卒業研究論文

並列コンピュータ・ソフトウェアの性能評価

２００２年2月5日

高知工科大学

知能機械システム工学科

磯村研究室

片山　直人

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たたない。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８CPU

　　　Windows　NT マシン１６CPU

　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。

　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

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。

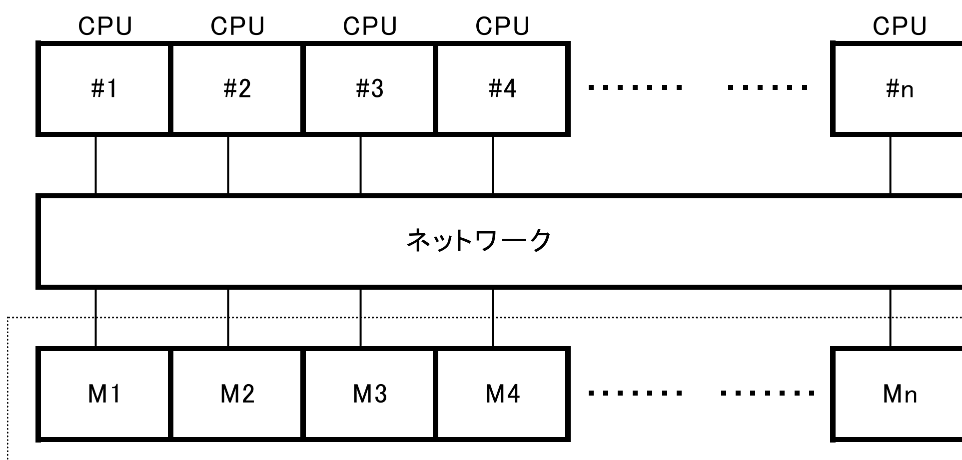
　本稿では、８台の並列 CPU を使用し、オイラーの関数すなわち素数の個数計算を例にとり、並列計算ソフトウェアの性能評価を行った結果について述べている。素数は、現在インターネット上を行き交う情報の暗号化に重要な役割を演ずる数である。

　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、８台でも可能であり、８台が可能ならば１００台でも８１９２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 CPU１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のアドレスをもち、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）

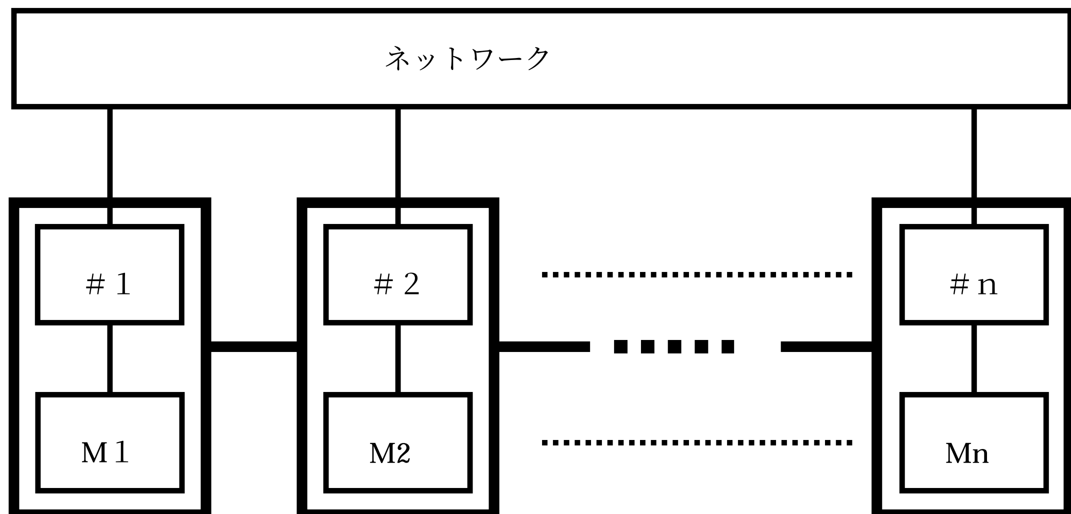


図２分散共有記憶マルチプロセッサ構成

したがって、この構成では見かけ上、各メモリM1，M2，M3,・・・Mm 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運ぶ。

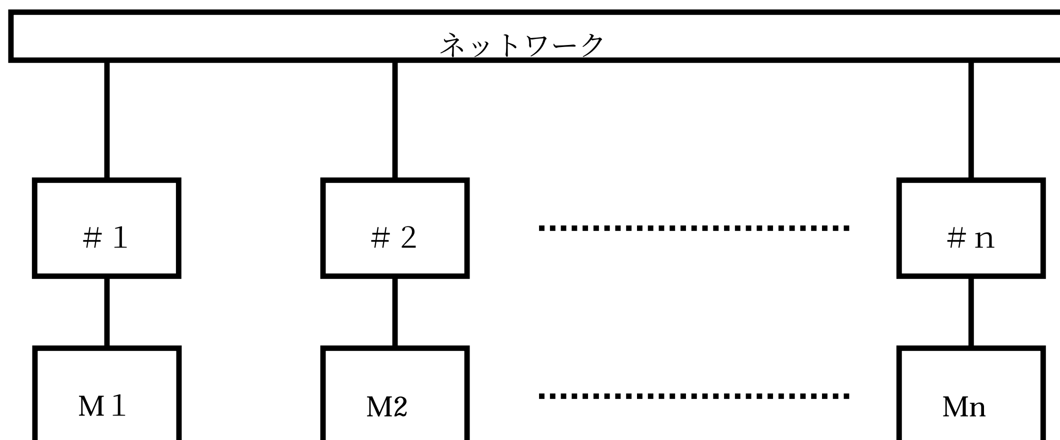


図３メッセージ通信マルチプロセッサシステム

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エア構成

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。このうち、本研究に使用したワークステーションは○印で示した。

Compaq（ES40）　並列コンピュータ（Alpha１＆ Alpha2）
５００MHｚX４CPUX２＝４GHz８CPU
２GB主メモリ X2
３６GBX2 補助メモリ

○Compaq（SP750 ＆ AP550X2）
７３３MHｚ X3

２５６MB主メモリ X3
１４０ＧＢ ＆ １８GBX2 補助メモリ

○INTERGRAPH（Visual W/SX2）
７５０MHz X2
２５６MB主メモリ X3
１０GBX2 補助メモリ

○Silicon Graphics（Visual WS ５４０）
５５０MHz
２５６MB主メモリ
８ＧＢ補助メモリ

○INTERGRAPH（W/S）　７５０MHz
２５６MB主メモリ
９GB 補助メモリ

SUN（Ultra10）　４４０MHz
５１２MB主メモリ
９GB 補助メモリ

○NEC(Express 5800)　５５０MHz
２５６MB主メモリ
１３GB 補助メモリ

○Compaq（AP550）　５５０MHz
２５６MB主メモリ
９GB 補助メモリ

LAN 上における各ワークステーションのＩＰアドレスは次の通りである。

①Alpha1	100.0.0.10
②Alpha2	100.0.0.11
③SP750	100.0.0.31
④AP550-1	100.0.0.30
⑤AP550-2	100.0.0.34
⑥Intergraph1	100.0.0.1
⑦Intergraph2	100.0.0.2
⑧Silicon540	100.0.0.33
⑨Intergraph3	100.0.0.3
⑩SunUltra10	100.0.0.20
⑪NEC	100.0.0.32
⑫AP500	100.0.0.35

３ 並列計算テストプログラム

８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＝８０００００００）までの数の中に素数が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,...，ｎ－1 のうちでｎと互いに素なものの個数をφ（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，……,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

・

・

・

min7～max7

というように８グループに分け、#0 から#7 までの CPU に配分する。

#0 CPU をマスターとし、#１～#7CPU はスレーブとする。#0 CPU は数 mini と数 maxi（I=１，2,...，7）を作り、#ｉCPU に mini と maxi を送る。受け取った#ｉCPU は mini から maxi 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#ｉCPU（ｉ=1，2,...，7）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で進めない

（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、7 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)一台での実行時間

　　もっとも遅い　Compaq　Ａｐ500　一台での実行時間は 1865 秒(約 31 分)、素数の個数は 63951 であった。

（2） 最初のトライアル計算

　n=800,000 として与えた。最初は m 個の CPU に均等配分して

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

とする。N=800,000、m=8 なので

$n_1^{(1)}=n_2^{(1)}= n_3^{(1)}=\cdots\cdots=n_8^{(1)}=100,000$

である。

　実行結果は次のようになった。

コンピュータ名	$\tau_i^{(1)}$	φ(ｎ)：素数の個数
Intergraph 1	30 sec	9592
Intergraph 2	75 sec	8392
Intergraph 3	110 sec	8013
Compaq Ap550-1	150 sec	7863
Compaq Sp750	190 sec	7678
Silicon 540	305 sec	7560
NEC	360 sec	7445
Compaq Ap500	410 sec	7408
	計	63951

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。1 台での処理時間 1865 秒に対し、410 秒は 1/4.5 にすぎない。

（3）２回目の配分

２回目の配分 $n_1^{(2)}$、$n_2^{(2)}$、 $n_3^{(2)}$、……、$n_8^{(2)}$ は次のようにして求められる。

$$(\tau_1^{(1)}/n_1^{(1)})x_1=(\tau_2^{(1)}/n_2^{(1)})x_2=\ldots=(\tau_m^{(1)}/n_m^{(1)})x_m=\tau^{(1)}$$

$$\tau^{(1)}\Sigma(n_i^{(1)}/\tau_i^{(1)})=n \qquad (i=1, 2, 3, \cdots\cdots, m)$$

$$\tau^{(1)}=n/\Sigma(n_i^{(1)}/\tau_i^{(1)}) \qquad (i=1, 2, 3, \cdots\cdots, m)$$

$$x_i=n/\Sigma(n_i^{(1)}/\tau_i^{(1)})\cdot n_i^{(1)}/\tau_i^{(1)} \qquad (i=1, 2, 3, \cdots\cdots, m)$$

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

計算結果は

コンピュータ名	$n_i^{(2)}$	$\Sigma n_i^{(2)}$
Intergraph 1	351025	351025
Intergraph 2	140410	491435
Intergraph 3	95734	587169
Compaq Ap550-1	67940	655109
Compaq Sp750	55425	710534
Silicon 540	34527	745061
NEC	29253	774314
Compaq Ap500	25686	800000

となる。このデータを入力した実行結果は次のようになった。

コンピュータ名	$\tau_i^{(2)}$	φ(n)：素数の個数
Intergraph 1	270 sec	30053
Intergraph 2	243 sec	10824
Intergraph 3	210 sec	7262
Compaq Ap550-1	190 sec	5069
Compaq Sp750	165 sec	4137
Silicon 540	140 sec	2537
NEC	123 sec	2175
Compaq Ap500	110 sec	1894
	計	63951

負荷のアンバランスはかなり改善されたが、2 倍以上の差があり、まだ十分とは言えない。計算時間も 270/1865=1/6.9 で理論値 1/8 に及ばない。

（4）３回目の配分

$$\Sigma(n_i^{(2)}/\tau_i^{(2)})=351025/270$$
$$+140410/243$$
$$+\ 95734/210$$
$$+67940/190$$

+55425/165

+34527/140

+29253/123

+25686/110=3745.235

$n_i^{(3)}=n/\Sigma(n_i^{(2)}/\tau_i^{(2)})\cdot n_i^{(2)}/\tau_i^{(2)}$ (i=1, 2, 3, ……, m)

の計算結果は次のようになる。

コンピュータ名	$n_i^{(2)}$	$\Sigma n_i^{(2)}$
Intergraph 1	277706	277706
Intergraph 2	123425	401131
Intergraph 3	97377	498508
Compaq Ap550-1	76381	574889
Compaq Sp750	71752	646641
Silicon 540	52679	699320
NEC	50801	750121
Compaq Ap500	49879	800000

このデータを入力した実行結果は次のようになった。

コンピュータ名	$\tau_i^{(3)}$	φ(ｎ)：素数の個数
Intergraph 1	175sec	24258
Intergraph 2	178sec	9683
Intergraph 3	180sec	5374
Compaq Ap550-1	170sec	5777
Compaq Sp750	178sec	5374
Silicon 540	190sec	3929
NEC	207sec	3748
Compaq Ap500	210sec	3705
	計	63951

最長と最短の処理時間の差は 210-170=40 秒に減少した。したがって、並列化により、計算速度は 1865/210≑ 9 倍に改善された。

結言

8 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８０００００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。

1．各 CPU の負荷が均一になるよう数を配分すると、計算時間は一台で実行した場合に比べ約 1/9 になった。

2．ｎ＝８０００００を等分して分担した場合、７００００１から８０００００を分担

する CPU がもっとも時間がかかり、４１０秒であった。

3．最も少ない時間を要したのは１～１０００００を分担した。CPU で３０秒で、CPU の負荷のアンバランスは１０倍を超えた。

4．再配分を行ったところ最長時間を要したのは１～３５００００を負担したCPUで計算時間は２７０秒で、アンバランスは３倍以下に減少した。

5．再々配分を行った結果、最長と最短の処理時間差は 210-170=40 秒となり、かなり均一化した。210 秒という並列処理時間は１台の CPU 処理時間 1865 秒に比べ 1/9 に相当する。

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・システムの一部を使用した。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/

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-np4
```

または、

```
prog1-np4
```

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 は通信操作のスコープ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ー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スコープ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、

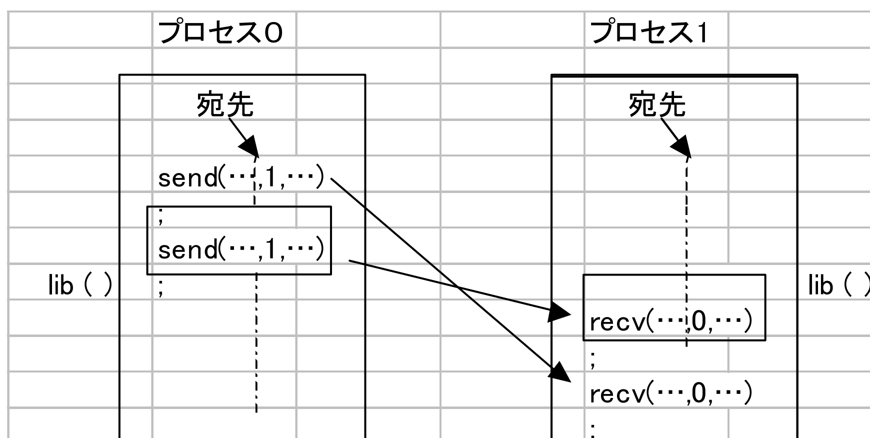
```
        if  （myrank==0）{
            int  *data;
            data = ( int *)malloc(nproc * sizeof ( int) );
          .
          .
         }
```

A－２メッセージ通信ルーチン

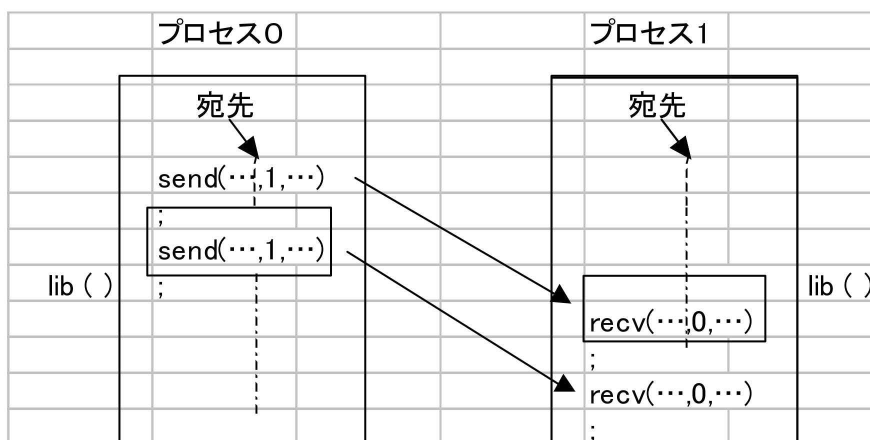
メッセージ通信は誤った操作の原因になることがあり、MPI は安全な通信環境を提供することを意図している。安全でない通信の例を図 A1 に示した。この図において、プロセス０はプロセス１にメッセージを送信しようとしているが、ライブラリルーチン間でもメッセージ通信がある。この場合、各 send/recv の組は発信元と宛先がマッチしていても誤ったメッセージ通信が発生する。ワイルドカードを用いると、正しくない操作やデッドロックはさらに起きやすくなる。一つのプロセス内の非ブロック型受信の発信元とタグが両方ともワイルドカードで、別の二つのプロセスの組がメッセージ通信を必要とするライブラリコールをしたとしよう。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および集合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



（b）ありうる動作

図 A!　ライブラリによる安全でないメッセージ通信

応用内のプロセスに対する最初のコミュニケータとしてデフォルトのイントラコミュニケータ MPI_COMM_WORLD があるので、新しいコミュニケータをつくる必要はない。簡単な応用では MPI_COMM_WORLD をすべての１対１および集合的操作に利用できる。新しいコミュニケータは既存のコミュニケータをもとにしてつくられ、既存コミュニケータからコミュニケータ（および既存グループからグループ）を形成する MPI ルーチン群が用意されている。

A－３　１対１通信

メッセージ通信はよく知られている送受信コールによって行われる。メッセージタグがあって、受信ルーチンのタグおよび発信元のかわりに (MPI_ANY_TAG と MPI_ANY_SOURCE)を使うことができる。

パラレル・バーチャル・マシン(PVM)スタイルのデータのパッキングとアンパッキングは、メッセージの発信元または宛先とともにパラメータ内に定義される MPI データ型を用いて通常避けられる。そのデータ型は基準の MPI データ型（MPI_INT, MPI_FLOAT, MPI_CHAR など）でもユーザーがつくったものでもよい。ユーザー定義のデータ型は既存のデータ型からつくられる。これによって任意の複雑さのメッセージを形成するための構造をつくりだせる。

たとえば、２個の整数と１個の浮動小数点を１メッセージで送信したければ、それを含む構造をつくることができる。パック/アンパックルーチンの必要性をなくすほかにも、データ型を宣言して用いればその再利用ができるという利点もある。また、明示的送受信バッファも必要ない。これは発信元内から明示的送信バッファへのコピーをなくすので、大きなメッセージに対する記憶要求を減少させる点で特に有用である。

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ることなく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)

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)

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_Rercv()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、最大長より短か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, staus);
}   else if (myrank == 1) {
    int x;
    MPI_Recv(&x, 0, MPI_INT, 1, magtag, 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ラ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。

ブロードキャスタおよびスキャタルーチン

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キャ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。