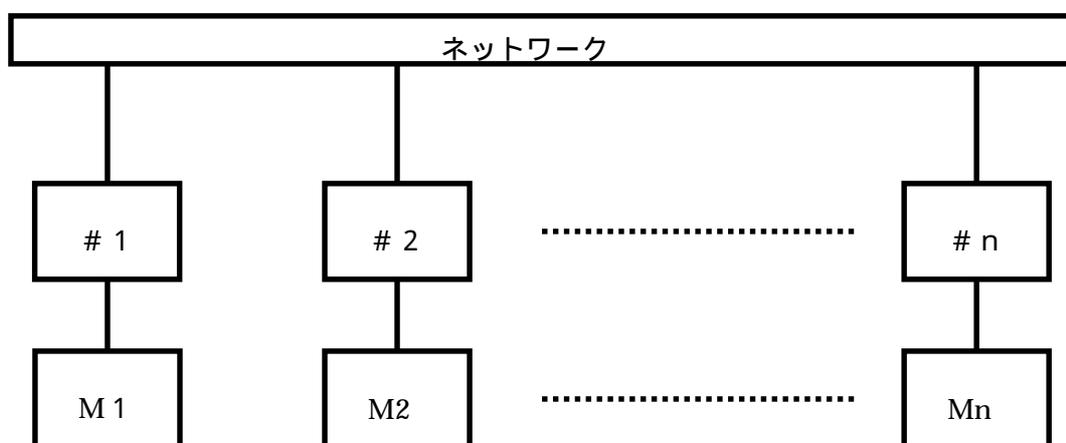


図 3 メッセージ通信マルチプロセッサシステム

メッセージ通信マルチコンピュータのプログラミングでは問題を同時に実行する部分に分割する。このプログラミングには並列言語や拡張逐次言語を使うこともできるが、よく行われる方法はメッセージ通信ライブラリルーチンを用いるものである。問題の独立した並列実行可能な部分をしばしばプロセスとよぶ。

すなわち、問題を多数の並列プロセスに分割する。プロセスはそれぞれ別のコンピュータで実行してもよく、6個のプロセスと6台のコンピュータがあるならそれぞれのプロセスを1台のコンピュータで実行すればよいが、もし、コンピュータ数よりプロセス数が多ければ複数のプロセスを1台のコンピュータ上で時分割実行することになるだろう。プロセスはメッセージを送信することで通信し、それが、データや結果をプロセス間に分配する唯一の方法である。我々の研究室でこのシステムを採用し、メッセージ通信には MPI (Message Passing Interface) ライブラリの一つ MPICH を ANL (Argonne National Laboratory) からダウンロードして使用した。

2 ソフトウェア評価用ハードウェア構成

連携研究センターには次のコンピュータが設置され、LAN で接続されている。

	コンピュータ名(機種)	プロセス	クロック
Intergraph1	(Visual W/S)	0	750MHz
Intergraph2	(Visual W/S)	1	750MHz
Intergraph3	(W/S)	2	750MHz
NEC	(Express 5800)	3	550MHz
Silicon	(Visual WS 5 4 0)	4	550MHz
Shuttle1	(Fv24)	5	902MHz
Shuttle2	(Fv24)	6	902MHz
Shuttle3	(Fv24)	7	902MHz
Shuttle4	(Fv24)	8	902MHz
Compaq1	(SP750 & AP550X2)	9	733MHz
Compaq2	(SP750 & AP550X2)	10	733MHz
Compaq3	(SP750 & AP550X2)	11	733MHz
Compaq4	(AP550)	12	733MHz
Compaq10	(SP750 & AP550X2)	13	2400MHz
Compaq10	(SP750 & AP550X2)	14	2400MHz
Compaq10	(SP750 & AP550X2)	15	2400MHz
Compaq11	(Evo W4000)	16	2400MHz
Compaq12	(Evo W4000)	17	2400MHz
Compaq13	(Evo W4000)	18	2400MHz
Compaq14	(Evo W4000)	19	2400MHz
Compaq15	(Evo W4000)	20	2400MHz
Compaq16	(Evo W4000)	21	2400MHz
Compaq17	(Evo W4000)	22	2400MHz
Compaq18	(Evo W4000)	23	2400MHz
Compaq11	(Evo W4000)	24	2400MHz
Compaq12	(Evo W4000)	25	2400MHz
Compaq13	(Evo W4000)	26	2400MHz
Compaq14	(Evo W4000)	27	2400MHz
Compaq15	(Evo W4000)	28	2400MHz
Compaq16	(Evo W4000)	29	2400MHz
Compaq17	(Evo W4000)	30	2400MHz
Compaq18	(Evo W4000)	31	2400MHz

計 2 2 CPU 3 2 プロセス

3 並列計算テストプログラム

2 2 台のワークステーションを LAN で接続した並列コンピュータ・システムの性能をテストするため、1 から n (ここでは n = 3 2 0 0 0 0) までの数の中に素数はいくつあるか

数えるオイラー関数のテストプログラムを使用した。

ちなみに、桁数の大きな素数はインターネット上を行き交う情報の暗号化と解読に大きな力を発揮する公開鍵暗号に使われる。大きな素数の掛け算は簡単であるが、その積が与えられて因数に分解するのは極めて難しい。このように逆算の困難は関数のことを一方向関数といい、オイラー関数はそのような関数の一例である。

整数 n に対し、 $1, 2, 3, \dots, n-1$ のうちで n と互いに素なものの個数を $\phi(n)$ で表し、これをオイラーの関数という。これは $1/n, 2/n, 3/n, \dots, n-1/n$ のなかの既約分数の個数に等しい。

3 - 1 素数の個数を求める並列計算アルゴリズム

素数の個数を求めようとしている数 n を

min0 ~ max0

min1 ~ max1

min2 ~ max2

⋮

⋮

⋮

min 3 1 ~ max 3 1

というように 3 2 グループに分け、#0 から # 3 1 までの CPU に配分する。

#0 CPU をマスターとし、# 1 ~ # 3 1 CPU はスレーブとする。#0 CPU は数 min_i と数 max_i ($i=1, 2, \dots, 31$) を作り、# i CPU に min_i と max_i を送る。受け取った # i CPU は min_i から max_i までの数に素数がいくつあるか調べて結果を #0 CPU すなわちマスターに送り返す。

#0 CPU は min0 から max0 までの素数を調べると同時に # i CPU ($i=1, 2, \dots, 31$) から受け取った素数の数を集計する。

素数の個数を求めるアルゴリズムは次の通りである。

(1) 素数の個数のカウンタをリセット

(2) 数 n を読み込む

(3) 1 及びその数自身 ($= n$) 以外の数 i ($1 < i < n$) で割ったとき、剰余 $R(n/i)$ が 0 になるような数が一つでも存在するときは、素数ではないのでカウンタはそのまま進めない

(4) 存在しないときは素数なのでその値をセーブしカウンタを一つ進める

(5) (1)へ戻る

3 - 2 アプリケーションプログラムのインストール

前節のアルゴリズムにしたがってプログラムを書き、マスター CPU 上でコンパイルす

る。コンパイラは visual C++を使用した。ビルドを行って実行可能なプログラムを作成、7台のスレーブ CPU に送り込む。

3 - 3 実行

1台のマスター、31台のスレーブ CPU の名前、プロセスのディレクトリ、実行プログラムファイル名を記入したコンフィギュレーションファイル conf.txt を作る。MPI 実行コマンド MPIRun conf.txt で CPU から全てのプロセスを起動する。

4 テスト結果

(1) 一回目の計算結果

n=320,000 として与えた。最初は m 個のプロセスに均等配分して

$$n_1^{(1)}=n_2^{(1)}=n_3^{(1)}=\dots=n_m^{(1)}=n/m$$

とする。N=320,000、m=8 なので

$$n_1^{(1)}=n_2^{(1)}=n_3^{(1)}=\dots=n_8^{(1)}=10,000$$

である。

実行結果は次のようになった。

```
C:\¥mpi¥C¥Prime320K>mpirun prime320.txt
```

```
rank=31 310001 <=> 320000 Number of prime sent from Compaq18-2 to Intergraph1  
>> 808
```

```
CPU Time=17265msec
```

```
rank=30 300001 <=> 310000 Number of prime sent from ompaq17-2 to Intergraph1  
>> 803
```

```
CPU Time=16828msec
```

```
rank=29 290001 <=> 300000 Number of prime sent from Compaq16-2 to Intergraph1  
>> 773
```

```
CPU Time=15937msec
```

```
rank=28 280001 <=> 290000 Number of prime sent from Compaq15-2 to Intergraph1  
>> 792
```

```
CPU Time=15546msec
```

```
rank=26 260001 <=> 270000 Number of prime sent from Compaq13-2 to Intergraph1  
>> 805
```

CPU Time=14437msec

rank=27 270001 <=> 280000 Number of prime sent from Intergraph3-2 to Intergraph1-1>>
790

rank=27 270001 <=> 280000 Number of prime sent from Compaq14-2 to Intergraph1
>> 790

CPU Time=14828msec

rank=25 250001 <=> 260000 Number of prime sent from Compaq12-2 to Intergraph1
>> 793

CPU Time=13875msec

rank=23 230001 <=> 240000 Number of prime sent from Compaq18-1 to Intergraph1
>> 784

CPU Time=13750msec

rank=21 210001 <=> 220000 Number of prime sent from Compaq16-1 to Intergraph1
>> 811

CPU Time=12765msec

rank=24 240001 <=> 250000 Number of prime sent from Compaq11-2 to Intergraph1
>> 823

CPU Time=13281msec

rank=22 220001 <=> 230000 Number of prime sent from Compaq17-1 to Intergraph1
>> 819

CPU Time=13484msec

rank=17 170001 <=> 180000 Number of prime sent from Compaq12-1 to Intergraph1
>> 845

CPU Time=10671msec

rank=20 200001 <=> 210000 Number of prime sent from Compaq15-1 to Intergraph1
>> 823

CPU Time=12343msec

rank=18 180001 <=> 190000 Number of prime sent from Compaq13-1 to Intergraph1
>> 828

CPU Time=11109msec

rank=19 190001 <=> 200000 Number of prime sent from Compaq14-1 to Intergraph1
>> 814

CPU Time=11546msec

rank=16 160001 <=> 170000 Number of prime sent from Compaq11-1 to Intergraph1
>> 814

CPU Time=9796msec

rank=15 150001 <=> 160000 Number of prime sent from Compaq10-3 to Intergraph1
>> 835

CPU Time=13250msec

rank=14 140001 <=> 150000 Number of prime sent from Compaq10-2 to Intergraph1
>> 838

CPU Time=12937msec

rank=13 130001 <=> 140000 Number of prime sent from Compaq10-1 to Intergraph1
>> 851

CPU Time=12093msec

rank=12 120001 <=> 130000 Number of prime sent from Shuttle4 to Intergraph1
>> 858

CPU Time=8352msec

rank=11 110001 <=> 120000 Number of prime sent from Shuttle3 to Intergraph1
>> 848

CPU Time=7871msec

rank= 8 80001 <=> 90000 Number of prime sent from Compaq4 to Intergraph1
>> 876

CPU Time=6468msec

rank=10 10001 <=> 110000 Number of prime sent from Shuttle2 to Intergraph1
>> 861

CPU Time=7360sec

rank= 9 90001 <=> 100000 Number of prime sent from Shuttle1 to Intergraph1
>> 879

CPU Time=6799msec

rank= 4 40001 <=> 50000 Number of prime sent from Toshiba4 to Intergraph1-1>>
930

CPU Time=4140msec

rank= 3 30001 <=> 40000 Number of prime sent from Nec to Intergraph1
>> 958

CPU Time=3284msec

rank= 5 50001 <=> 60000 Number of prime sent from Compaq1 to Intergraph1
>> 924

CPU Time=5015msec

rank= 7 70001 <=> 80000 Number of prime sent from Compaq3 to Intergraph1
>> 902

CPU Time=5546msec

rank= 1 10001 <=> 20000 Number of prime sent from Intergraph2 to Intergraph1
>> 1033

CPU Time=1531msec

rank= 2 20001 <=> 30000 Number of prime sent from Intergraph3 to Intergraph1
>> 983

CPU Time=2218msec

rank= 0 1 <=> 10000 Number of prime sent from Intergraph1 to Intergraph1
>> 1229

CPU Time=10500msec

Number of processes=32

```
1 <=> 10000 Number of prime receipt from Intergraph1 >> 1229
10001 <=> 20000 Number of prime receipt from Intergraph2 >> 1033
20001 <=> 30000 Number of prime receipt from Intergraph3 >> 983
30001 <=> 40000 Number of prime receipt from Nec >> 958
40001 <=> 50000 Number of prime receipt from Silicon >> 930
50001 <=> 60000 Number of prime receipt from Compaq1 >> 924
60001 <=> 70000 Number of prime receipt from Compaq2 >> 878
70001 <=> 80000 Number of prime receipt from Compaq3 >> 902
80001 <=> 90000 Number of prime receipt from Compaq4 >> 876
90001 <=> 100000 Number of prime receipt from Shuttle1 >> 879
100001 <=> 110000 Number of prime receipt from Shuttle2 >> 861
110001 <=> 120000 Number of prime receipt from Shuttle3 >> 848
120001 <=>rank= 6 60001 <=> 70000 Number of prime sent from Compaq2 to
Intergraph1 >> 878
```

CPU Time=4812msec

```
130000 Number of prime receipt from Shuttle4 >> 858
130001 <=> 140000 Number of prime receipt from Compaq10-1 >> 851
140001 <=> 150000 Number of prime receipt from Compaq10-2 >> 838
150001 <=> 160000 Number of prime receipt from Compaq10-3 >> 835
160001 <=> 170000 Number of prime receipt from Compaq11-1 >> 814
170001 <=> 180000 Number of prime receipt from Compaq12-1 >> 845
180001 <=> 190000 Number of prime receipt from Compaq13-1 >> 828
190001 <=> 200000 Number of prime receipt from Compaq14-1 >> 814
200001 <=> 210000 Number of prime receipt from Compaq15-1 >> 823
210001 <=> 220000 Number of prime receipt from Compaq16-1 >> 811
220001 <=> 230000 Number of prime receipt from Compaq17-1 >> 819
230001 <=> 240000 Number of prime receipt from Compaq18-1 >> 784
240001 <=> 250000 Number of prime receipt from Compaq11-2 >> 823
250001 <=> 260000 Number of prime receipt from Compaq12-2 >> 793
260001 <=> 270000 Number of prime receipt from Compaq13-2 >> 805
270001 <=> 280000 Number of prime receipt from Compaq14-2 >> 790
280001 <=> 290000 Number of prime receipt from Compaq15-2 >> 792
```

290001 <=> 300000 Number of prime receipt from Compaq16-2 >> 773
 300001 <=> 310000 Number of prime receipt from Compaq17-2 >> 803
 310001 <=> 320000 Number of prime receipt from Compaq18-2 >> 808

Total number of prime=27608

CPU Time=19125msec

C:\mpi\C\Prime320K>

処理時間に 10 倍以上開きがあるほど計算機負荷がアンバランスでは、並列化のメリットが十分ではない。

(3) 2 回目の配分

2 回目の配分 $n_1^{(2)}$ 、 $n_2^{(2)}$ 、 $n_3^{(2)}$ 、……、 $n_m^{(2)}$ は次のようにして求められる。

$$\frac{\tau_1^{(1)}}{n_1^{(1)}} x_1 = \frac{\tau_2^{(1)}}{n_2^{(1)}} x_2 = \frac{\tau_3^{(1)}}{n_3^{(1)}} x_3 = \dots = \frac{\tau_m^{(1)}}{n_m^{(1)}} x_m = \tau^{(1)}$$

$$\tau^{(1)} \sum_{i=1}^m \frac{n_i^{(1)}}{\tau_i^{(1)}} = n$$

$$\tau^{(1)} = \frac{n}{\sum_{i=1}^m \frac{n_i^{(1)}}{\tau_i^{(1)}}}$$

$$x_i = \frac{n}{\sum_{i=1}^m \frac{n_i^{(1)}}{\tau_i^{(1)}}} \frac{n_i^{(1)}}{\tau_i^{(1)}} \quad (i = 1, 2, 3, \dots, m)$$

$$n_i^{(2)} = x_i \quad (i = 1, 2, 3, \dots, m)$$

C:\mpi\C\Prime320K>mpirun prime320.txt

rank=31 315845 <=> 320000 Number of prime sent from Compaq18-2 to Intergraph1 >> 347

CPU Time=11281msec

rank=30 311581 <=> 315844 Number of prime sent from ompaq17-2 to Intergraph1 >> 340

CPU Time=10968msec

rank=29 307078 <=> 311580 Number of prime sent from Compaq16-2 to Intergraph1 >> 345

CPU Time=10953msec

rank= 8 167387 <=> 178479 Number of prime sent from Compaq4 to Intergraph1 >> 908

CPU Time=10875msec

rank=25 287483 <=> 292653 Number of prime sent from Compaq12-2 to Intergraph1 >> 405

CPU Time=10500msec

rank=27 297624 <=> 302462 Number of prime sent from Intergraph3-2 to Intergraph1-1>> 374

rank=27 297624 <=> 302462 Number of prime sent from Compaq14-2 to Intergraph1 >> 374

CPU Time=10703msec

rank=26 292654 <=> 297623 Number of prime sent from Compaq13-2 to Intergraph1 >> 384

CPU Time=10734msec

rank=24 282080 <=> 287482 Number of prime sent from Compaq11-2 to Intergraph1 >> 419

CPU Time=10406msec

rank=22 271540 <=> 276861 Number of prime sent from Compaq17-1 to Intergraph1 >> 425

CPU Time=9406msec

rank=23 276862 <=> 282079 Number of prime sent from Compaq18-1 to Intergraph1 >> 417

CPU Time=9656msec

rank=20 260106 <=> 265918 Number of prime sent from Compaq15-1 to Intergraph1 >> 459

CPU Time=9140msec

rank=17 240709 <=> 247432 Number of prime sent from Compaq12-1 to Intergraph1 >> 550

CPU Time=8953msec

rank=21 265919 <=> 271539 Number of prime sent from Compaq16-1 to Intergraph1 >> 464

CPU Time=9578msec

rank=19 253892 <=> 260105 Number of prime sent from Compaq14-1 to Intergraph1 >> 494

CPU Time=9343msec

rank=18 247433 <=> 253891 Number of prime sent from Compaq13-1 to Intergraph1 >> 524

CPU Time=9500msec

rank=15 227969 <=> 233383 Number of prime sent from Compaq10-3 to Intergraph1 >> 447

CPU Time=11078msec

rank=16 233384 <=> 240708 Number of prime sent from Compaq11-1 to Intergraph1 >> 573

CPU Time=8875msec

rank=14 222423 <=> 227968 Number of prime sent from Compaq10-2 to Intergraph1 >> 443

CPU Time=10796msec

rank=12 207898 <=> 216488 Number of prime sent from Shuttle4 to Intergraph1 >> 696

CPU Time=9653msec

rank=11 198782 <=> 207897 Number of prime sent from Shuttle3 to Intergraph1 >> 740

CPU Time=9563msec

rank=10 189033 <=> 198781 Number of prime sent from Shuttle2 to Intergraph1 >> 800

CPU Time=9563sec

rank=13 216489 <=> 222422 Number of prime sent from Compaq10-1 to Intergraph1 >> 493

CPU Time=10718msec

rank= 9 178480 <=> 189032 Number of prime sent from Shuttle1 to Intergraph1 >> 887

CPU Time=9663msec

rank=28 302463 <=> 307077 Number of prime sent from Compaq15-2 to Intergraph1 >> 384

CPU Time=10906msec

rank= 7 154449 <=> 167386 Number of prime sent from Compaq3 to Intergraph1 >>
1062

CPU Time=11390msec

rank= 5 125231 <=> 139537 Number of prime sent from Compaq1 to Intergraph1 >>
1216

CPU Time=13312msec

rank= 4 107899 <=> 125230 Number of prime sent from Toshiba4 to Intergraph1-1>> 1488

CPU Time=13910msec

rank= 2 53700 <=> 86049 Number of prime sent from Intergraph3 to Intergraph1 >> 2893

CPU Time=11781msec

rank= 3 86050 <=> 107898 Number of prime sent from Nec to Intergraph1 >> 1900

CPU Time=14480msec

rank= 1 6835 <=> 53699 Number of prime sent from Intergraph2 to Intergraph1 >> 4593

CPU Time=8140msec

rank= 0 1 <=> 6834 Number of prime sent from Intergraph1 to Intergraph1 >> 880

CPU Time=8109msec

Number of processes=32

1 <=> 6834 Number of prime receipt from Intergraph1 >> 880
6835 <=> 53699 Number of prime receipt from Intergraph2 >> 4593
53700 <=> 86049 Number of prime receipt from Intergraph3 >> 2893
86050 <=> 107898 Number of prime receipt from Nec >> 1900
107899 <=> 125230 Number of prime receipt from Silicon >> 1488
125231 <=> 139537 Number of prime receipt from Compaq1 >> 1216
139538 <=> 154448 Number of prime receipt from Compaq2 >> 1258

154449 <=> 167386 Number of prime receipt from Compaq3 >> 1062
 167387 <=> 178479 Number of prime receipt from Compaq4 >> 908
 178480 <=> 189032 Number of prime receipt from Shuttle1 >> 887
 189033 <=> 198781 Number of prime receipt from Shuttle2 >> 800
 198782 <=> 207897 Number of prime receipt from Shuttle3 >> 740
 207898 <=> 216488 Number of prime receipt from Shuttle4 >> 696
 216489 <=> 222422 Number of prime receipt from Compaq10-1 >> 493
 222423 <=> 227968 Number of prime receipt from Compaq10-2 >> 443
 227969 <=> 233383 Number of prime receipt from Compaq10-3 >> 447
 233384 <=> 240708 Number of prime receipt from Compaq11-1 >> 573
 240709 <=> 247432 Number of prime receipt from Compaq12-1 >> 550
 247433 <=> 253891 Number of prime receipt from Compaq13-1 >> 524
 253892 <=> 260105 Number of prime receipt from Compaq14-1 >> 494
 260106 <=> 265918 Number of prime receipt from Compaq15-1 >> 459
 265919 <=> 271539 Number of prime receipt from Compaq16-1 >> 464
 271540 <=> 276861 Number of prime receipt from Compaq17-1 >> 425
 276862 <=> 282079 Number of prime receipt from Compaq18-1 >> 417
 282080 <=> 287482 Number of prime receipt from Compaq11-2 >> 419
 287483 <=> 292653 Number of prime receipt from Compaq12-2 >> 405
 292654 <=> 297623 Number of frank= 6 139538 <=> 154448 Number of prime sent from
 Compaq2 to Intergraph1 >> 1258

CPU Time=11906msec

prime receipt from Compaq13-2 >> 384
 297624 <=> 302462 Number of prime receipt from Compaq14-2 >> 374
 302463 <=> 307077 Number of prime receipt from Compaq15-2 >> 384
 307078 <=> 311580 Number of prime receipt from Compaq16-2 >> 345
 311581 <=> 315844 Number of prime receipt from Compaq17-2 >> 340
 315845 <=> 320000 Number of prime receipt from Compaq18-2 >> 347

Total number of prime=27608

CPU Time=14843msec

C:\mpi\C\Prime320K>

結言

22台のワークステーションをLANで接続、MPICHライブラリを用いて並列化したシステム上で、数320000にある素数の個数を求めるテストプログラムを走らせ性能をテストし、次の結果を得た。

CPUの負荷が均一になるよう数を配分すると、計算時間は19125ミリ秒が14843ミリ秒になった。

1. $n = 320000$ を等分して分担した場合、3100001から3200000を分担するCPUがもっとも時間がかかり、17秒であった。
2. 最も少ない時間を要したのは10001~20001を分担した。CPUで1.5秒で、CPUの負荷のアンバランスは10倍を超えた。CPUの処理時間の最も短いプロセスは10001~20000を分担したプロセスで1.5秒であった。したがって、その差は15.8秒である。
4. 2回目の計算で、CPUの処理時間の最も長いプロセスの計算時間は14.5秒最も短いプロセスの計算時間は8.1秒で、その差が6.4秒に縮んだ。
5. **Intergraph1 (733MHz)** 1台で
1~320000の素数の個数を求めたときの計算時間は221.4秒であった。
これは、22CPUに比べ**15倍**になる。
Compaq10 (2400MHz) 1台で
1~320000の素数の個数を求めたとき
計算時間は103.7秒であった。
これは、22CPUに比べ**12倍**になる。

今後の課題として、個数探索の初期段階で配分を最適化するアルゴリズムの開発が考えられる。

謝辞

本研究では、磯村研究室で開発中の並列コンピュータ・システムを使用した。担当教員の磯村教授に感謝する。

参考文献

Barry Wilkinson and Michael Allen ; Parallel Programming, 1999 by Prentice-Hall, Inc.

参考ウェブサイト

“ Argonne National Laboratory “ , <http://www.anl.gov/>

“ MCS: Mathematics and Computer Science Division ” ,

<http://www-fp.mcs.anl.gov/division/welcome/default.asp>

“ MPI Forum “ , <http://www-unix.mcs.anl.gov/mpi/>

“ MPICH Home Page ” , <http://www-unix.mcs.anl.gov/mpi/mpich/>

[http://www9.ocn.ne.jp/~isomura/個人研究](http://www9.ocn.ne.jp/~isomura/個人研究/Prime)”Prime Calculation by Parallel Computer”

付録 A メッセージパッシング・インターフェイス (MPI)

2つのプロセスの間でのメッセージのやりとりは、比較的分かりやすい概念である。あえて別の見方を挙げると、メッセージ通信は、あるプロセッサのメモリ空間から、もう一方のプロセスのメモリ空間へのデータの転送とみることにもできる。もちろん一般には、メッセージをやりとりするのは2つのプロセスとは限らず、もっと多くのプロセスが関与し得る。

分散メモリ型の並列計算機では、メッセージ通信によるプログラミングが最もハードウェアの構成に即したプログラミングであり、最高の性能が期待できる。また、共有メモリ型の並列計算機でも、メッセージ通信で書いたプログラムを実行するのは容易である。

MPI は現在最も主要なメッセージ通信ライブラリの一つである。

MPI は非常に豊富な機能を持つシステムであり、MPI のごく基本的な使い方と、利用頻度が高いと思われる、3つ以上のプロセスによる集団通信のための関数群を中心に述べる。

MPI は歴史的経緯により、基本仕様 (MPI-1) と拡張機能仕様 (MPI-2) という2つの部分に分かれているが、MPI-1 のみを扱う。また、MPI は Fortran, C および C++ の3つの言語からの利用形式を規定しているが、ここでは、C および C++ のみを使用した。

MPICH は、恐らく最も重要な MPI ライブラリである。アメリカのアーゴン国立研究所 (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ている。移植しやすさを重視したつくりになっているため盛んに移植が行われ、日本を含め、世界中のほとんどのベンダの並列マシンの上で利用できる。また LAN 環境でも利用できる。

MPI-1 には 127 個の関数があるが、最低 6 個の関数だけで、任意個のプロセスの間での 1対1通信を始めることができる。ここでは、これら 6 個の関数に限定した。

1対1の通信があれば、それを組み合わせることで、任意の通信パターンを表現できる。しかし、MPI には、1対1の通信のほかに、様々な通信パターンのための関数がある。これら通信パターン向け関数を使うことは、単にプログラムがすっきりするだけでなく、より高い性能を得ることができる。

最初に、通信が起きる舞台となる環境について述べる。ここで考える並列処理は、幾つかのプロセスによる並列処理である。幾つかのプロセスが強調して、1つのアプリケーションを実行する。これらのプロセスは、すべて同じ計算機上にあるかもしれないし、幾つもの計算機上にあるかもしれない。分散している場合でも、そのうちの幾つかのプロセスは、同じ計算機上にあるかもしれない。

1つのアプリケーションの実行にかかわる全プロセスは、同じプログラムを実行しているかもしれないし、幾つかのプロセスは他のプロセスとは違うプログラムを実行しているかもしれない。

こういったプロセスとプログラムの配置は、MPI-1 では、前提として、アプリケーションの実行が始まる前にすべて決まっていることになっている。MPI-1 では、1つのアプリケーションの実行にかかわる全プロセスは、一斉に実行を開始し、また、一斉に終了する。その総数はアプリケーションの処理が続く間、変わらない。言い換えると、MPI-1 を使って新たなプロセスを起動したり、プロセスを終了させたりすることができない。MPI-2 では新しいプロセスを起動することができるが、ここでは扱わない。

では、プロセスとプログラムの配置はどうやって決めればよいか。これは、MPI プログラムを実行するために使う並列プラットフォームに依存する。すなわち、並列マシンや LAN 環境などのハードウェアと、その上の OS など、並列実行環境を提供するソフトウェアとして何を使うかによって、その方法が異なる。ここでは、代表例として、MPICH を LAN 環境で使う場合について述べる。

MPI プログラムはコマンド行でスタートされ、たとえば、実装に依存して

```
mpirun prog1-mp4
```

または、

```
prog1-mp4
```

により、同じ実行可能プログラムを別のプロセッサで同時に起動できる。

これらのコマンドは実行可能プログラム prog1 の4個のコピーをスタートするが、どこでそれを実行するかについては何も指定していない。プロセスのプロセッサへのマッピングも MPI 標準には定義されていなくて、必要なら実装ごとに別に提供しなければならない。特定のマッピングはコマンド行で指定されたり、実行可能プログラムの名称とそれを実行するプロセッサの名を保持するファイルを用いて行われる。

はじめて MPI 関数コールをする前にコードを MPI_Init() で初期化し、すべての MPI 関数コールの後で MPI_Finalize() で終了させねばならない。コマンド行引数を MPI_Init() に渡して MPI のセットアップ操作をすることができる。すなわち

```
main (int argc, char *argv[ ])
{
    MPI_Init(&argc, &argv);      /*initialize MPI */
    .
    .
    .
    MPI_finalize();             /*terminate MPI */
}
```

(逐次の C プログラムと同様に、&argc、すなわち引数カウントは引数の個数を与え、&argv、

すなわち引数ベクトルは文字ストリングの配列へのポインタである。)

最初は、全プロセスは MPI_COMM_WORLD とよばれる「世界」に登録されていて、それぞれ一意のランク (n プロセッサの場合 0 から n - 1 の番号) を与えられている。MPI の用語では MPI_COMM_WORLD は通信操作の範囲を定義するコミュニケータ (communicator) で、プロセスはコミュニケータに関連づけられたランクを有する。プロセスのグループをつくるためにこれ以外のコミュニケータを設定することもできるが、簡単なプログラムではデフォルトのコミュニケータ MPI_COMM_WORLD で十分である。しかし、この概念によってメッセージに対して別の範囲を有するプログラム、特にライブラリを構築することが可能である。

A - 1 単一プログラム複数データモデル (SPMD)

このモデルは各プロセスが事実上同じコードを実行するときに理想的であるが、普通はあらゆる応用において 1 個あるいはそれ以上のプロセッサが別のコードを実行する必要がある。一つのプログラムでそれをできるようにするためには、それぞれのプロセッサがコードのどの部分を実行するかを選択する文を挿入する必要がある。よって、SPMD モデルでも、マスタコードとスレーブコードを同じプログラム内に置かねばならないためにマスタスレーブアプローチを使えないということはない。次の部分 MPI コードはどうすればそれができるかを示している。

```
main (int argc, char *argv[])
{
    MPI_Init (&argc, &argv);
    .
    .
    .
    MPI_Comm_rank ( MPI_COMM_WORLD, &myrank );
                                     /*find process rank */
    if ( myrank==0 )
        master ( );
    else
        slave ( );
    .
    .
    .
    MPI_Finalize ( );
}
```

ここで、master() と slave() はそれぞれマスタプロセスとスレーブプロセスによって実行さ

れる手続きである。この方法は3個以上のコード列に対しても使うことができる。SPMDモデルは、各プロセッサが完全に異なるコードを実行する場合はメモリ利用の面で非効率的であるが、幸いなことにその必要性はあまりない。SPMDモデルのもう一つの利点はコマンド行引数を各プロセスに渡せることである。

SPMDモデルを使うことにすると、変数の大域宣言は各プロセスで複製することになり、複製されない変数は局所的に宣言する必要がある。すなわち、そのプロセスでのみ実行されるコード内で宣言する。たとえば、

```
MPI_Comm_rank ( MPI_COMM_WORLD , &myrank ); /*find process rank */
    if ( myrank==0 ){                          /*process 0 actions/local variables */
        int x , y ;
        .
        .
        .
    } else if ( myrank== 1 ){                  /*process 1 actions/local variables
*/
        int x , y ;
        .
        .
        .
    }
```

ここで、プロセス0のxとyはプロセス1のxとyとは異なる局所変数である。

データの大きさが変化する場合には、メモリはmalloc()を用いて動的に割り付けられる。たとえば、

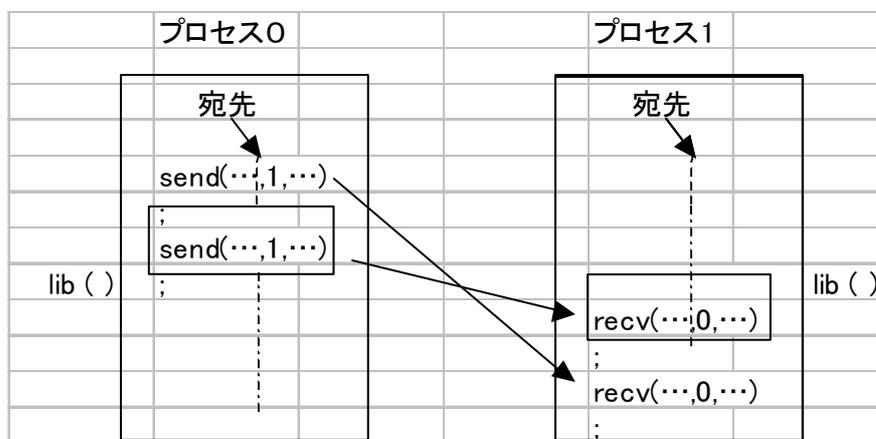
```
    if ( myrank==0 ){
        int *data;
        data = ( int *)malloc(nproc * sizeof ( int ) );
        .
        .
    }
```

A - 2 メッセージ通信ルーチン

メッセージ通信は誤った操作の原因になることがあり、MPIは安全な通信環境を提供することを意図している。安全でない通信の例を図A1に示した。この図において、プロセス0はプロセス1にメッセージを送信しようとしているが、ライブラリルーチン間でもメッ

メッセージ通信がある。この場合、各 send/recv の組は発信元と宛先がマッチしていても誤ったメッセージ通信が発生する。ワイルドカードを用いると、正しくない操作やデッドロックはさらに起きやすくなる。一つのプロセス内の非ブロック型受信の発信元とタグが両方ともワイルドカードで、別の二つのプロセスの組がメッセージ通信を必要とするライブラリコールをしたとしよう。このライブラリルーチンの最初の送信はワイルドカードを用いている非ブロック型ルーチンとマッチして誤った動作をすることがありうる。

MPI では、すべての1対1および集散的メッセージ通信においてコミュニケータが用いられる。コミュニケータとは相互に通信することが許されるプロセスの集合を定義する通信領域である。これを用いれば、ライブラリの通信領域をユーザープログラムの領域と分離することができる。各プロセッサはコミュニケータ内で0からn-1(nはプロセッサ数)までの整数であるランクを有する。実際にはコミュニケータには、グループ内の通信用のイントラコミュニケータ(intracommunicator)とグループ間の通信用のインターコミュニケータ(intercommunicator)の2種類がある。グループは、この目的のためにプロセスの集合を定義するために用いられる。プロセスはグループ内で一意のランク(グループ内プロセッサ数がmのとき0からm-1)を有し、一つのプロセスが複数のグループのメンバになれる。ただし、簡単なプログラムではイントラコミュニケータを使いさえすればよく、グループについてのその他の概念は不要である。



(a) 意図した動作

送信バッファへのコピーは時間的不利益および記憶場所を2倍使うという問題がある。

A - 4 完了

送受信ルーチンはいくつかの種類があって、局所完了と大域完了という概念で分類される。ルーチンは、その操作の自分の部分をすべて終了したとき局所完了となり、その操作にかかわるすべてのものが自分の部分を完了したとき大域完了となる。すなわち、操作が大域完了となるためにはすべてのルーチンが局所完了でなければならない。

A - 5 ブロック型ルーチン

MPI ではブロック型ルーチン（送受信とも）は局所完了したときリターンする。ブロック型送信ルーチンの局所完了条件は、メッセージを保持していた場所が再利用できる、すなわち、送信されるメッセージに影響することなく変更できるということである。ブロック型送信はメッセージを送信してリターンするが、これはメッセージが受信されたことを意味するわけではなく、プロセスがメッセージに悪影響を与えずに先へ進めることだけを意味する。要するに発信元プロセスはデータアクセスに必要な最小時間だけブロックされる。ブロック型受信ルーチンも局所完了したときにリターンするが、この場合はメッセージが宛先に受信され、それをリードすることができることを意味する。

ブロック型送信のパラメータの一般形は以下の通りである。

MPI_Send(buf, count, datatype, dest, tag, comm)					
送信バッファ のアドレス	送信する 要素数	各要素の データ型	宛先プロセス のランク	メッセージ タグ	コミュニケータ

ブロック型受信のパラメータの一般形は次のようになる。

MPI_recv(buf, count, datatype, src, tag, comm, status)						
受信バッファ のアドレス	受信する 最大要素数	各要素の データ型	発信元プロセス のランク	メッセージ タグ	コミュニケータ	操作後 の状態

最大メッセージ長が MPI_Rrecv()内に指定されていることに注意しなければならない。最大長より大きいメッセージが受信されるとオーバーフローエラーが発生し、最大長より短ければメッセージはバッファの前方に格納されて残りは変化しない。しかし、ふつうは大きさがわかっているメッセージが送信されることを期待する。なお、MPI では発信元あるいは宛先の指定はデータの後に与えられることに注意する必要がある。

【例】整数 x をプロセス 0 から 1 へ送信する。

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank) /*find process rank */
```

```

if ( myrank == 0) {
    int x;
    MPI_Send(&x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD);
} else if (myrank == 1) {
    int x;
    MPI_Recv(&x, 1, MPI_INT, 0, msgtag, MPI_COMM_WORLD, status);
}

```

A - 6 非ブロック型ルーチン

非ブロック型ルーチンはただちにリターンする。すなわち、そのルーチンが局所完了であってもなくても次の文の実行を許す。非ブロック型送信 `MPI_Isend()` (I はただちにという意味の `immediate` を表す) は発信元内のメモリ位置の変更が安全になる前でもリターンする。非ブロック型受信 `MPI_Irecv()` は受信するメッセージがない場合でもリターンする。これらのフォーマットは

```

MPI_Isend(buf, count, datatype, dest, tag, comm, request)
MPI_Irecv(buf, count, datatype, source, tag, comm, request)

```

である。

完了は別のルーチン `MPI_Wait()` と `MPI_Test()` によって検出できる。`MPI_Wait()` はその操作が実際に完了するまで待ってからリターンし、`MPI_Test()` は操作がその時点で完了しているかどうかを示すフラグをセットしてただちにリターンする。これらのルーチンは特定の操作が完了したかどうかを知る必要があり、それは `request` パラメータをアクセスすることでわかる。非ブロック型ルーチンは通信遅延が大きいとき本質的に必要な通信と計算のオーバーラップ機能を提供する。

【例】プロセス 0 からプロセス 1 へ整数 `x` を送信し、プロセス 0 の実行を継続する。

```

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);    /* find process rank */
if (myrank == 0) {
    int x;
    MPI_Isend(&x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD, req1);
    compute();
    MPI_Wait(req1, status);
} else if (myrank == 1) {
    int x;
    MPI_Recv(&x, 1, MPI_INT, 0, msgtag, MPI_COMM_WORLD, status);
}

```

A - 7 送信モード

MPI の送信ルーチンは、送受信プロトコルを定義する。標準、バッファ付、同期、レディの4種の送信モードのいずれかをもつ。

標準モードでは、対応する受信ルーチンがスタートしていることを仮定しない。バッファがある場合のその量は実装依存で MPI には定義されていない。もしバッファされていれば送信は受信に到達する前でも完了できる。(非ブロック型なら完了は対応する MPI_Wait() または MPI_Test() がリターンしたときに発生する。)

バッファ付モードでは、送信は対応する受信の前に始まり、リターンするかもしれない。このモードではアプリケーション・プログラム内にバッファ領域を用意する必要はなく、バッファ領域は MPI ルーチン MPI_Buffer_attach() によってシステムに用意され、また、MPI_Buffer_detach() によって取り除かれる。

同期モードでは、送受信ルーチンは相手より前にスタートできるが、同時にしか完了しない。

レディモードでは、送信は対応する受信に到達してからしかスタートできず、そうでなければエラーとなる。このモードの使用では誤った操作をしないように注意しなければならない。

4種のモードはブロック型および非ブロック型の両方の送信ルーチンに用いることができる。3種の非標準モードは記憶用文字(バッファ付 -b, 同期付 -s, レディ付 -r)で区別され、たとえば、MPI_Issend() は非ブロック型同期送信ルーチンである。ブロック型も非ブロック型も受信ルーチンには標準モードしかなく、対応する送信ルーチンのスタートは仮定されない。どのタイプの送信ルーチンもどのタイプの受信ルーチンとともに使用できる。

A - 8 集合通信

1対1通信では1個の発信元と1個の宛先プロセスのみが関与するのに対して、ブロードキャストなどの集合通信ではプロセスの集合が関係する。それらのプロセスはイントラコミュニケータで定義され、メッセージタグは存在しない。

ブロードキャストおよびスカタルーチン

MPI はブロードキャストルーチンと各種のギャザーおよびスカタルーチンを提供する。コミュニケータが集合操作に関与するプロセスの集合を定義し、新しいコミュニケータ作成用に多種のルーチンが用意されている。データに対する主な集合操作は以下の通りである。

MPI_Bcast()	ルートから他の全プロセッサに一斉に同報通信
MPI_Gather()	プロセスのグループから値を収集
MPI_Scatter()	バッファをプロセスのグループに分配

MPI_Alltoall()	全プロセスから全プロセスへデータを転送
MPI_Reduce()	全プロセス上のデータを組み合わせて1個の値をつくる
MPI_Reduce_scatter()	値を組み合わせて結果を分配
MPI_Scan()	プロセス上のデータのプレフィックス簡約を計算

関与するプロセスは同じコミュニケータ内のものである。各ルーチンにはいくつかの変形がある。

【例】ルートプロセス内に動的に割り付けられたメモリを用いて、プロセス群からプロセス0にデータを集める。

```
int data[10];          /*data to be gathered from processes*/
.
.
.
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);          /*find rank */
if (myrank == 0) {
    MPI_Comm_size(MPI_COMM_WORLD, &grp_size);    /*find group size*/
    buf = (int *)malloc(grp_size*10*sizeof(int)); /*allocate memory*/
}
MPI_Gather(data, 10, MPI_INT, buf, grp_size*10, MPI_INT, 0,
           MPI_COMM_WORLD);
```

MPI_Gather()はルートを含む全プロセスからギャザーを行うことに注意せねばならない。