JAERI-Data/Code
98-030

Paragon上でのスカラー超並列プログラム開発ガイド

1998年10月

上島　豊・荒川拓也・佐々木明・横田　恒*

日本原子力研究所
Japan Atomic Energy Research Institute

本レポートは，日本原子力研究所が不定期に公刊している研究報告書です。
入手の問合わせは，日本原子力研究所研究情報部研究情報課（〒319-1195　茨城県那珂郡東海村）あて，お申し越しください。なお，このほかに財団法人原子力弘済会資料センター〒319-1195　茨城県那珂郡東海村日本原子力研究所内）で複写による実費頒布をおこなっております。

This report is issued irregularly.
Inquiries about availability of the reports should be addressed to Research Information Division, Department of Intellectual Resources, Japan Atomic Energy Research Institure, Tokai-mura, Naka-gun, Ibaraki-ken 319-1195, Japan.

編集兼発行　日本原子力研究所

JAERI－Data/Code 98－030

Paragon上でのスカラー超並列プログラム開発ガイド

日本原子力研究所関西研究所光量子科学センター
上島　豊※・荒川　拓也・佐々木　明・横田　恒*

（1998年9月11日受理）

日本で100並列を越える並列計算が、実際に行われるようになったのは、つい数年ほど前からである。日本原子力研究所（原研）のIntel製75MP834＜関西研究所＞、Paragon XP/S 15GP256 ＜那珂研究所＞は、本格的超並列計算機の先駆けとして光量子、核融合の大規模、超並列計算を目的に導入されている。これらの計算機を使って超並列計算を行うために、多くの超並列計算プログラムが移植や新規作成されている。これらのプログラムは、従来、ワークステーションやベクトル計算機上で動作していたものをそのまま移植したものか、並列用にアルゴリズムの変更を施したものである。異なる計算機及びオペレーティングシステム（OS）の基でのプログラム開発には、細心の注意とノウハウが必要であるが、Paragonに到ってはユーザ数が極めて少ないため、ノウハウの集積と環境の標準化が大変困難な状況にある。そのため、原研関西研究所におけるParagon XP/S 75MP834 上での超並列計算プログラム開発において得た情報をParagon上での超並列プログラム開発ガイドとしてまとめた。

関西研究所：〒572-0019　大阪府寝屋川市三井南町25-1
※ 博士研究員
* 財団法人 高度情報科学技術研究機構

JAERI－Data/Code 98－030

Guide to Development of a Scalar Massive Parallel Programming on Paragon

Yutaka UESHIMA*, Takuya ARAKAWA, Akira SASAKI
and Hisasi YOKOTA*

Advanced Photon Research Center
Kansai Research Establishment
Japan Atomic Energy Research Institute
Miiminami-machi, Neyagawa-Shi, Osaka

(Received September 11, 1998)

Parallel calculations using more than hundred computers had begun in Japan only several years ago. The Intel Paragon XP/S 15GP256 <Naka Fusion Research Establishment>, 75MP834<Kansai Research Establishment> were introduced as pioneers in Japan Atomic Energy Research Institute (JAERI) to pursue massive parallel simulations for advanced photon and fusion researches. Recently, large number of parallel programs have been transplanted or newly produced to perform the parallel calculations with those computers. However, these programs are developed based on software technologies for conventinal super computer, therefore they sometimes cause troubles in the massive parallel computing. In principle, when programs are developed under different computer and operating system (OS), prudent directions and knowledge are needed. However, integration of knowledge and standardization of environment are quite difficult because number of Paragon system and Paragon's users are very small in Japan. Therefore, we summarized information which was got through the process of development of a massive parallel program in the Paragon XP/S 75MP834.

Keywords: A Massive Parallel, Paragon, Guide, Standardization, Integration of Knowledge

---

* Post-doctoral Fellow
* Research Organization for Information Science & Technology

目　　次

Contents

This is a blank page.

1. はじめに

ここ４、５年前までは、最も高速な計算機といえば、いわゆるスーパーコンピュータと呼ばれるベクトル型計算機であった。この型の計算機は４、５年で計算速度が１０倍ずつ高速になってきた。しかし、現在、1CPU の演算速度が限界に達してきている。今まで使われていたベクトル計算機の 1CPU 性能は、数 GFLOPS で搭載メモリも数 Gbyte であったが、将来、格段に速い単体の計算機は出てきそうにない。これにかわって、より高速な計算機として並列計算機が次々と導入されてきている。

日本原子力研究所（原研）には、Intel 製 Paragon XP/S 75MP834 <関西研究所>[1, 2]、15GP256 <那珂研究所>が、本格的超並列計算機の先駆けとして光量子、核融合の大規模、超並列計算を目的に導入されている。これらの計算機を使って超並列計算を行うために、多くのプログラムの移植や新規作成が行われている。これらのプログラムは、従来、ワークステーションやベクトル計算機上で動作していたプログラムをそのまま移植したものか、並列用に多少のアルゴリズム変更を施したものである。一般的に、異なる計算機及びオペレーティングシステム（OS）の基でプログラムを開発するには、細心の注意とノウハウが必要であるが、従来、ワークステーションからベクトル計算機、異なるメーカの計算機へのプログラムの移植は、ユーザ数の多さが幸いして、多くのノウハウと標準化された環境のおかげで、比較的短時間で作業を完了することができていた。しかし、超並列計算機に関しては、ユーザ数が極めて少ないため、ノウハウの集積と環境の標準化が大変困難な状況にある。そのため、原研関西研究所における Paragon XP/S 75MP834 上での超並列計算プログラム開発において得た情報を Paragon 上での超並列プログラム開発の指針としてまとめた。

関西研究所で稼働中の分散メモリスカラ超並列計算機 Paragon XP/S 75MP834 は、システム全体で 120GFLOPS、100Gbyte の性能を有している。この数字は、現有の単体の最高速計算機の約 100 倍の演算速度と搭載メモリの性能を意味するものである[3, 4]。実際、この性能がどの程度のものかを理解してもらうために、シミュレーションしている系（レーザー・プラズマ相互作用）で例を挙げる。従来の単体ベクトル計算機では、典型的なシミュレーションで粒子数が数千万粒子、空間格子が 1,000×1,000 格子のシミュレーションが限界であった。しかし、この Paragon 上で並列化されたプログラムを使うと、粒子数が数十億粒子、空間格子が 1,000×100,000 格子または、10,000×10,000 格子のシミュレーションが実行できる[4]。このシステムサイズの増大がもたらす波及効果は、絶大なものである。もし、空間格子を 0.1μm（レーザー波長の 1/10）とすると、100μm（レーザー広がり方向）×10cm（レーザー伝搬方向）という、実際の実験に近いレーザーの伝播を計算することができる。また、２次元計算が困難とされていた固体（電子密度~$10^{23}/cm^3$）との相互作用では、初期温度 100eV で 5μm（レーザー広がり方向）×1μm（固体の厚さ）のレーザーの反射（高調波発生）吸収のシミュレーションも行うことができる。

2. Paragonの特徴

Paragonとは、Intel社製のi860XPプロセッサを使った分散メモリスカラー超並列計算機群の総称名であり、原研では那珂研究所（那珂研）と関西研究所（関西研）に、それぞれParagon XP/S 15GP256、 Paragon XP/S 75MP834が稼働中である。

プログラム開発に関して関西研のParagonを使用したため、以下、関西研のParagon XP/S 75MP834についてさらに詳しく述べておく。関西研のParagon XP/S 75MP834のハードウェア仕様は、 Table.3.1の通りである。

Table.3.1 Pragon XP/S-75MP834と分割後のS120MP、S5MPのハードウェア仕様

	XP/S-75MP834	S120MP	S5MP
Compute ノード名	MP ノード	MP ノード	MP ノード
Compute ノード数	834	800	34
1 ノードの演算プロセッサ数	2×i860XP	2×i860XP	2×i860XP
演算性能 1 ノード	150MFLOPS	150MFLOPS	150MFLOPS
全 ノード	125.1GFLOPS	120GFLOPS	5.1GFLOPS
Memory 1 ノード	128MB	128MB	128MB
全 ノード	106.752GB	102.4GB	4.352GB
ノード内転送速度	400MB/s	400MB/s	400MB/s
ノード間結合方式	2次元Mesh	2次元 Mesh	2次元 Mesh
ノード間転送速度	175MB/s	175MB/s	175MB/s
磁気ディスク	432GB	20GB(ufs) 240GB(pfs)	10GB(ufs) 8GB(pfs)

このParagon XP/S 75MP834は、バッチジョブ・大規模計算用のParagon S120MP（800ノード）とインタラクティブデバック・小規模計算用のParagon S5MP（34 ノード）にハードウェア的にもソフトウェア的にも分割され、ともに独立に運用されている。1 MPノードの演算性能は、150MFLOPS（= 75 MFLOPS / i860XP × 2）であり、メモリ容量は128Mbyteである。システム全体では、Paragon S120MPは、120GFLOPS、102.4Gbyte、また、 Paragon S5MPは、5.1GFLOPS、4.352Gbyteである。また、MPノード内のメモリとi860 XPプロセッサ間のデータ転送速度は、400MB/sである。MPノード間は、2次元相互接続ネットワークアーキテクチャによって結合しており、隣り合った任意のMPノード間のデータ転送速度は、送受信とも175 MB/s、双方向で350MB/sである。

磁気ディスク装置はRAID3を採用し、Paragon S120MPの磁気ディスク容量が400Gbyte、Paragon S5MPの磁気ディスク容量が32Gbyteである。また、この磁気ディ

スクは、通常のUnix File System (ufs=work) 領域の他に、並列計算機で発生する大容量データの入出力を高速に行うためのParallel File System (pfs)という並列I/Oが可能な領域を持っている。Paragonから実際に利用できる磁気ディスクシステムは、上記に述べたRAID3でありParagon S120MPにufs領域20Gbyteとpfs領域240 Gbyte、Paragon S5MPにufs領域10Gbyteとpfs領域8Gbyteに分割されている。その他にNetwork File Systemにより接続されたファイルサーバ（S-4/1000E）があり、RAID5フォーマットのufs領域210G byteを利用できる。

搭載されているソフトウェアは、Table.3.2の通りであり、2つのParagonは、UNIXに準じたParagon用マルチコンピュータ･オペレーティングシステムOSF/1により制御されている。

Table.3.2 Pragonソフトウェア仕様

ソフトウェア名	概要
Paragon OSF/1	Paragon用 UNIX OS
if77	Paragon用 Fortran 77
icc	Paragon用 ANSI C
iCC	Paragon用 C++
ihpf	Paragon用 HPF
NXライブラリ	並列化のための基本ライブラリ
BLAS 1	数値計算ライブラリ Vector-Vector
BLAS 2	数値計算ライブラリ Matrix-Vector
BLAS 3	数値計算ライブラリ Matrix- Matrix
Pro Solver Parallel FFT	数値計算ライブラリ FFT
IPD	並列化プログラムデバッカ
XIPD	X対応の並列化プログラムデバッカ
Peformance Analyzer	プログラム並列化支援性能評価ツール

また、並列計算機ではあるが、個々の計算ノードにUNIX OSが動いているのではなく、個々の計算ノードには、分散OS(Mach3.0マイクロカーネル)が搭載され、分散OSのUINXに透過的なインターフェイスを介して、サービスノードに搭載されているUNIX OSの機能を利用する構造となっている。このサービスノードは、全計算用ノードの制御も行っている。また、並列化に必要なライブラリとしてNXライブラリが標準でサポートされており、このNXライブラリをベースにしたメッセージパッシングライブラリとしてMPIが使用できる。

3 並列計算プログラムの特徴

超並列計算機へのプログラムの移植は、まだ始められて間もないため、経験の集積と環境の標準化がほとんどないが、構造化プログラミング、イベント駆動プログラムなどの最近のプログラム技術を活用することで、多くの問題を解決できると考えられる。

並列化の場合は、従来のシングルプロセッサの計算機とは異なり、プロセッサ間データの共有、同期やI/Oアクセスについて配慮が必要である。Paragonの構成はFig.4.1に示すとおりであり、計算ノードが相互にメッセージパッシングの方法で、データの授受およびI/Oを行いながらプログラムを実行する。

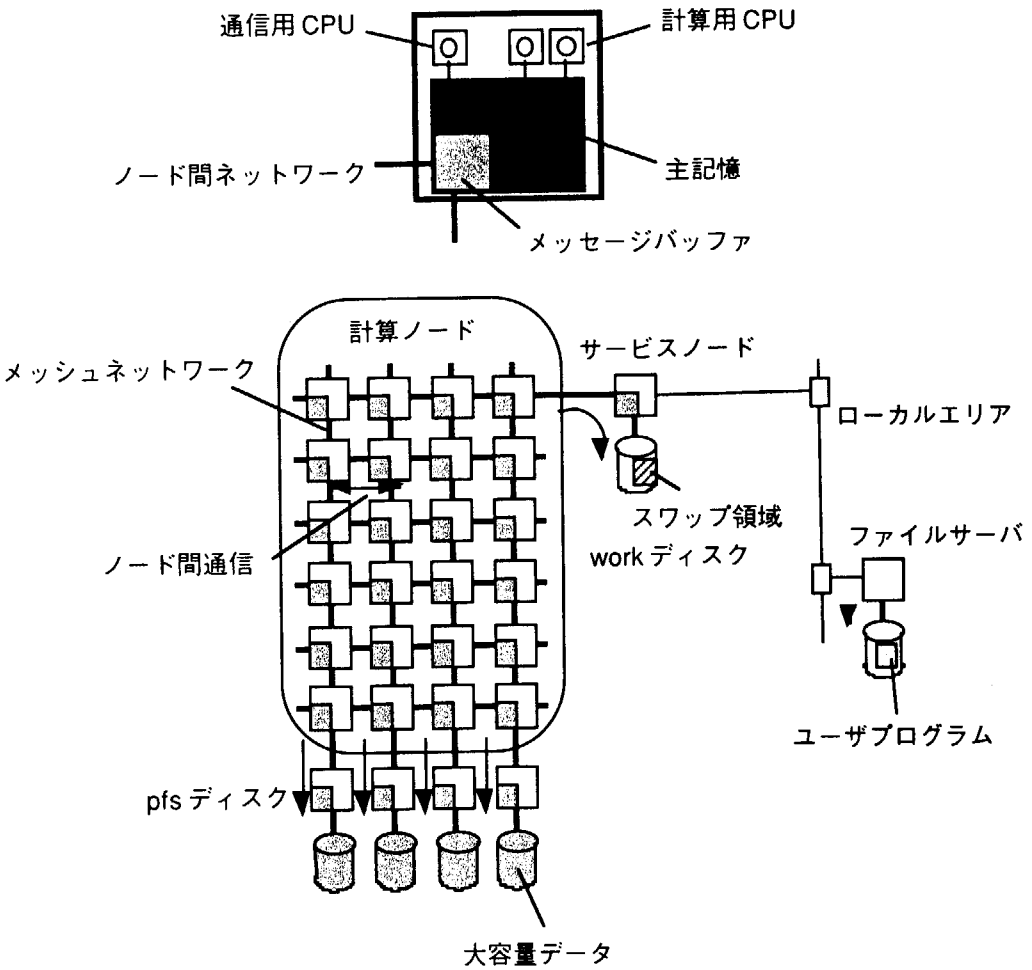


Fig.4.1 Paragonの構成

当然のことだが、入力データや計算された値ですべてのノードで必要となるものは、ノード間で通信して送信しないとデータが共有できない。しかし、この問題は、初歩的だが非常

に重要な問題で、いままでシングルプロセッサの計算機しか使ってこなかった者には、相当意識しないとうっかりと忘れてしまうことである。また、データが共有できていて各プロセッサで同じプログラムが走っていたとしても、陽にノード間の同期を取らない限り、同じ命令が同時刻に実行されることは全く保証されない。

一回の通信で転送されるデータの大きさは、メッセージバッファの大きさで制約される。メッセージバッファは、各ノード間ごとの通信用に固定で比例細分化されるので、全バッファの大きさが同じでも、各ノードのバッファ領域はノード数に反比例して減少する仕組みになっているため、大きなデータのノード間の交換や入出力の際に後述のように動作不確実となるプログラムが存在する。

I/O アクセスの問題としては、800 台の計算ノードに対し、ディスクを備えているノードは 20 数台であることから I/O は原理的に低速であることがあげられる。work 領域はサービスノードに接続されているディスクにおかれているため、これに対して集中的にアクセスが行われたとき低速となりシステム障害を起こすことがある。ファイルサーバに接続されたディスクへのアクセスはさらに低速である。

前節で書いたように、通常の ufs とは異なり、Paragon では、大容量データの入出力を高速に行うための pfs が装備され、非常に大きなデータを分割して並列に同時入出力できる。しかし、並列に入出力という概念がシングルプロセッサの計算機にはないため、その使用規約を意識して使わなければならない。

4. Paragon 上での超並列プログラム開発の指針

＜一般的注意事項＞

1) 超並列計算機は他の汎用計算機やワークステーションに比べてシステム保護機構が弱いため、プログラムを作成するときには、できるだけ“fail to safe”の精神でプログラム設計をする必要がある。

1-1) 第 2 章で説明したとおり、Paragon は、並列計算機ではあるが、個々の計算ノードに UNIX OS が動いているのではなく、個々の計算ノードには、分散 OS（Mach3.0 マイクロカーネル）が搭載され、分散 OS の UINX に透過的なインターフェイスを介して、サービスノードに搭載されている UNIX OS の機能を利用する構造となっている。超並列計算機は、その最大性能を引き出すために OS レベルでもシステム保護機構が、非常に簡略化されている。このような状況で各 CPU の動作状況を OS に監視させながらプログラムを実行するようにすると、その監視に負荷がかかりすぎて、計算機の実行性能が極端に低下してしまう。したがって、できるだけ並列計算機の性能を引き出すためには、“fail to safe”の精神でプログラム設計をする必要がある。また、現在の並列化コンパイラが汎用計算機のものと比較してまだ未熟であることも大きな原因の一つであるが、当分は、そのような成熟した並列化コンパイラは実現しそうにない。

＜プログラム実行＞

2) デバッグは、必ず Paragon S5MP で行うこと。また、このことからプログラムを作るときには、ノード数に依存しないソースプログラムを作成しなければならないことになる。もちろん、大きなノード数で生じる問題のデバッグは、この限りではないが、その場合、ラージスケールでのデバッグを開始した旨を情報システム管理課に連絡しておくことを薦める。

2-1) 上記で説明したように、並列計算機では、システム保護機構が非常に簡略化されているため、ユーザのプログラムによってシステム全体をダウンさせる可能性がある。そのため、デバッグは必ず Paragon S5MP で行う。また、並列数に依存するプログラムは、デバッグから本番の計算までの間にプログラムに変更を加える必要が生じるため、デバッグの信頼性を高くするためにもノード数に依存しないソースプログラムを作成しなければならない。さらに、ラージスケールのシミュレーションを行う場合、並列化数の変化によって生じる問題も数多くあるので、Paragon S5MP のデバッグ終了後 Paragon S120MP でもデバッグをしておく必要がある。9) も参照のこと。情報システム管理課の連絡先は、e-mail kanrika@apr.jaeri.go.jp, tel 0720-31-0732 である。連絡は緊急の場合以外、電子メールによって行うこと。

3） 大きなメモリの割り付けは、基本的には動的に行うようにし、もし割り付けられなければ、全ノードが正常にプログラムを終了するようにしておくこと。5）も参照のこと。静的に割り付けると、初期データ（静的割り付けメモリーなど）を全ノードのメモリー上にのせる為、初期データを大量に通信転送しなければならなくなるので、通信やサービスノードに過負荷がかかり、初期ロード時間が非常に長くなる。また、静的に割り付けられたメモリ（ソース、データ、static に割り付けた配列）の大きさは、**size860** コマンドであらかじめ確認しておくこと。各ノードには、128Mbyte のメモリ容量があるが、ユーザの使用するメモリ領域は、通信バッファを含めて 100Mbyte 以下にしておく必要がある。それ以上にするとシステムが不安定になりシステムダウンを引き起こす可能性がある。

3-1） 動的と静的メモリ割り付けのサンプルプログラム

メモリの動的割り付け（dynamic allocate）

サンプルプログラム memory_dynamic.f から一部抜粋

```
      program memory_dynamic
c
      real*4 a
      pointer(p,a(200000000))
      allocate(a,stat=i)
c
      deallocate(a)
      stop
      end
```

メモリの静的割り付け（dynamic static）

サンプルプログラム memory_static.f から一部抜粋

```
      program memory_static
c
      real*4 a
      common /memory/ a
      dimension a(20000000)
c
      stop
      end
```

コンパイル方法

```
% if77 -nx memory_dynamic.f -o memory_dynamic
% if77 -nx memory_static.f -o memory_static
```

3-2） **size860** コマンド

上記コンパイルによって作成したロードモジュールの静的に割り当てた大きさを **size860** コマンドで調べる。一般的には size コマンドが知られているが、 Paragon 対応として **size860** コマンドを使用した。

text：プログラムリスト自体のサイズ

data：プログラムを実行するにあたっての空きメモリー領域

bss：プログラム内で静的に割り当てたメモリー領域のサイズ

size860 の使用例

```
% size860 -f memory_dynamic
214784(.text) + 37664(.data) + 163488(.bss) = 415936

% size860 -f memory_static
214272(.text) + 37472(.data) + 80163488(.bss) = 80415232
```

ロードモジュール名	text (byte)	data (byte)	bss (byte)
memory_dynamic	214784	37664	415936
memory_static	214272	37472	80163488

3-3） メモリがとれなかった場合の状況の説明

メモリがとれなかった場合、関西研 paragon では標準エラー出力にエラーメッセージが出力される。エラーメッセージとしては以下の２つが確認されている。

```
stdio error ; Result too large
Operation not supported on socket
```

これら２つのメッセージはいずれも、計算時間が Que-class によって制限された時間を越えるか、もしくは qdel をしたときに標準エラー出力にエラーメッセージとして出力される。つまり、メモリがとれていないプログラムを Que に投入すると、関西研の paragon において、S120MP では最大時間まで、S5MP では無制限に時間を費やしてしまうことになる。このため、予め計算時間を把握しておくことが極めて重要となる。

4) 大きなファイルを作るプログラムを実行する場合は、あらかじめ入出力するディスクに十分な空きがあるかを確かめてから実行する。6)でも述べるように、プログラム中でも、ディスクの空きがなくなったとき、I/O処理のエラー値をとり正常に終了できるようにしておく。

4-1) 入出力するディスクの空きの確認方法
入出力するディスクに対してshowfsコマンドを発行し、予め容量の確認をしておく。

showfsの使用例（フラグkはkbytes単位を示す。）

```
aprspp%showfs -k /pfs16-2
Mounted on      kbytes      avail  capacity    sunits     factor
/pfs16-2      33106560   22060152    26%        65536        16
   sdirs:/home/.sdirs/vol1h
         /home/.sdirs/vol2h
         /home/.sdirs/vol4h
         /home/.sdirs/vol5h
         /home/.sdirs/vol7h
         /home/.sdirs/vol8h
         /home/.sdirs/vol10h
         /home/.sdirs/vol11h
         /home/.sdirs/vol13h
         /home/.sdirs/vol15h
         /home/.sdirs/vol16h
         /home/.sdirs/vol17h
         /home/.sdirs/vol19h
         /home/.sdirs/vol20h
         /home/.sdirs/vol22h
         /home/.sdirs/vol23h
```

＜プログラム設計＞
5) プログラム終了時は、全ノードで終了できるように終了前に同期をとる。また、エラーで終了させる場合も、全ノードで終了できるようにエラー処理後、エラー情報を全ノードに送信し、各ノードが同期後にその情報により正常に終了できるようにする。そうしないと、いくつかのノードが終了できないままずっと通信待ちになってしまう可能性がある。

5-1) 正常にプログラム終了させるプログラムの例１
ノード０番で計算上エラーを起こしてしまい、これ以上の計算が無意味である場合。

サンプルプログラム stop_normal.f

```
      program stop_normal
c
      include 'fnx.h'
      real*8 error_flg,tmp
      character*34 check_file
c
      error_flg =0.d0
      iam = mynode()
c
      call gsync()
c  error occurred at node 0
      if ( iam .eq. 0 ) then
ccc      call error_stop(13,dummy,dummy,check_file,error_flg)
ccc same role as call error_stop ccc
         error_flg = 1.d0
         write(*,*) 'error occurred'
cccccccccccccccccccccccccccccccccccc
      endif

ccc   call error_exit(error_flg)
ccc same role as call error_exit ccc
      call gsync()
      call gdsum(error_flg,1,tmp)
      if( error_flg .gt. 0.d0 ) then
         write (*,*) 'error stop'
         stop
      endif
cccccccccccccccccccccccccccccccccccc
c
      call gsync()
      stop
      end
```

基本的な考えは次の通りである。まず、エラーの有無を error_flg という変数に（有り1、無し0）で各ノードで蓄えておく。その後で、全ノードの error_flg の和をとることにより、全ノードにエラーの有無を伝え、その情報（error_flg の和）により、全ノード同期（gsync）をとり、終了するようにする。例のプログラムではノード 0 がエラーを発見した（error_flg = 1.d0）と想定し、 error_flg を gdsum により和をとることで全ノードで情報を共有し、 error_flg>0 となるので、全ノードで終了させている。プログラム内の error_stop（エラーを発見したとき、そのノードで error_flg = 1.d0 とする）と error_exit（全ノードで error_flg の和をとり、その情報からプログラムの終了判定をする）はそれらをツール化したサブルーチンであり、付録2の error.f に掲載した。

5-2）正常にプログラム終了させるプログラムの例2
あるノード（istop）に対して過ったエラー処理のあとで、ストップ文を知らずに発行していた場合。
　これは、初期の並列化プログラミングにおいて陥り易いプログラミングエラーである。その過った例を下記に示す。

```
      error_flg =0.d0
      iam = mynode()
c
      call gsync()
c
c  error occurred at node ix
      if ( iam .eq.istop ) then
         error_flg = 1.d0
         write(*,*) 'error occurred'
         stop
      endif
c
      call gsync()
```

このようなプログラミングをすると全ノードが同期待ち文 call gsync()で ix ノードの実行を待つため、先にすすめなくなる。このため、 call gsync()の手前にプログラムの実行中にストップしているノード（istop）があれば、全ノードを終了させるツール **call node_check** が必要になる。

```
      error_flg =0.d0
      iam = mynode()
c
      call gsync()
c
c  error occurred at node ix
      if ( iam .eq.istop ) then
         error_flg = 1.d0
         write(*,*) 'error occurred'
         stop
      endif
c
      call node_check(error_flg)
c
      call gsync()
```

サンプルプログラム内の**node_check**.はノード間の非同期通信命令を使い、ストップしているノードがあれば、他のノードの全てを終了させるサブルーチンであり、付録２のnode_check.f に掲載した。

6) ファイル処理には、エラー処理をいれる。ファイル処理を行ったノードでエラーが起こり、そのノードだけが終了してしまうときがある。この場合、ジョブの実行状態を端末から確かめると正常に実行を継続をしているように見えるが、実際には、1 ノードが終了し、他のノードは通信待ちになっていてプロセスは全く進んでいない。

6-1) I/O 処理のエラー値をとり正常に終了させるプログラム。
サンプルプログラム file_access.f

```
      program file_access
c
      include 'fnx.h'
      integer iam
      real*8 error_flg
      character*34 check_file
c
      call gsync()
      iam = mynode()
```

```
c
      check_file='test_file'
      call file_exist_check(check_file,error_flg)
      if (iam .eq. 0) then
        call sopen(10,check_file,error_flg)
        write (10,*) 'open test'
      endif
      close(10,status='keep')
      call error_exit(error_flg)
c
      stop
      end
```

このプログラムはファイルアクセスのうち **open** についてサブルーチン化した（**call sopen**）を使用したものである。**sopen** は、オープン時にエラーがあると error_flg=1 を代入するサブルーチンであり、付録2の sopen.f に掲載した。I/O 処理のエラー値をとり正常に終了させることは関西研 Paragon ではファイルのアクセス（open, read, write, inquire 等）にはノード数制限があるために、並列化プログラミングにおいて必要不可欠なことである。これについては詳細を 9)に記述する。ここで、サブルーチン error_exit は、前述したように全ノードで error_flg の和をとり、それが 0 より大きければ全ノード終了させる働きをしている。

7）プログラム中で UNIX コマンドをシステム文を用いて呼び出すことは極力さけること。システム文で UNIX コマンドを発行すると、これはサービスノードにおいて子プロセストシて実行されるようである。UNIX コマンドは元のプログラムと非同期で実行されるので、UNIX コマンドで行った **mkdir** 等の処理が完了強いたことを **inquire** 等の別のコマンドで確認することが必要である。また、同じ UNIX コマンドを多ノードで同時に発行することはシステムの負荷を非常に高めるので避けなければならない。例えば、システム文を用いて **mkdir** の発行の後、そのディレクトリ下にファイルを作ろうとする場合、ディレクトリがまだできていないのにファイルを作ろうとして、エラーとなる可能性がある。

7-1）call system（C 言語の関数）と i=system（Fortran のシムテム関数）の例とそれと等価なシェルの例
サンプルプログラム mkdir_system1.f

```
      program mkdir_system1
c
      include 'fnx.h'
      integer iam
c
      iam = mynode()
c
      if (iam .eq. 0 ) then
         call system('mkdir  test_dir')
         open (10, file= 'test_dir/data')
         write (10,*) 'hallo'
         close (10,status='keep')
      endif
c
      stop
      end
```

サンプルプログラム mkdir_system2.f

```
      program mkdir_system2
c
      include 'fnx.h'
      integer iam,i_system
c
      iam = mynode()
c
      if (iam .eq. 0 ) then
         i_system=system('mkdir  test_dir')
         open (10, file= 'test_dir/data')
         write (10,*) 'hallo'
         close (10,status='keep')
      endif
c
      stop
      end
```

上記サンプルプログラムではファイルのオープンの直前に system 文もしくは system 関

数を使い親ディレクトリを作成している。ファイルをオープンする前に親ディレクトリを作成すること自体は正しいことであるが、シェルの中で作成しておけば、より信頼性の向上と高速化につながる。下記にそのシェルスクリプトを示す。

```
#!/bin/csh
#  qsub and mkdir

qsub -r test                ¥
     -o ./out.log           ¥
     -e ./err.log           ¥
     -q spps4-i << EOF

if (! -d test_dir ) then
      mkdir test_dir
endif

ロードモジュール名  -plk -sz 4 -mbf 10240000

exit
EOF
```

＜ファイル処理関係＞

8)　ファイル処理する直前に必ずその親ディレクトリの存在を 1 ノードに制限した inquire 文を使って確かめ、ファイルが存在しない場合は、正常終了させるようにプログラミングする。これを怠ると、プロセスを握ったままの状態になり、qdel してもそのプロセスは解放されず、いくつかのノードが新たなプロセスを受け付けなくなり、Paragon を再起動しなければならなくなる。

8-1)　inquire 文の例。そのときの qdel の方法。

先の紹介したサンプルプログラム中にある

```
      check_file='test_file'
      call file_exist_check(check_file)
```

サブルーチン **file_exist_check** が該当する。

サンプルプログラム file_exist.f

```
      subroutine file_exist_check(check_file)
c
      include 'fnx.h'
c
      integer iam
      character*34 check_file
      logical*4 ex
c
      iam = mynode()
      if (iam .eq. 0) then
         inquire(file=check_file,exist=ex)
      endif

      if ( .not. ex ) then
         call error_stop(18,idummy,idummy,check_file.error_flg)
      end if
c
      call error_exit(error_flg)
c
      return
      end
```

このサブルーチン file_exist_check では変数名 check_file に割り当てられたファイルもしくはディレクトリの存在チェックを inquire 文で1ノードに制限して行っている。ファイルもしくはディレクトリの存在・不在の情報（ ex の値 ）を使って、プログラムを終了させている。

9） 1 ノードがファイルを開くことができる数（論理機番を割り付けられる数）は 61 個である。また、1 つのディスクに対して同時に **open**、**read**（**write**）入出力のできるノード数は、32 未満とする。32 ノード以上で同時に入出力するとシステムダウンを引き起こす可能性がある。

9-1） 論理機番を割り付けられる数とノード数の制限
論理機番を割り付けられる数は実際には 64 個であるが、そのうち標準エラー出力 0 番、標準入力 5 番、標準出力 6 番とすでに割り付けられているため、61 個である。
ノード数の制限には、2 通りある。1 つはプログラムの中で分業するような作業がある場合に計算を行うノードを制限する場合と、ファイルにアクセスするためにノードを制限する場合がある。そのどちらにおいてもエラーが起こった処理を全ノードで正しくする必要

がある。

10) 32 ノード以上で同時に入出力するためには、**gopen** という命令を使い、ファイルを開き、**cread** (**cwrite**) 、**iread** (**iwrite**) を使って入出力しなければならない。ただし、**cread** (**cwrite**) は、同期型並列入出力であり、詳細なエラー情報がとれない為、できるだけ非同期型並列入出力の **iwrite** (**iread**) を使い、入出力の確実な終了確認をとり、エラー時も正常にプログラムを終了させなければならない。これらの並列入出力は、オプション ***M_RECORD*** でストライピング（イメージは１つだが物理的には複数のディスクに分けられて格納されている)されたpfsファイルとしてpfsディスクにのみ並列入出力ができる。同様に、***M_SYNC*** でストライピングしないファイルとしてufsディスクに入出力ができる。

10-1) gopen M_record , cread +エラー処理の例。
サンプルプログラム cread_M_RECORD.f

```
      program cread_M_record
c
      include 'fnx.h'
      integer iam,nodes
      real*8 error_flg
      character*34 check_file
c
      iam = mynode()
      nodes= numnodes()
      call gsync()
      check_file='/pfs16-2/j6446/'
      call file_exist_check(check_file,error_flg)
c
      call gopen(10,check_file//'test',M_RECORD)
      if (iomode(10).ne.M_RECORD) then
        close(10)
        call error_stop(17,dummy,dummy,check_file,error_flg)
      end if
      call error_exit(error_flg)
c
      gsync()
c
```

```
      irw = (nodes+15)/16
      do  ir = 1,irw
         if ( imod(iam,irw).eq.ir-1 ) then
            call cread(10, nodes, 4)
         endif
      enddo
      close(10)
c
      stop
      end
```

上記プログラムの**gopen**のモード***M_RECORD***は、パラゴンで最も早い入出力方法である。内部的な処理として、個別のファイルポインタをもち、全ノードともノンブロッキングで同じオペレーションを同じ順番でおこなう。このプログラムの流れは次の通りである。まず、サブルーチン file_exist_check で入出力するファイルが存在するかどうか確認し、**gopen**でファイルをオープンする。次に、**iomode**によって正常に**gopen**できたかどうかを確かめ、次に**gsync**で全ノード同期をとる。最後に、より高速に並列入出力を行うためにストライプ数と同数のノードずつ**cread**を実行する。このサンプルプログラムは、**cread**を通るノードが常に16ノードに制限されていることから、16ストライピングされたpfsファイルに対して有効的な手法をとっているといえる。

10-2） gopen M_sync , cread +エラー処理の例。
サンプルプログラム cread_M_SYNC.f

```
      program cread_M_SYNC
c
      include 'fnx.h'
      integer iam,nodes
      real*8 error_flg
      character*34 check_file
c
      iam = mynode()
      nodes = numnodes()
      call gsync()
      check_file='/pfs16-2/j6446/'
      call file_exist_check(check_file, error_flg)
c
```

```
      call gopen(10,check_file//'test',M_SYNC)
      if (iomode(10).ne.M_SYNC) then
        close(10)
        call error_stop(17,dummy,dummy,check_file,error_flg)
      end if
      call error_exit(error_flg)
c
      call cread(10, nodes, 4)
      close(10)
c
      stop
      end
```

サンプルプログラムの流れは次の通りである。まず、サブルーチン file_exist_check で入出力するファイルが存在するかどうか確認し、 **gopen** でファイルをオープンする。次に、**iomode** によって正常に **gopen** できたかどうかを確かめ、 **cread** を実行する。上記プログラムの **gopen** のモード ***M_SYNC*** は、内部的な処理として、全てのノードが同じファイルポインタを持つため、並列入出力向きではないといえる。利用方法としてストライピングしないファイルとして ufs ディスクに入出力することが考えられる。ファイルにアクセスに行く順番がノード番号順であるために、容量の大きいファイルの入出力をすると時間がかかることになり、そのあいだは、ノードごとの逐次処理になるため並列化効率の劣化を招くことになる。しかし、これらのプログラムの **cread**（**cwrite**）は同期型並列入出力であり、詳細なエラー情報がとれない為、入出力の確実な終了確認ができない。そのため、入出力の確実な終了確認をとり、エラー時も正常にプログラムを終了できる、非同期型並列入出力の **iwrite**（**iread**）を使うほうがよい。

10-3） gopen M_record ,iwrite +エラー処理の例。
サンプルプログラム iwrite_M_RECORD.f

```
      program iwrite_M_record
c
      include 'fnx.h'
      integer iam,nodes,iwrite_id
      real*8 error_flg,start_time,total_time
      character*34 check_file
c
      iam = mynode()
```

```
      nodes = numnodes ()
      call gsync()
      check_file='/pfs16-2/j6446/'
      call file_exist_check(check_file,error_flg)
c
      call gopen(10,check_file//'test',M_RECORD)
      if (iomode(10).ne.M_RECORD) then
         close(10)
         call error_stop(17,dummy,dummy,check_file)
      end if
      call error_exit(error_flg)
c
      irw = (nodes +15)/16
      do  ir = 1,irw
         if ( imod(iam,irw).eq.ir-1 )   then
            iwrite_id =iwrite(10, nodes, 4)
            total_time = 0.d0
            start_time = dclock()
            do while(iodone(iwrite_id) .eq. 0)
               total_time = dclock() - start_time
               if ( total_time .gt. 1.d1 ) then
                  call error_stop(14,dble(iam),dummy,error_flg)
               endif
            enddo
         endif
         call gsync()
         call error_exit(error_flg)
      enddo
      close(10)
c
      stop
      end
```

非同期型並列入出力の **iwrite** を使用した例である。基本的には、前の同期入出力と同じだが、**iodone** で完了ステータスが取れることを利用して、**iodone** 完了ステータスの合否を判定条件にして無限ループ構造をつくり、長い時間ループを抜けられない時「エラーが起こった」と判断することにしている。

```
    if ( total_time .gt. 1.d1 ) then
      call error_stop(14,dble(iam),dummy,check_fileerror_flg)
    endif
```

の１行目で指定するべき時間（下線部）はwriteされる変数の大きさによって変わるので注意が必要である。

11） pfsファイルは、通常のufsファイルではないため非フォーマットでしかデータを保存できなく、また、常にそのデータは作成したときと同じノード数の並列プログラムでしか利用できない。さらに、2Gbyte（32bit アドレッシングの上限値）以上のファイルを作成することが可能であるが、2Gbyte以上のファイルに対しては、ほとんどの（**ls**、**mv**、**cat**などは有効であるが）従来のUNIXコマンドを受け付けなくなる。

11-1） 2Gbyte以上のファイルが判定できなかったシェルの例。

```
#!/bin/csh
if ( -f restart_file ) then
   echo "ok"
   qsub restart.bat
else
   echo "ng"
endif
exit
```

シェルスクリプトを組んでジョブのリスタート等を行う上で注意するべき点である。上記のシェルスクリプトの場合restart_fileが存在すればリスタート用のqsubを行うことになる。ここで問題なのは、restart_fileのサイズである。サイズが2Gbyte以上である場合には、判定if (-f restart_file)が出来ないためにリスタートが出来なくなってしまう。

＜通信関係＞

12） ノード間の情報の送受信は、**csend**（**crecv**）、**isend**（**irecv**）で行うが、双方向送受信時では**csend**と**crecv**の組み合わせは、システムデッドロックを引き起こす可能性があるので使用は避けなければならない。また、**isend**と**irecv**組み合わせで

は、送受信が正常に完了したかの返値を message 関数に入れて必ず確認する。**isend** と **crecv** を組み合わせることにより、システムデッドロックを引き起こさないだけでなく、終了確認も省略する簡易的な使い方ができる。

12-1) isend crecv +エラー処理の例。
サンプルプログラム tarns_check.f

```
      subroutine trans_check
     1  (trans_type,send_buf,send_count,send_node,send_ptype,
     2              recv_buf,recv_count,error_flg)
c
      include 'fnx.h'
      character*34 check_file
c
      integer trans_type,send_count,send_node,send_ptype,
     1        recv_count,trans_id,trans_sum_id,iam, nodes,counter,
     2        isend_id,irecv_id
      byte    send_buf(send_count),recv_buf(recv_count)
      real*8  wait_t_time,start_t_time,error_flg
c
      iam = mynode()
      nodes = numnodes ()
      trans_id = 0
      counter = 0
      wait_t_time = 0.0
c     if (iam.eq.0) then
c        write (*,97) trans_type
c97      format('subroutine trans_check',i5,' start')
c     endif
      call gsync()
      start_t_time = dclock()

      isend_id = isend
     1          (trans_type,send_buf,send_count,send_node,send_ptype)
c     call csend (trans_type,send_buf,send_count,send_node,send_ptype)
c
      call crecv (trans_type,recv_buf,recv_count)
      do while( trans_id .eq. 0 )
```

```
        counter = counter + 1
        trans_id = msgdone(isend_id)
        wait_t_time = dclock() - start_t_time
        if( wait_t_time .gt. 60d0 ) then
           trans_id = 1
        endif
      end do
c     write (*,98) iam,counter,wait_t_time
c     call msgwait( isend_id )
c
      call gsync()
      call gisum( trans_id , 1 , tmp )
c98   format ('iam=',i3,' counter=',i5,' wait_t_time=',1p,e15.8)
      call gsync()
      if (trans_id .ne. nodes) then
         call error_stop(20,idummy,idummy,check_file,error_flg)
      endif
      call error_exit(error_flg)
c     if (iam.eq.0) then
c        write (*,99) trans_type,trans_id
c99      format('subroutine trans_check',i5,' end trans_id=',i3)
c     endif
c
      return
      end
```

上記サブルーチンは、双方向送受信時に **csend** と **crecv** の組み合わせを用いるとシステムデッドロックを引き起こす可能性がある（付録3参照）ために **isend** と **crecv** の組み合わせにして、システムデッドロックを回避しているプログラムである。

13） 通信のバッファサイズは、実行時のオプション ***-mbf*** ???（???はバッファサイズ[byte]）で行う。一回の通信で転送されるデータの大きさは、メッセージバッファの大きさで制約される。メッセージバッファは、各ノード間ごとの通信用に固定で比例細分化（付録3参照）されるので、全バッファの大きさが同じでも、各ノードのバッファ領域はノード数に反比例して減少する仕組みになっているため、注意が必要である。

13-1） csend crecv のデッドロックプログラムと PE 数依存性。
前項12)で用いたサブルーチン trans_check において **isend** を **csend** と置き換え、

通信量をふやし、***-mbf*** ???を減らすとシステムデッドロックを引き起こす。詳細については付録3メッセージパッシングの手法に記述する。

＜コンパイル・実行時のオプション＞

14) ロードモジュールのコンパイルの最適化オプションは、Intel に相談後に既定値から変更すること。***-nx*** オプションは必ずつけること。***-nx*** オプションとは、プログラムを計算ノード上で実行させるためのもので、これがなければサービスノード上で実行され、I/O のレスポンスの悪化などを招く。特に、デバック中は***-Mbounds***（配列の領域外参照チェック）をつけておく。

15) 実行時のオプションには、***-plk*** オプションは必ずつけること。***-plk*** オプションとは、メモリ上に載らなかった情報をいったんディスクに待避させるいわゆる仮想記憶を不使用にするオプションである。このオプションを適切に作動させるためには、メモリを割り付けた直後、常に全ノード通信を行いメモリをロックする必要がある。これをしなければ、Paragon では全てのノードがディスクにつながっているわけではないので仮想記憶時に非常に大きな通信負荷が生じ、実質的に計算速度が激減する。

16) Nx ライブラリーを使用するサブルーチンやプログラムでは、**fnx.h** をインクルードしておくこと。

5. 結び

Paragon の動作不確実の原因の一つは、ディスク I/O が通信負荷を増加させることにある。各ノードがディスクを持ち、スワップ領域をローカルディスク上にとれば、通信負荷を増やさず、仮想記憶を活用することができてプログラムの柔軟性が増えるだろう。また、メッセージバッファ容量に関して、初期に指定した大きさをノード数で均等に分ける方式はシステムスケーラビリティを損なっており、より柔軟にメッセージバッファを活用できることが望まれる。さらに、FORTRAN の通信及び入出力システム関数などの関数やサブルーティンがより適切なエラーを返すことが、プログラムの信頼性を高めるために望まれる。

謝辞

本研究は、日本原子力研究所関西研究所光量子科学センターで行われたものであり、すばらしい研究の場を提供していただくとともに、研究に対する深い御理解と御支援をいただいた大野英雄所長に謝意を表します。協力体制を整備し研究を推進してくだださました有澤孝センター長に深く感謝いたします。また、光量子シミュレーション研究グループにおいて、この研究を強く御支援していただきました田島俊樹グループリーダー、井原均サブグループリーダー、那珂研究所炉心プラズマ研究部プラズマ理論研究室長岸本泰明氏に深厚なる謝意を表します。最後に、関西研駐在の情報システム管理課及びシステムエンジニア、 Intel の皆様には、いつも計算機運用支援及び情報提供などでお世話になりました。改めて、ここで厚くお礼申し上げます。

参考文献

1) 関西研究所　並列計算機　(Paragon)　利用手引き
日本原子力研究所計算科学技術推進センター

2) PARAGON TM Fortran Compiler User's Guide

3) PARAGON TM Fortran System Calls Reference Manual

4) 佐々木明 "MPI を用いた２次元流体シミュレーション" ,
JAERI-Data/Code 97-048

5) K. Tani, "*Advanced Phton Simulation*",
Journal of Plasma and Fusion Reserch 72, 935 (1996)
(光量子シミュレーション、プラズマ・核融合学会誌)

付録1　サンプルプログラム　メインルーチン

cread_M_RECORD.f	：オプション M_RECORD の同期並列入力
cread_M_SYNC.f	：オプション M_SYNC の同期並列入力
cwrite_M_RECORD.f	：オプション M_RECORD の同期並列出力
cwrite_M_SYNC.f	：オプション M_SYNC の同期並列出力
file_access.f	：入出力処理のエラー値をとり異常があった場合、正常にプログラムを終了
iwrite_M_RECORD.f	：オプション M_RECORD の非同期並列出力
memory_dynamic.f	：動的なメモリー割り付け
memory_static.f	：静的なメモリー割り付け
mkdir_system1.f	：システムコールによるディレクトリーの作成
mkdir_system2.f	：システム文によるディレクトリーの作成
stop_abnormal.f	：あるノードが予期せず終了していたときの全ノード終了
stop_normal.f	：あるノードでエラーを検知して全ノード終了
test_trans.f	：通信が正常に終了したかどうかを確認

```
c=====================================================================
c     program cread_M_record
c                    error.f is needed
c              file_exist.f is needed
c=====================================================================
      program cread_M_record
      include 'fnx.h'
      integer iam, nodes
      real*8 error_flg
      character*34 check_file
c
      iam = mynode()
      nodes = numnodes ()
      check_file='/pfs16-2/j6446/'
      call gsync()

c     checking the existence of a file
      call file_exist_check(check_file,error_flg)
c
c     confirmation of gopen at option= M_RECORD
      call gopen(10,check_file//'test',M_RECORD)
      if (iomode(10).ne.M_RECORD) then
        close(10)
        call error_stop
     &       (17,dummy,dummy,check_file,error_flg)
      end if
      call error_exit(error_flg)

c     16 spliting cread
      irw = (nodes +15)/16
      do  ir = 1,irw
         if ( imod(iam,irw).eq.ir-1 ) then
            call cread(10, nodes, 4)
         endif
      enddo
      close(10)
      stop
      end
```

```
c=====================================================================
c     progran cread_M_SYNC
c                    error.f is needed
c             file_exist.f is needed
c=====================================================================
      program cread_M_SYNC
      include 'fnx.h'
      integer iam, nodes
      real*8 error_flg
      character*34 check_file
c
      iam = mynode()
      nodes = numnodes ()
      check_file='/pfs16-2/j6446/'
      call gsync()

c     checking the existence of a file
      call file_exist_check(check_file)
c
c     confirmation of gopen at option= M_SYNC
      call gopen(10,check_file//'test',M_SYNC)
      if (iomode(10).ne.M_SYNC) then
        close(10)
        call error_stop
     &       (17,dummy,dummy,check_file, error_flg)
      end if
      call error_exit(error_flg)
c
      call cread(10, nodes, 4)
      close(10)
c
      stop
      end
```

```
c=====================================================================
c     program cwrite_M_record
c                    error.f is needed
c             file_exist.f is needed
c=====================================================================
      program cwrite_M_record
      include 'fnx.h'
      integer iam, nodes
      real*8 error_flg
      character*34 check_file
c
      iam = mynode()
      nodes = numnodes ()
      check_file='/pfs16-2/j6446/'
      call gsync()

c     checking the existence of a file
      call file_exist_check(check_file)
c
c     confirmation of gopen at option= M_RECORD
      call gopen(10,check_file//'test',M_RECORD)
      if (iomode(10).ne.M_RECORD) then
        close(10)
        call error_stop
     &       (17,dummy,dummy,check_file, error_flg)
      end if
      call error_exit(error_flg)
c
c     16 spliting cwrite
      irw = (nodes +15)/16
      do  ir = 1,irw
         if ( imod(iam,irw) .eq. ir-1 ) then
            call cwrite(10, nodes, 4)
         endif
      enddo
      close(10)
      stop
      end
```

```
c=====================================================================
c     program cwrite_M_SYNC
c                    error.f is needed
c              file_exist.f is needed
c=====================================================================
      program cwrite_M_SYNC
      include 'fnx.h'
      integer iam, nodes
      real*8 error_flg
      common /error$/ error_flg
      character*34 check_file
c
      iam = mynode()
      nodes = numnodes ()
      check_file='/pfs16-2/j6446/'
      call gsync()

c     checking the existence of a file
      call file_exist_check(check_file)
c
c     confirmation of gopen at option= M_SYNC
      call gopen(10,check_file//'test',M_SYNC)
      if (iomode(10).ne.M_SYNC) then
        close(10)
        call error_stop
     &       (17,dummy,dummy,check_file, error_flg)
      end if
      call error_exit(error_flg)
c
      call cwrite(10, nodes, 4)
      close(10)
c
      stop
      end
```

```
c=====================================================================
c     program file_access
c                 error.f is needed
c            file_exist.f is needed
c                 sopen.f is needed
c=====================================================================
      program file_access
      include 'fnx.h'
      integer iam
      real*8 error_flg
      character*34 check_file
c
      iam = mynode()
c
      check_file='test_file'
      call gsync()

c     checking the existence of a file
      call file_exist_check(check_file)
      if (iam .eq. 0) then
         call sopen(10, check_file,error_flg )
         write (10,*) 'open test'
      endif
      close(10,status='keep')
      call error_exit(error_flg)
c
      stop
      end
```

```
c=====================================================================
c     program iwrite_M_record
c                 error.f is needed
c            file_exist.f is needed
c                  wait.f is needed
c=====================================================================
      program iwrite_M_record
      include 'fnx.h'
      integer iam, nodes,iwrite_id
      real*8 error_flg,start_time,total_time
      character*34 check_file

c
      iam = mynode()
      nodes = numnodes ()
      check_file='/pfs16-2/j6446/'
      call gsync()

c     checking the existence of a file
      call file_exist_check(check_file)
c
c     confirmation of gopen at option= M_RECORD
      call gopen(10,check_file//'test',M_RECORD)
      if (iomode(10).ne.M_RECORD) then
         close(10)
         call error_stop
     &       (17,dummy,dummy,check_file, error_flg)
      end if
      call error_exit(error_flg)
c
c     confirmation of 16 spliting iwrite
      irw = (nodes +15)/16
      do  ir = 1,irw
         if ( imod(iam,irw).eq.ir-1 )   then
            iwrite_id =iwrite(10, nodes, 4)
            total_time = 0.d0
            start_time = dclock()
```

```
          do while(iodone(iwrite_id) .eq. 0)
             total_time = dclock() - start_time
             if ( total_time .gt. 1.d1 ) then
                call error_stop
     &         (14,dble(iam),dummy,check_file,error_flg)
             endif
          enddo
       endif
       call gsync()
       call error_exit(error_flg)
    enddo
    close(10)
c
    stop
    end
```

```
c=====================================================================
c     program memory_dynamic
c=====================================================================

      program memory_dynamic
      include 'fnx.h'
      real*4 a

      pointer(p,a(200000000))
      allocate(a,stat=i)
c
      iam = mynode()
c
      a(1) = 1.0d0
      a(123456789) = 2.0d0
c
      write (*,*) iam,i,a(1),a(123456789)
100    format('iam=',i2,' stat=',i1,2f15.8)
c
      call gopen(51,'/work01/j6446/data01',M_SYNC)
      call cwrite (51,real(iam),4)
      close (51,status='keep')
c
      deallocate(a)
      stop
      end
```

```
c=======================================================================
c     program memory_static
c=======================================================================

      program memory_static
      real*4 a
      include 'fnx.h'
      common /memory/ a
      dimension a(20000000)
c
      iam = mynode()
c
      a(1) = 1.0do
      a(123456789) = 2.0do
c
      write (*,*) iam,i,a(1),a(123456789)
100    format('iam=',i2,' stat=',i1,2f15.8)
c
      call gopen(51,'/work01/j6446/data01',M_SYNC)
      call cwrite (51,real(iam),4)
      close (51,status='keep')
c
      call gsync()

      stop
      end
```

```
c=====================================================================
c     program mkdir_system1
c=====================================================================

      program mkdir_system1
      include 'fnx.h'
      integer iam
c
      iam = mynode()
c
      if (iam .eq. 0 ) then
         call system('mkdir test_dir')
         open (10, file= 'test_dir/data')
         write (10,*) 'hallo'
         close (10,status='keep')
      endif
c
      stop
      end
```

```
c=====================================================================
c     program mkdir_system2
c=====================================================================

      program mkdir_system2
      include 'fnx.h'
      integer iam,i_system
c
      iam = mynode()
c
      if (iam .eq. 0 ) then
         i_system=system('mkdir test_dir')
         open (10, file= 'test_dir/data')
         write (10,*) 'hallo'
         close (10,status='keep')
      endif
c
      stop
      end
```

```
c=====================================================================
c     program stop_abnormal
c                 node_check.f is needed
c                      error.f is needed
c=====================================================================
      program stop_abnormal
      include 'fnx.h'
      real*8 error_flg
c
      iam = mynode()
c
c     node 1 stop
      if ( iam .eq. 1 ) then
         write (*,*) 'iam=',iam,' is stop'
         stop
      endif
c
c     checking of stopping nodes
      call node_check(error_flg)
c
      stop
      end
```

```
c=====================================================================
c     program stop_normal
c                         error.f is needed
c=====================================================================
      program stop_normal
      include 'fnx.h'
      real*8 error_flg,tmp
      character*34 check_file
c
      error_flg =0.d0
      iam = mynode()
c
      call gsync()
c
c     error occurred at node 0
      if ( iam .eq. 0 ) then
         error_flg = 1.d0
         write(*,*) 'error occurred'
cccc        call error_stop
cccc &            (13,dummy,dummy, check_file, error_flg)
      endif
c
c     transference of an error information and all node stop
      call gsync()
      call gdsum(error_flg,1,tmp)
      if( error_flg .gt. 0.d0 ) then
         write (*,*) 'error stop'
         stop
      endif
c
cccc     call error_exit(error_flg)
c
      call gsync()
      stop
      end
```

```
c=====================================================================
c     program test_trans
c                 trans_check.f is needed
c                       error.f is needed
c=====================================================================
      program test_trans
      include 'fnx.h'
c
      integer is,iam, nodes,nid,isend_id,n_time
      real*8 a,b,error_flg,tmp
      character*34 check_file
      dimension a(150000),b(150000)
c
      do i= 1,150000
         a(i) = 0.0
         b(i) = 0.0
      enddo
c
      a(1)=1.0
c
      do i = 2, 150000
         a(i) = i + a(i-1)
      enddo
c
      call gsync()
      iam = mynode()
      nodes = numnodes ()
c
      do i= 1,150000
         a(i) = iam * a(i)
      enddo
c
c     initial data at node 0
      if (iam.eq.0) then
         write (6,10) (i,a(i),i=1,10)
         write (6,10) (i,a(i),i=14991,15000)
      endif
```

```
c
      call gsync()
c
c...... set the "network ID"
      if ( iam .eq. nodes -1 ) then
        nid = 0
      else
        nid = iam + 1
      endif
c
c...... send & recieve the buffer data
c     (nodes - 1) times transference to next node
      n_time = nodes - 1
      do i=1,n_time
        if (iam.eq.0) then
          write (*,*) 'loop = ',i
        endif
        call  trans_check
     1    (777,a,8*150000,nid,0,
     2         b,8*150000,
     3     check_file,error_flg )
      enddo
c
c     data after transference at node (nodes - 1)
      if (iam.eq.nodes) then
        write (6,20) (i,is,isend_id,b(i),i=1,10)
        write (6,10) (i,b(i),i=14991,15000)
      endif
10    format (i2,1x,1p,e15.8)
20    format (3i2,1x,1p,e15.8)
c
      call gsync()

      stop
      end
```

付録2　サンプルプログラム　サブルーチン

error.f	：エラーフラグへの値代入とエラーメッセージ出力
error_exit.f	：エラーフラグの全ノード和と全ノード終了
file_exist.f	：ファイルの存在確認
node_check.f	：ノードの動作-非動作の確認
sopen.f	：エラー処理付きファイルオープン
trans_check.f	：通信完了確認
wait.f	：時間待ち

```
c=====================================================================
c       subroutine error_stop
c             checking errors
c=====================================================================
      subroutine error_stop(no_err,prm1,prm2,err_file, error_flg)
c
      integer no_err
      real*8 prm1,prm2
      character*34 err_file
c
      error_flg = 1.d0
      if( no_err .eq. 13 )   then
         write(*,*)
     ¥   '### error 13 :,' error'
      end if
      if( no_err .eq. 14 )   then
         write(*,*)
     ¥   '### error 14 : iam=',prm1,' iwrite error'
      end if
      if( no_err .eq. 16 )   then
         write(*,*)
     ¥   '### error 16 : ',err_file,' cannot open'
      end if
      if( no_err .eq. 17 )   then
         write(*,*)
     ¥   '### error 17 : ',err_file,' is iomode error'
      end if
      if( no_err .eq. 18 )   then
         write(*,*)
     ¥   '### error 18 : ',err_file,' is none'
      end if
      if( no_err .eq. 20 )   then
         write(*,*)
     ¥   '### error 20 : ', ' isend-crecv error'
      end if

      return
      end
```

```
c=====================================================================
c       subroutine error_exit
c         program stop if error occured
c=====================================================================
      subroutine error_exit(error_flg)
c
      include 'fnx.h'
c
      call gdsum(error_flg,1,tmp)
      if( error_flg .gt. 0.d0 ) then
         if (iam .eq. 0) then
            open  (99,file='err.out', status='unknown')
            write (99,*) 'stopped by error_exit'
            close (99)
         endif
         call gsync()
         stop
      endif
c
      return
      end
```

```
c=====================================================================
c     subroutine file_exist_check
c                 error.f is needed
c=====================================================================
      subroutine file_exist_check(check_file,error_flg)
c
      include 'fnx.h'
c
      integer iam, nodes
      character*34 check_file
      logical*4 ex
c
      iam = mynode()
      if (iam .eq. 0) then
         inquire(file=check_file,exist=ex)
      endif
c
      if ( .not. ex ) then
         call error_stop
     &        (18,idummy,idummy,check_file,error_flg)
      end if
c
      call error_exit(error_flg)
c
      return
      end
```

```
c=====================================================================
c     subroutine node_check
c                    wait.f is needed
c=====================================================================
      subroutine node_check
      include 'fnx.h'
      integer iam,isend_id,irecv_id,check_ node
     1      , trans_send_id(0:511),trans_recv_id(0:511)
      real*8  total_time,start_time
     1      , wait_t_time , start_t_time
c
      iam = mynode()
      node = numnodes ()
c
      call wait_time(1.d0)
c
      total_time = 0.d0
      start_time = dclock()
c
      do check_node = 0, nodes-1
         wait_t_time = 0.d0
         start_t_time = dclock()
         trans_send_id(check_node) = 0
         trans_recv_id(check_node) = 0
c
         if (iam .eq. check_node) then
            isend_id = isend(check_node,total_time,8,-1,0)
            do while(trans_send_id(check_node) .eq. 0)
               trans_send_id(check_node) = msgdone(isend_id)
               wait_t_time = dclock() - start_t_time
               if ( wait_t_time .gt. 5.d-1 ) then
                  total_time = dclock() - start_time
                  write (*,*) 'iam=',iam
     1            ,' is stoped by isend error',total_time
                  stop
               endif
            enddo
```

```
      else
         irecv_id = irecv(check_node,total_time,8)
         do while(trans_recv_id(check_node) .eq. 0)
            trans_recv_id(check_node) = msgdone(irecv_id)
            wait_t_time = dclock() - start_t_time
            if ( wait_t_time .gt. 5.d-1 ) then
               total_time = dclock() - start_time
               write (*,*) 'iam=',iam
     1         ,' is stoped by irecv error',total_time
               stop
            endif
         enddo
      endif
      call wait_time(5.d-1)
   enddo
c
   total_time = dclock() - start_time
   write (*,*) 'iam=',iam,' is not stoped by node_check'
  1                ,total_time
c
   return
   end
```

```
c=====================================================================
c     subroutine sopen
c                 error.f is needed
c=====================================================================
      subroutine sopen(logical_unit,file_name,error_flg)

      character*34 file_name
c
      open ( logical_unit, file=file_name, err=100 )
c
      return
100    call error_stop
     &      (16,idummy,idummy,file_name,error_flg)
c
      return
      end
```

```
c=====================================================================
c     subroutine trans_check
c                 error.f is needed
c=====================================================================
      subroutine trans_check
     1  (trans_type,send_buf,send_count,send_node,send_ptype,
     2              recv_buf,recv_count,
     3   check_file,error_flg)
c
      include 'fnx.h'
      character*34 check_file
      real*8 error_flg
c
      integer trans_type,send_count,send_node,send_ptype,
     1        recv_count,trans_id,trans_sum_id,iam, node,counter,
     2        isend_id,irecv_id
      byte    send_buf(send_count),recv_buf(recv_count)
      real*8  wait_t_time,start_t_time
c
      iam = mynode()
      nodes = numnodes ()
      trans_id = 0
      counter = 0
      wait_t_time = 0.0
c     if (iam.eq.0) then
c        write (*,97) trans_type
c97      format('subroutine trans_check',i5,' start')
c     endif
      call gsync()
      start_t_time = dclock()

      isend_id = isend
     1          (trans_type,send_buf,send_count,send_node,send_ptype)
c     call csend (trans_type,send_buf,send_count,send_node,send_ptype)
c
      call crecv (trans_type,recv_buf,recv_count)
      do while( trans_id .eq. 0 )
```

```
        counter = counter + 1
        trans_id = msgdone(isend_id)
        wait_t_time = dclock() - start_t_time
        if( wait_t_time .gt. 60d0 ) then
           trans_id = 1
        endif
      end do
c     write (*,98) iam,counter,wait_t_time
c     call msgwait( isend_id )
c
      call gsync()
      call gisum( trans_id , 1 , tmp )
c98   format ('iam=',i3,' counter=',i5,' wait_t_time=',1p,e15.8)
      call gsync()
      if (trans_id .ne. nodes) then
         call error_stop(20,idummy,idummy,check_file,error_flg)
      endif
      call error_exit(error_flg)
c     if (iam.eq.0) then
c        write (*,99) trans_type,trans_id
c99      format('subroutine trans_check',i5,' end trans_id=',i3)
c     endif
c
      return
      end
```

```
c=====================================================================
c     subroutine wait_time
c=====================================================================
c
      subroutine wait_time(w_time)
      include 'fnx.h'
      real*8  w_time,s_time,t_time
      logical*4  wait_flg
c
c     wait_flg = .ture.
      s_time = dclock()
      do while ( wait_flg )
         t_time = dclock() - s_time
         if ( t_time .gt. w_time ) then
            wait_flg = .false.
         endif
      enddo
c
      return
      end
```

付録3 メッセージパッシングの手法

Intel paragonでは、各々のプロセッサ（ノード）がメッセージパッシングとよばれる手法でデータを相互に交換することによって処理が進められる。メッセージパッシングの基本は、ノード間の一対一のメッセージの送受信である。ブロードキャストや入出力もこの組み合わせで実現されている。後者は、I/Oノードへのメッセージの送受信によって行われる。送受信は、送り側ノードと受け側ノードで一対の関数、またはサブルーチンを実行することによって行われる。

Intel paragonのnxライブラリで通常用いられるメッセージの送受信の手続きには、ブロッキング型通信のcsend、crecvサブルーチンとノンブロッキング型のisend、irecv関数の２系統があり、高速かつ信頼性の高い通信を行うには、これらの特性を良く理解することが重要である。

メッセージの送受信とは、送り側ノードのプログラム中の変数や配列の形でメモリに保持されているデータを、受け側ノードのメモリへ転送する手順である。（Fig.A3.1）のように送り側ノードをノード０、受け側ノードをノード１としてその間のcsend、crecvによる通信を考える。まず、送り側ノードでcsendを発行すると、送信データを保持する変数や配列の内容が、オペレーティングシステムが通信用に確保しているバッファ（メッセージバッファ）にコピーされる。送り側ノードはこのデータを約1kBのパケットに分けてネットワークを通して受け側ノードに向けて送信する。受け側ノードはネットワーク上のデータを受け取りメッセージバッファに格納し、完了した時点でそのことを知らせる完了ステータスを送り側ノードに返す。完了ステータスを受けるとcsendは通信が正常に終了したものと判断して終了するので、プログラムは次の処理に移る。受け側ノードのcrecvは、メッセージバッファの内容を変数や配列にコピーし、完了ステータスをcrecvに返す。これによりcrecvは通信が正常に終了したと判断して終了し、プログラムは次の処理に移る。

以上は、通信するデータのサイズがメッセージバッファのサイズよりも小さい場合である。より大きなデータの通信を行う場合の一般的な手順を（Fig.A3.2）に示す。メッセージバッファのサイズは、全体の大きさが決められたサイズをプログラムに属するノードひとつひとつに均等に割り付けるという方式で決められる。デフォルトのメッセージバッファのサイズは1MBであり、プログラムが10ノードで実行される時、ある一つの相手ノードとの通信のために割り当てられるサイズは100kB、100ノードの場合は10kBとなる。

データのサイズがメッセージバッファのサイズよりも大きい場合、送信用の変数や配列の内容は、メッセージバッファのサイズごとに分割されてコピーされる。データが受け側データのメッセージバッファに格納されてそれに対する完了ステータスが返されると、送信データの次の部分がメッセージバッファにコピーされる。受け側ノードでcrecvが発行されていれば、メッセージバッファに受信されたデータはすぐに受信用の変数や配列にコピーされ、受け側ノードは次のデータをメッセージバッファに受信することができる。しかし、crecvが発行されていなかったり、何らかの原因で受信したデータをすぐに処理出来ない場合、新しいデータを受信することは不可能である。完了ステータスが得られなかった場合、送り側ノードは通信のリトライを繰り返す。このようにして最後のメッセージバッファの内容が受け側ノードに転送された時点で、csendは通信が正常に終了したものと判断して終了し、プログラムは次の処理に移る。

ここに、プログラムを書く上で間違いの原因となる問題がある。偏微分方程式を領域分割法で

解く場合には、タイムステップ毎に隣り合うノードがデータを交換しながら処理を進めることが多い。この時、

```
ノード0のプログラム                    ノード1のプログラム
...                                    ...
csend(type, A, count, 1, ptype)        csend(type, A, count, 0, ptype)
crecv(type, B, count)                  crecv(type, B, count)
...                                    ...
```

のようなプログラムは、原理的にハングアップする可能性がある。その理由は、フローチャートを（Fig.A3.3）に示すように、各々のノードでcsendが完了しない限りcrecvは発行されず、crecvが発行されない限りcsendが完了しないからである。ただし、通信量がメッセージバッファのサイズよりも小さい時は、crecvが発行されていなくてもデータは全て受け側のメッセージバッファにコピーされてcsendは完了してしまう。このプログラムは、転送データの量が十分小さい時は動作するが、メッセージバッファのサイズはノード数に反比例して減少することを考えると、少ないノード数では動作してもノード数を増加したときにデッドロックを発生すると考えられる。

この問題は、（Fig.A3.4）に示すように、通信の開始と完了の確認をわければ解決する。nxライブラリではcsendとcrecvの代わりにisend、irecv 関数とmsgdone関数を用いて実現することができる。この場合、isendの完了を待たずにirecvを発行することができるから、ノード0はノード1へデータを送りながらノード1からのデータを受け取ることができ、デッドロックなしにデータをやり取りできる。送信用配列、変数の内容をメッセージバッファにコピーする作業、送り側ノードのメッセージバッファの内容を受け側ノードに転送する作業、受信メッセージバッファの内容を受信用配列、変数にコピーする作業は、プログラム本体とは関係無く（非同期に）実行される。なお、isend、irecv関数が発行されてから対応するmsgdone関数が完了するまでの間、送受信用配列、変数のデータは不定となり、この値を変更したり参照したりしてはいけない。

isend、irecv、msgdone関数を用いる方式では、通信の完了を判断する部分をプログラムに付け加えなければならずプログラムが複雑になる問題がある。同じ処理は、（Fig.A3.5）に示すようにisendとcrecvの組み合わせで実現することができる。（Fig.A3.4）の例では、受信が完了した場合には送信は完了しているので、両方の確認を行うことは冗長である。そこで、isendを発行したあとでirecvとmsgdoneの一組を連続して発行するとすれば、これはcrecvを発行したのと同じことであるから、crecvで置き換えることができる。この方法により、メッセージバッファのサイズとデータのサイズに依存せずデッドロックすることなく通信を行うことができる。

```
ノード0のプログラムの実行位置                ノード1のプログラムの実行位置
...                                          ...
do i=1,imax                                  do i=1,imax
  ... 処理 ...                                 ... 処理 ...
                          (ループi+1番目)
  isend(type, A, count, 1, ptype) ←            isend(type, A, count, 0, ptype)
  crecv(type, B, count)                        crecv(type, B, count) ← (i番目)
end do                                       end do
```

しかし、もしこの処理が頻繁に繰り返される時は、問題が起こることがある。ノード0とノード1のプログラムの実行の順序は陽に同期を取らない限り保障されないので、ノード0でのプログラムの方がより早く進行して、上の例のようにノード0はすでにi+1番目のループの処理を実行してisend(type, A, count, 1, ptype)を実行しようとするのに、ノード1ではi番目のループのcrecv(type, B, count)が完了していない状態が起こるかも知れない。ノード0のループがi+1番目へ進むには、ノード1のisend(type, A, count, 0, ptype)とノード0のcrecv(type, B, count)が完了していれば良いからである。一方、これはノード0のi番目のループのisend(type, A, count, 1, ptype)が完了していないことを意味するので、さらにノード0でisend(type, A, count, 1, ptype)を発行しようとすると通信用配列、変数のデータが破壊され、プログラムの動作は異常になる。

isend、irecv、msgdoneの組み合わせを用いて通信すればこの問題は起こらない。また、ループを進める度にノード間の同期を取れることによっても回避できる。ノード0（1）が発行するisendのメッセージタイプをループの回数毎に変えれることでも回避できるが、極端に多くのisendが発行された場合の動作は、メッセージタイプが重複する可能性があることや、一度にシステムが保持できる通信要求の数に制限があるために確実とは言えない。

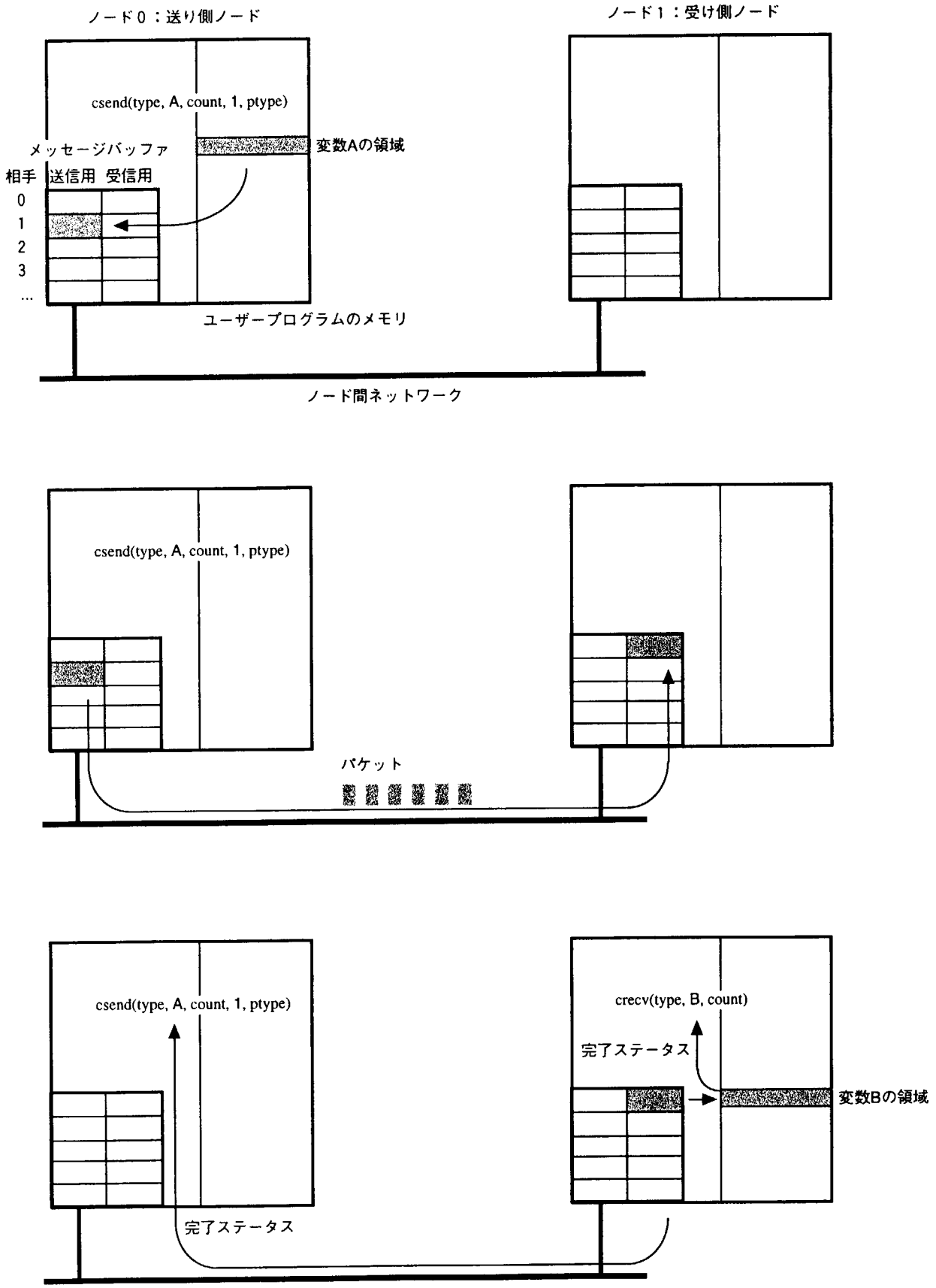
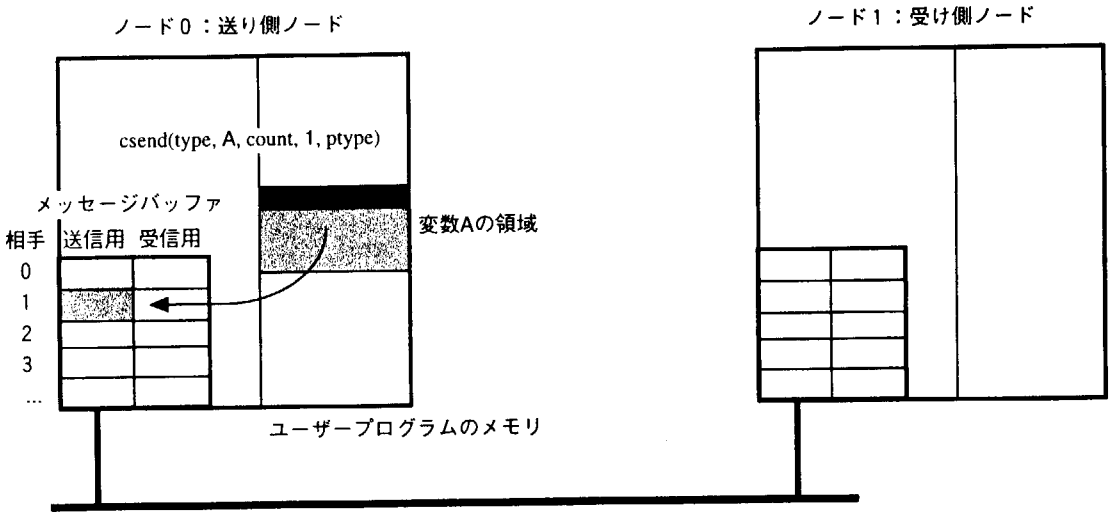
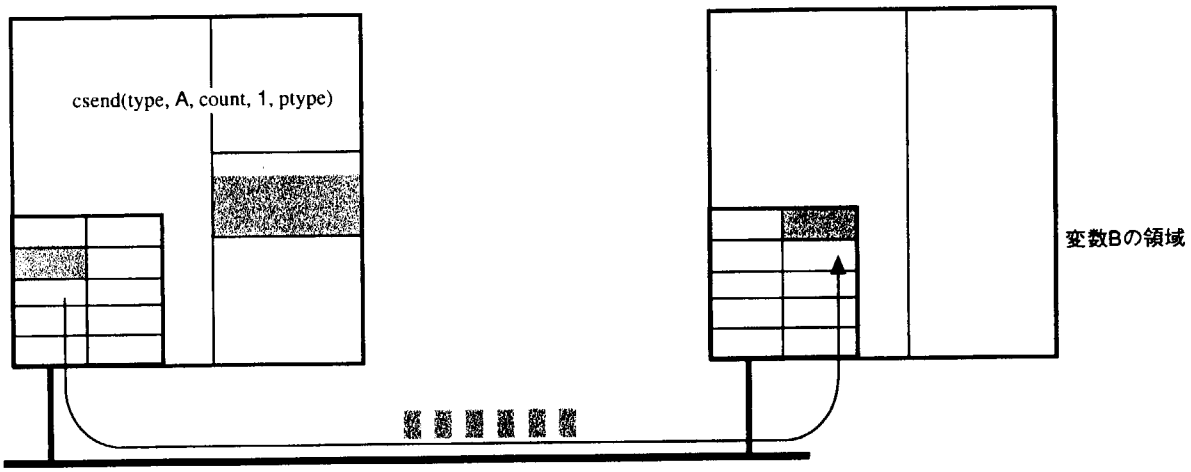
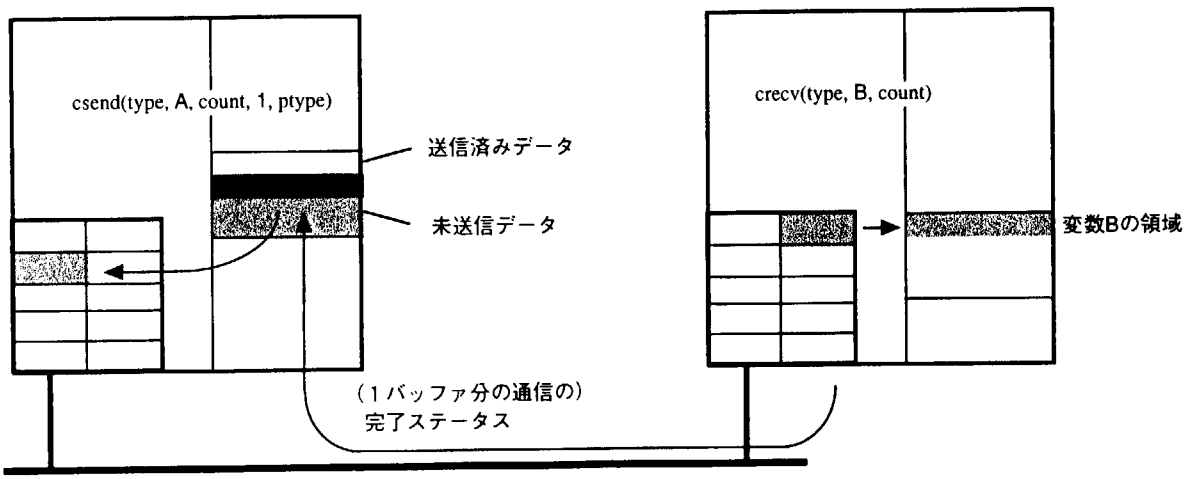


Fig.A3.1 ノード間のデータの送受信


ノード0：送り側ノード
ノード1：受け側ノード
csend(type, A, count, 1, ptype)
メッセージバッファ
相手 送信用 受信用
0
1
2
3
…
変数Aの領域
ユーザープログラムのメモリ
ノード間ネットワーク


csend(type, A, count, 1, ptype)
変数Bの領域


csend(type, A, count, 1, ptype)
crecv(type, B, count)
送信済みデータ
未送信データ
変数Bの領域
（1バッファ分の通信の）
完了ステータス

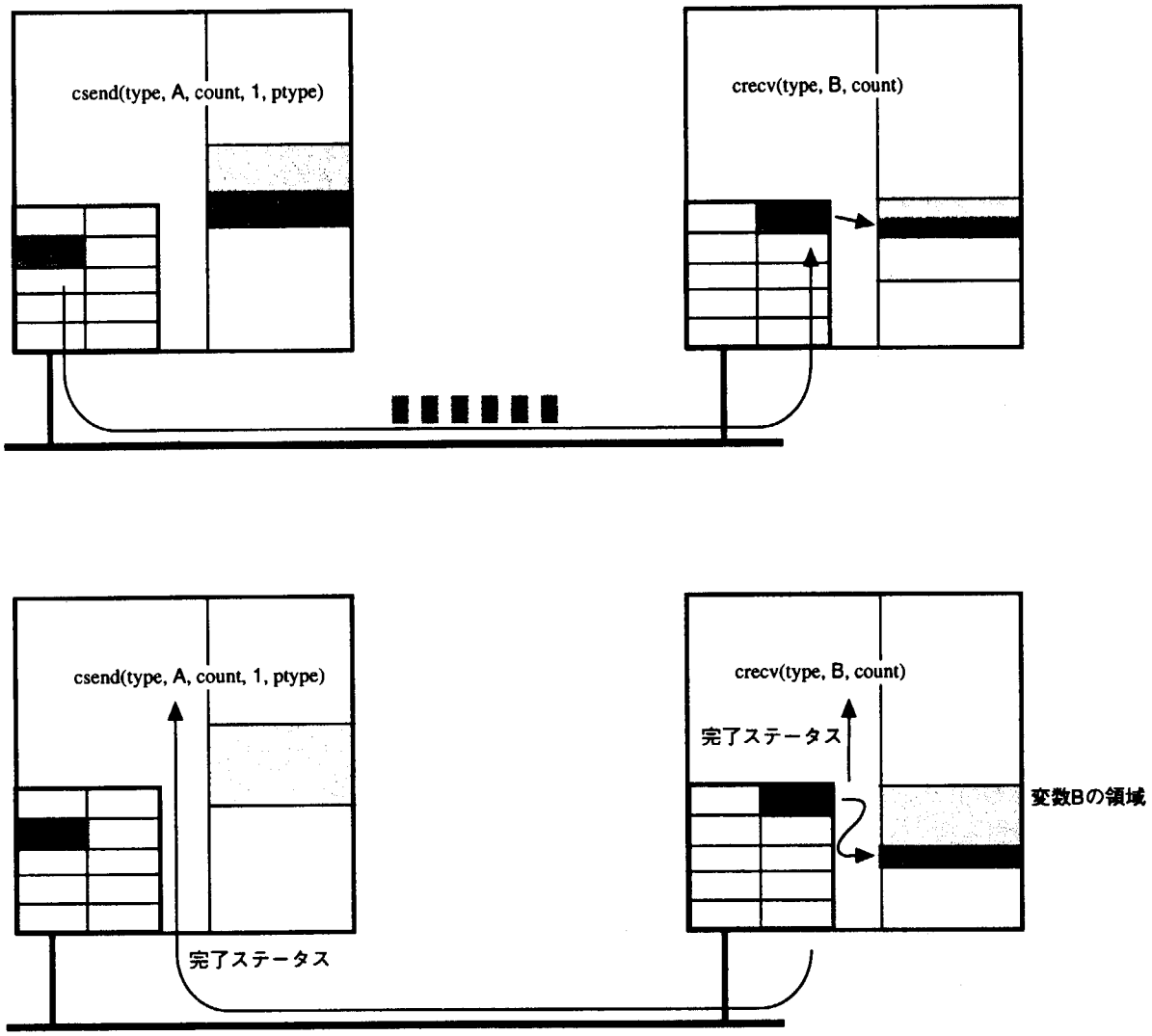
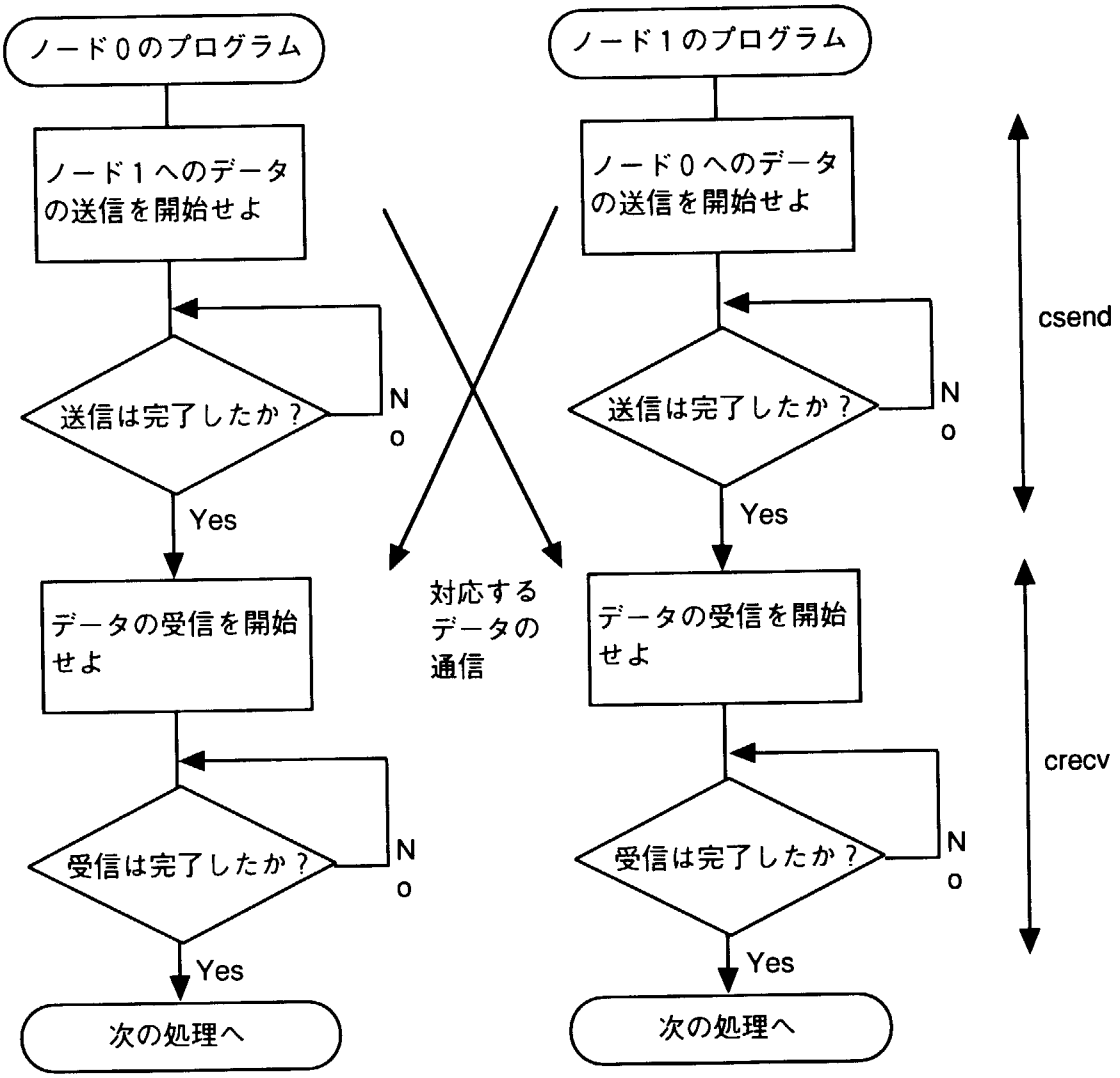


Fig.A3.2 データ量が多い場合のノード間の送受信



FigA.3.3 csendとcrecvによる通信

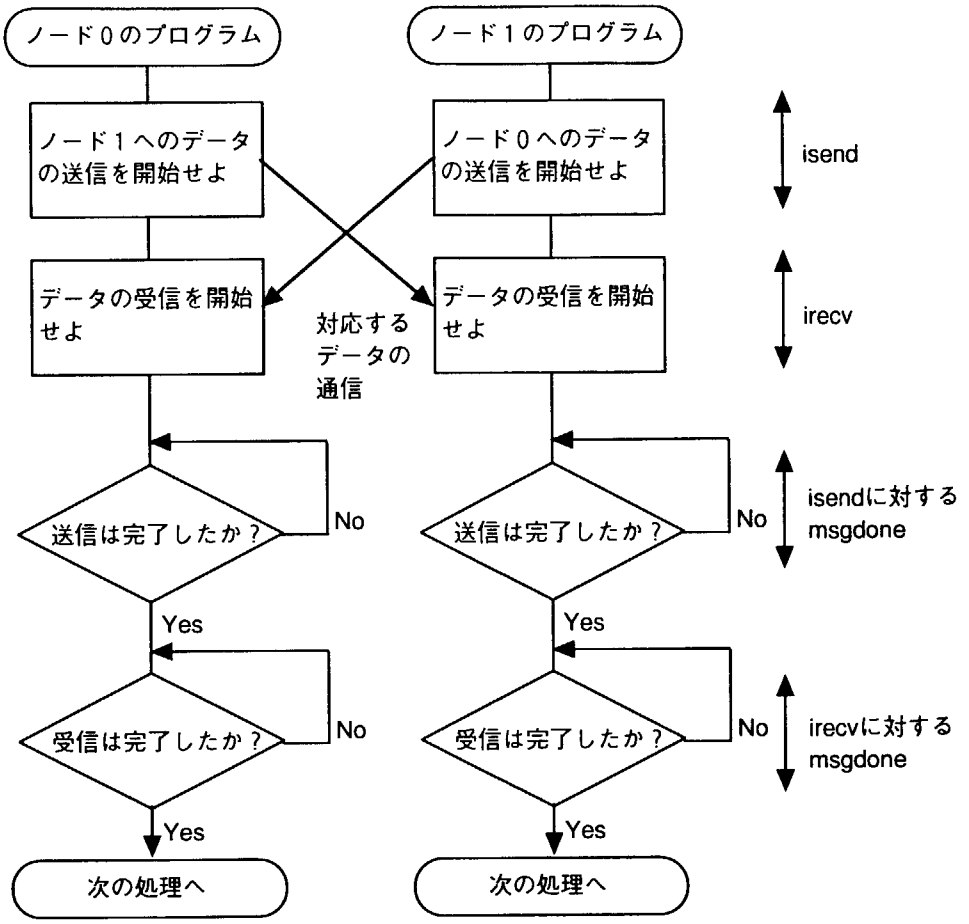


Fig.A3.4 isend、irecv、msgdoneによる通信

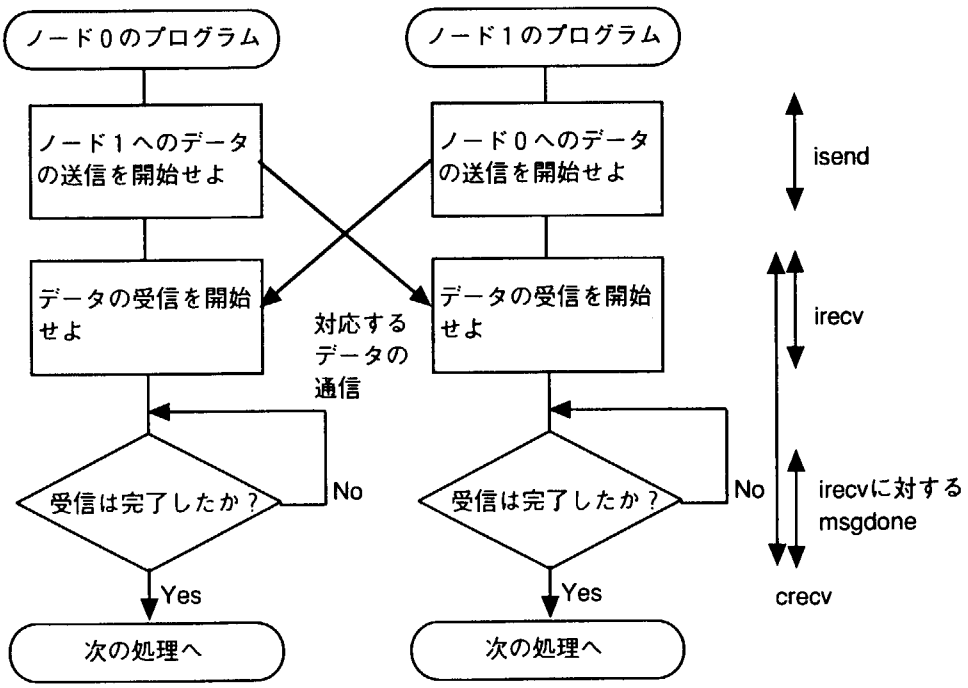


Fig.A3.5 isend、crecvによる通信

付録4 並列入出力の方法

分散記憶型超並列計算機では、計算用ノードの数に比例して保存しなければならないデータ量が非常に多いのに対して、ディスク等のI/Oデバイスの数は限られていて、しかもノード間のネットワークやローカルエリアネットワークを介して接続されているので、入出力が計算機の性能を制約する大きな要因になりうる。

fortranによる科学技術計算を行う時、データの入出力はシーケンシャルにできれば良い場合が大半と考えられる。（Fig.A4.1）に示すように、各々のwrite文ごとにそれぞれ異なるサイズのデータが順々にファイルに書き込まれる。計算に対してディスクへの書き込みの速度は通常遅いので、書き込みの完了を待つ時間が処理速度を制約する原因になる。

これを解決するため、1台のコンピュータに複数台のディスクを装備することがある。（Fig.A4.2）に示すように、データは各ディスクに分割して書き込まれる。ユーザーからは、このようなデータファイルはあくまでも単一のファイルに見えるような技術が使われていて、物理的にどのディスクに書き込まれるかはデータのサイズ等から自動的に計算され、プログラムからは見えないのが普通である。これは一種の出力の並列化であり、書き込みの時間がディスクの台数に反比例して減少し高速化されると期待される。

超並列計算機で計算したデータを出力する場合は新たな問題が発生する。Intel Paragonでは、ディスクは専用のI/O計算ノードに接続されているので、計算ノードからメッセージパッシングによる通信を行うことではじめて読み書きが可能となる。この場合、より多量のデータを通常のメモリ、MPUとディスクの間の転送速度よりも遅いノード間のネットワークを介して出力しなければならない。さらに、複数の計算ノードからひとつのファイルに読み書きを行うために、アクセスの調停を行わなければならない。

これを解決するため、Intel paragonでは、複数の計算用ノードから複数のI/Oノードとの間で読み書きを行うためのpfsが実装されており、メッセージパッシングを応用したデータの入出力の方法が定義されている。pfsに対する出力を開始するには、gopenサブルーチンによりファイルをオープンし、その後cwriteサブルーチンまたはiwrite関数によって書き込みを行う。gopenサブルーチンは全てのノードで実行しなければならないグローバル命令である。

さて、（Fig.A4.3）に示すように、各々のノードが各々のwrite文ごとに異なるサイズのデータの出力を行うとすると、ノード1はノード0の書き込みが終了しなければディスクファイル上の何処に書き込みを行って良いかわからない。gopenサブルーチンでM_SYNCモードを選択すると、アクセスの調停が行われ、各ノードからの読み書きが自動的に順番に行われるようになる。すなわち、（Fig.A4.3）に示すようにノード0の出力が終わった時点で、現在のディスクファイル上の書き込み位置（ファイルポインタの値）がノード1に送られ、ノード1からの出力が行われ、ファイル上には発行したwrite文毎、ノード順にデータが整列して保存される。しかしながら、この方法では同時に1台のノードしか読み書きを行うことが出来ない。ノード1はノード0の書き込みが完了しないと出力を行うことができない。他の全てのノードが読み書きを完了するまで次の処理に進めないので並列処理のメリットを発揮できない。

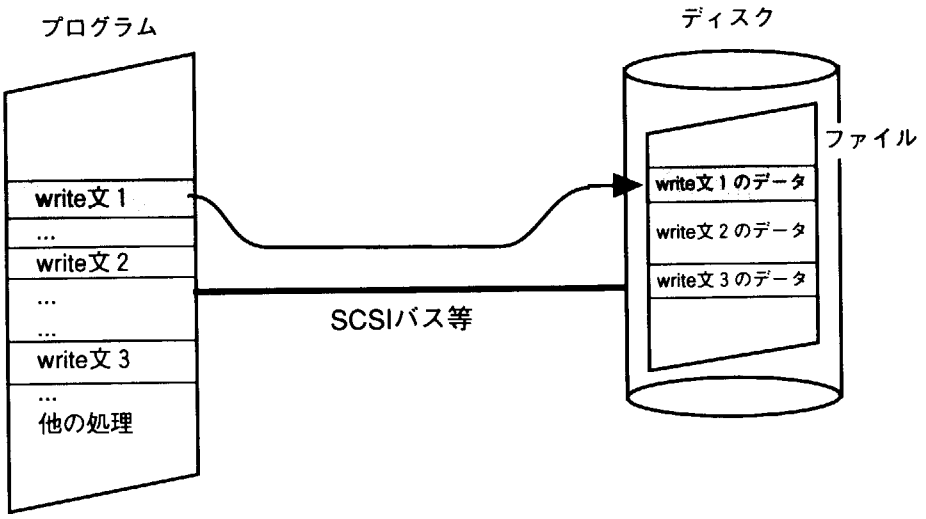
各ノードが発行するwrite文が厳密に同じサイズのデータの出力を行うことが仮定できる場合は、M_RECORDモードを用いてより効率的な出力を行うことができる。この場合、各ノードは他

のノードからの情報なしにそれぞれのwrite文におけるディスクファイル上の書き込み位置（ファイルポインタの値）を計算することができる。したがって、（Fig.A4.4）に示すように各ノードは独立に書き込みを行うことができる。図のように書き込みの進捗がノード毎に異なっていても構わない。出力ファイルはこれまでの場合と異なってシーケンシャルにアクセスされない。I/Oノードは自動的にファイルポインタを移動させて所定の場所にデータを書き込む処理を行う。

M_RECORDモードでの出力の対象となるディスクファイルは、物理的に（Fig.A4.2）のような分割されたディスクのどちらかでも良いし、（Fig.A4.5）のように別のI/Oノードに接続されているディスクでも良いと考えられる。pfsはこのような処理を実現するためのもので、複数のI/Oノードに接続されているディスクに分割されているファイルを、ユーザーからは論理的には一続きのシーケンシャルファイルのように見えるようにする。pfsを活用すると、ノード０からの出力はI/Oノード０で、ノード１からの出力はI/Oノード１で、独立に処理される。このように効率的に並列処理する方法をスプライティングと言う。

pfsを用い並列出力の効率を高めるための注意点は、スプライティングがサイズ64kB毎に機械的に行われることである。各ノードからのひとつのwrite文で出力されるデータサイズが64kB（以下）の場合は、各ノードからのデータが整然と各I/Oノードに送られ処理されるが、64kBを超える場合には別の出力ノードに分割されて振り分けられてしまう。結局、（Fig.A4.6）の例ではそれぞれのwrite文毎にデータがノード０と１の両方に送られ、ノード間ネットワークや各々のノードからの出力要求を受けるI/Oノードの負荷が増加する。これは処理速度の低下とプログラムの信頼性の低下を引き起こす可能性がある。スプライティングされて実際に出力が行われる先のI/Oノードをプログラムから指定することはできない。各write文のデータサイズを厳密に64kBに一致させることは容易ではないので、出力データを64kBずつバッファリングするサブルーチンが必要かも知れない。

以上のようなcwrite、iwriteサブルーチン、関数を用いて出力されたデータは、出力文と完全に一対一対応するcread、ireadサブルーチン、関数で読み込まなければならない。並列出力はI/Oノードへのメッセージパッシングであるとともに、ファイルを介したプログラムからプログラムへのメッセージパッシングと考えることもできる。



(Fig.A4.1) ディスクへのシーケンシャルファイルの書き出し

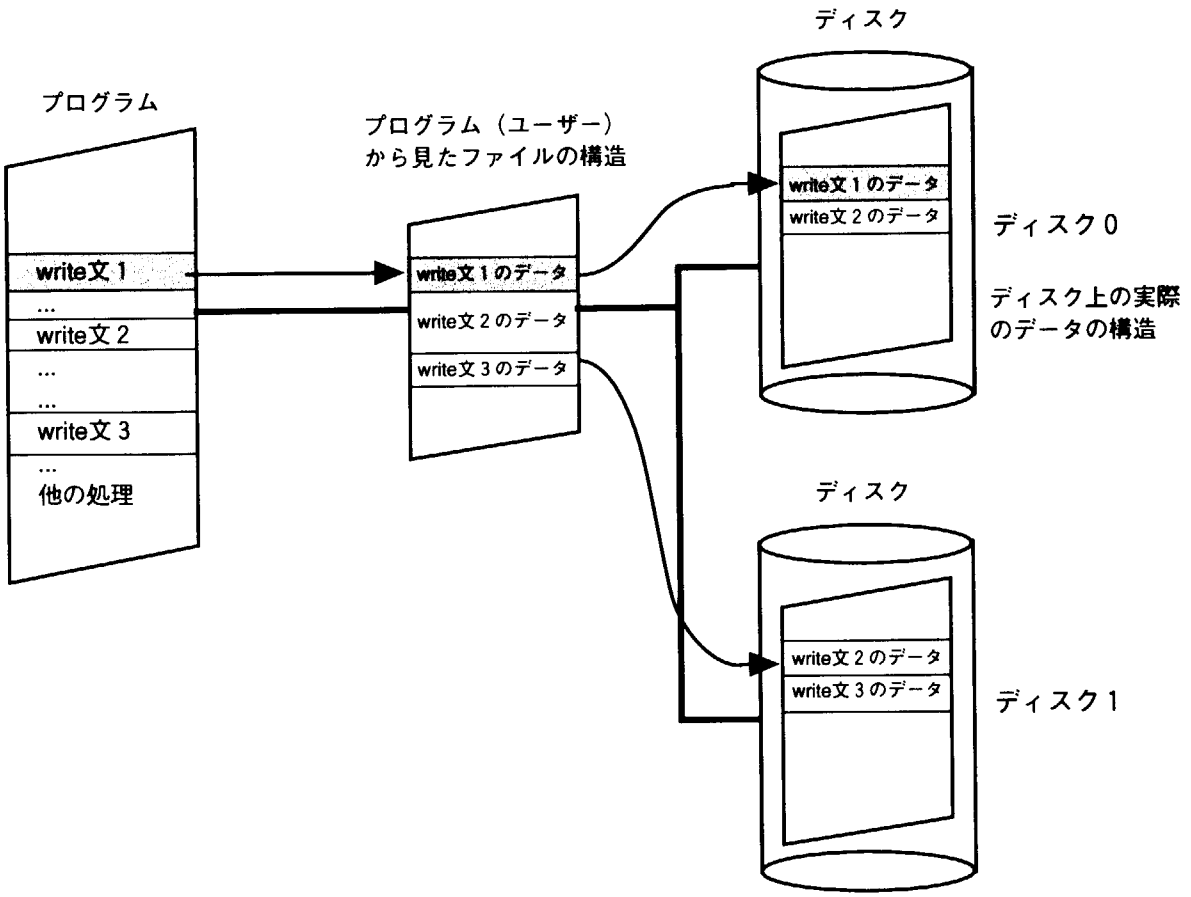


Fig.A4.2 複数ディスクへのシーケンシャルファイルの書き出し

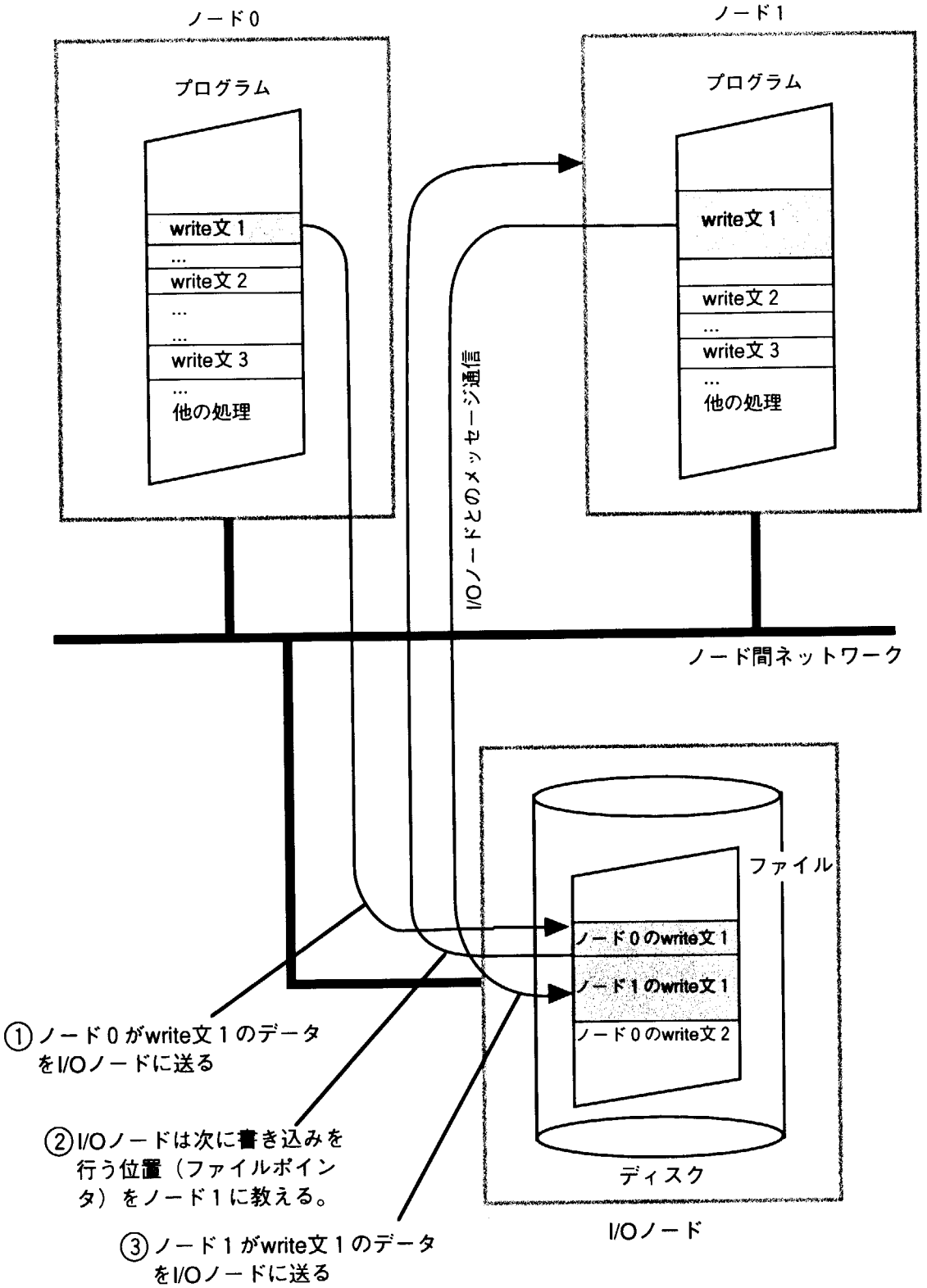
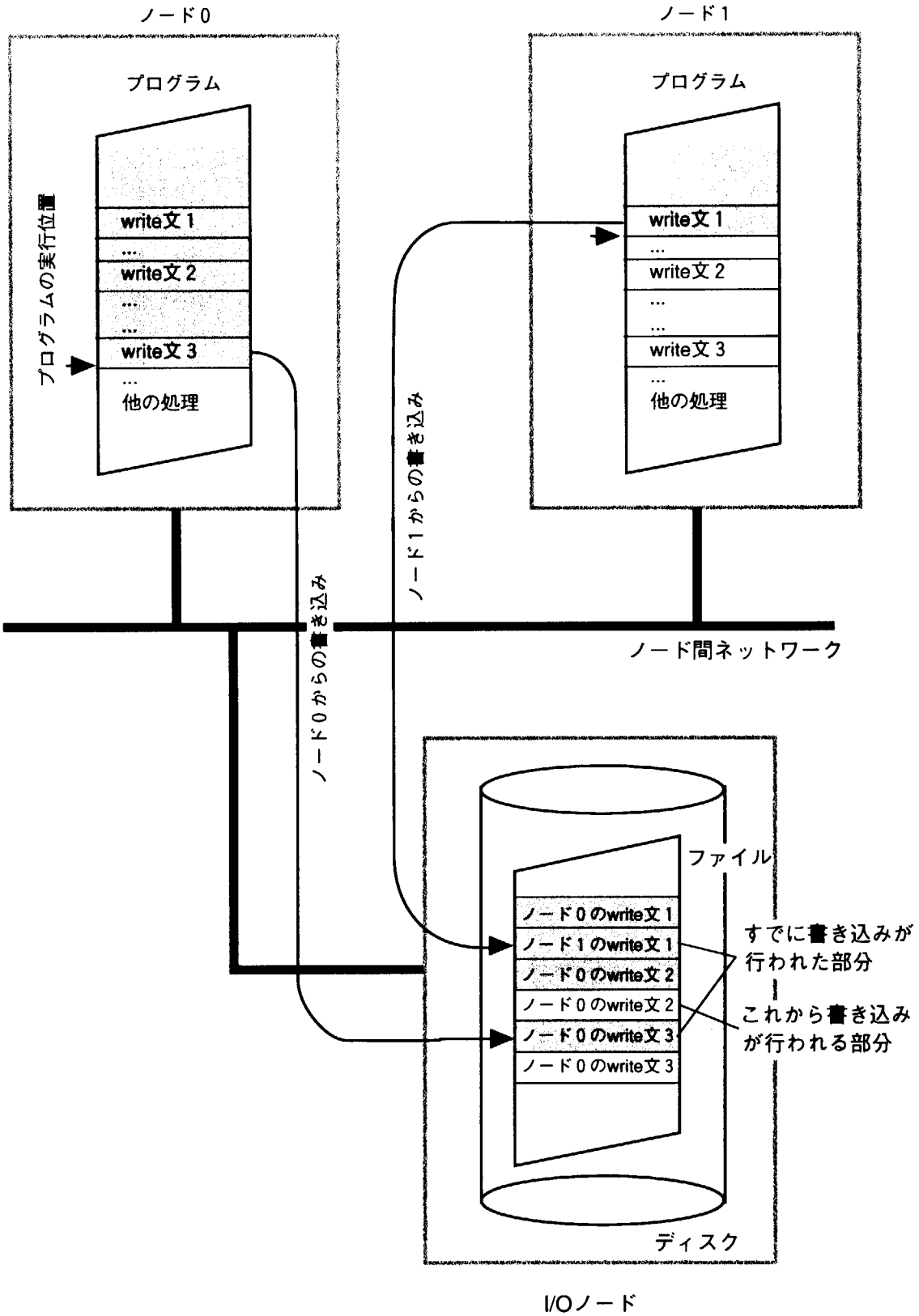
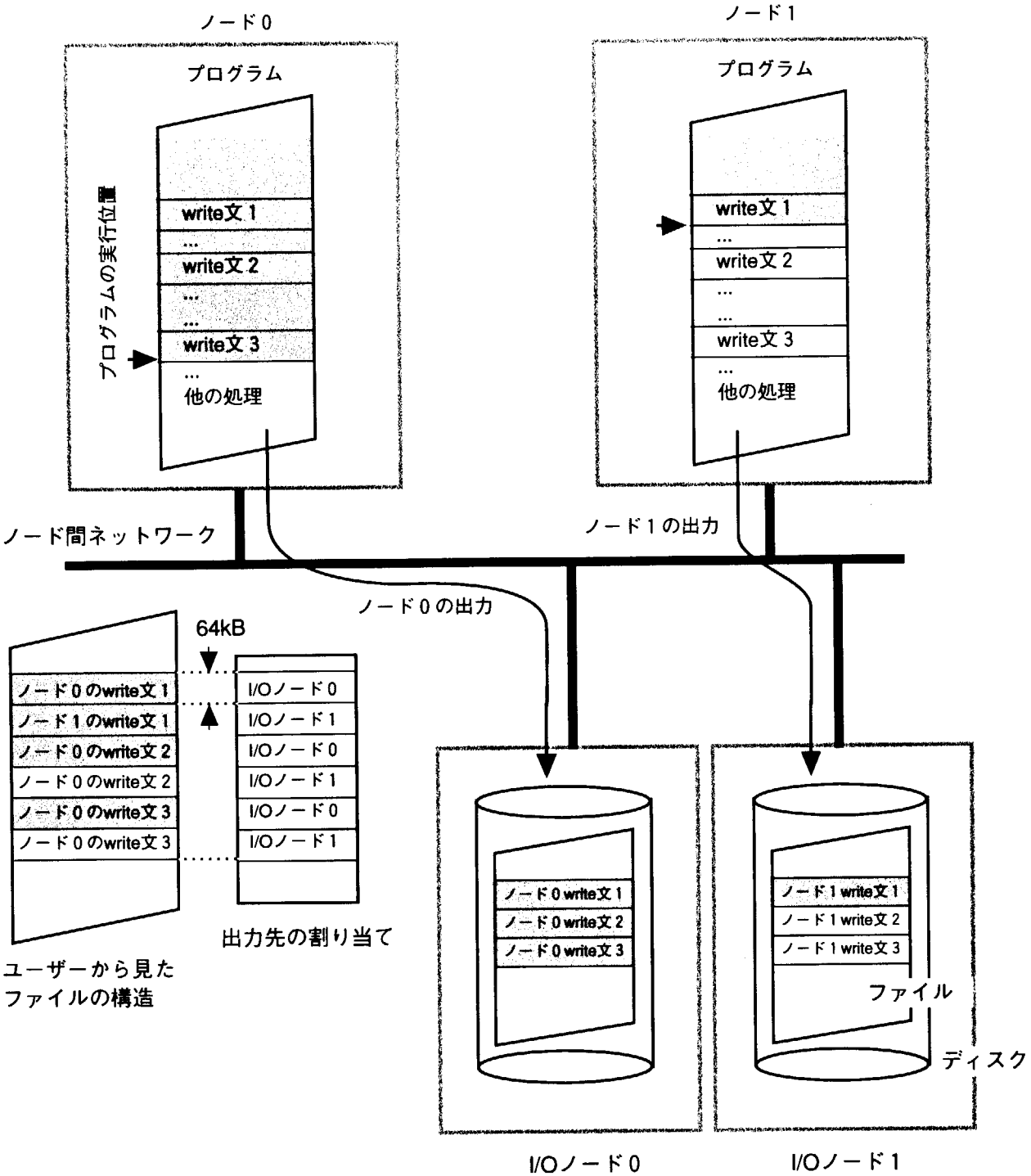


Fig.A4.3 M_SYNC I/Oモードによる複数ノードからディスクへの書き出し



FigA4.4 M_RECORD I/Oモードによる複数ノードからディスクへの書き出し



FigA4.5 pfsを用いた複数ノードから複数I/Oノードへの書き出し

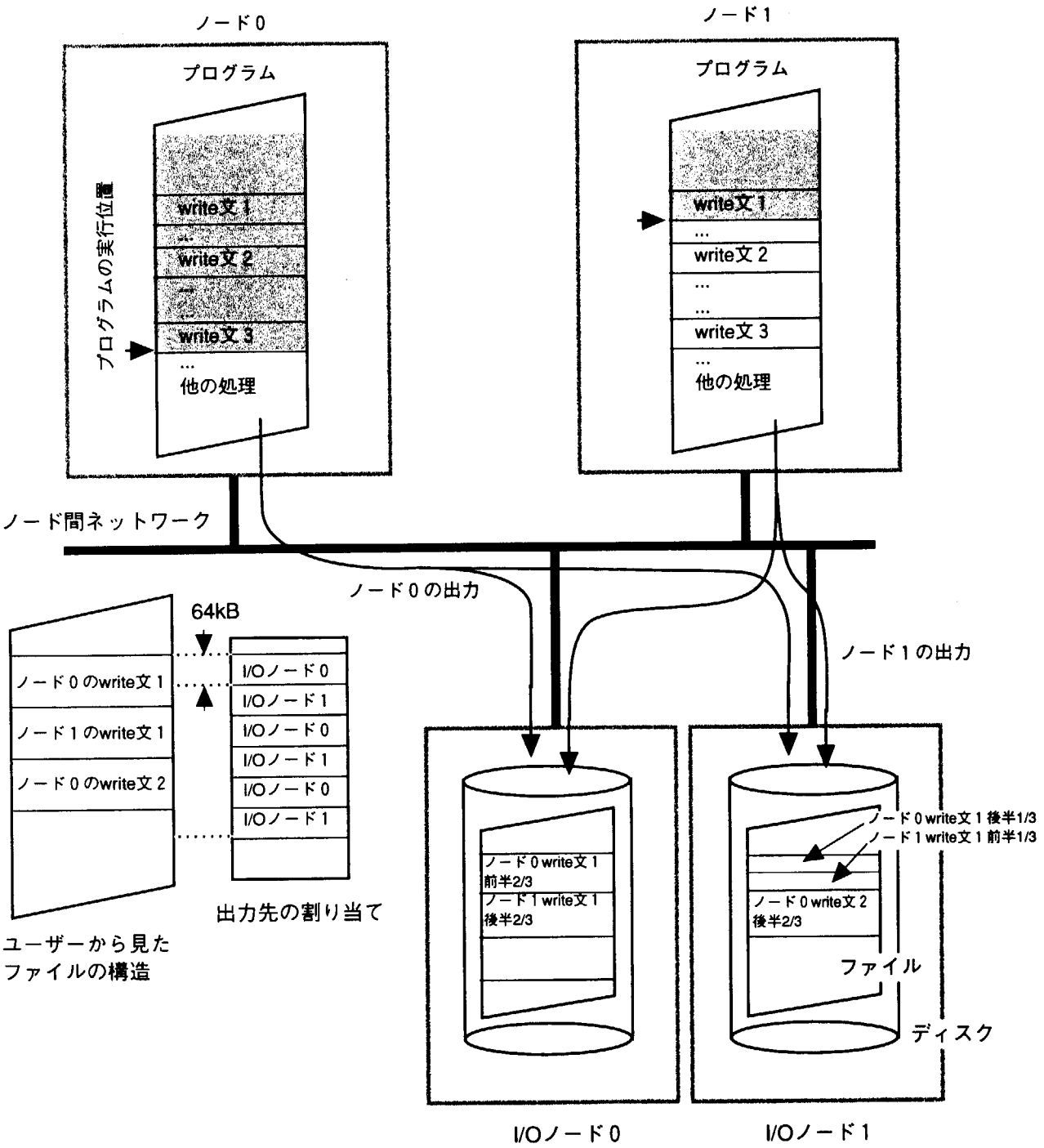


Fig.A4.6 スプライトサイズと出力サイズの不一致時の処理

付録5 割り込み処理によるプログラム実行の制御

プログラムを実行中、入力データの異常、計算結果の異常、入出力装置の異常によりこれを停止しなければならないことがしばしばある。しかし、並列計算機のひとつノードで実行されているプログラムに異常が発生した時、ジョブを安全に停止させるには、すべてのノードで実行されているプログラムを停止させることが必要となる。このような技術は自明ではない。

メッセージパッシングによる通信は、送り側ノードと受け側ノードが一対のcsend、isendとcrecv、irecvを発行することによって成立する。すなわち、各ノードがcsend、isend（crecv、irecv）を発行する時、相手側ノード上でプログラムが正常に動作してcrecv、irecv（csend、isend）が発行されることが前提である。このため、通常のプログラムでは、（Fig.A5.1）のように他のノードで異常が発生していることを積極的に知ることは不可能である。あるノードがstop文を実行したり、実行時エラーで終了しても、他のノードはそのことが分からないのでそのまま実行を続ける。その後のプログラム中に通信が含まれていれば、これは決して完了しないので、プログラムは確実にハングする。異常が発生したノードからこのことを他のノードに伝えようとしても、受け側のノードで適当なcrecv、irecvが発行されていなければ内容は伝わることがない。

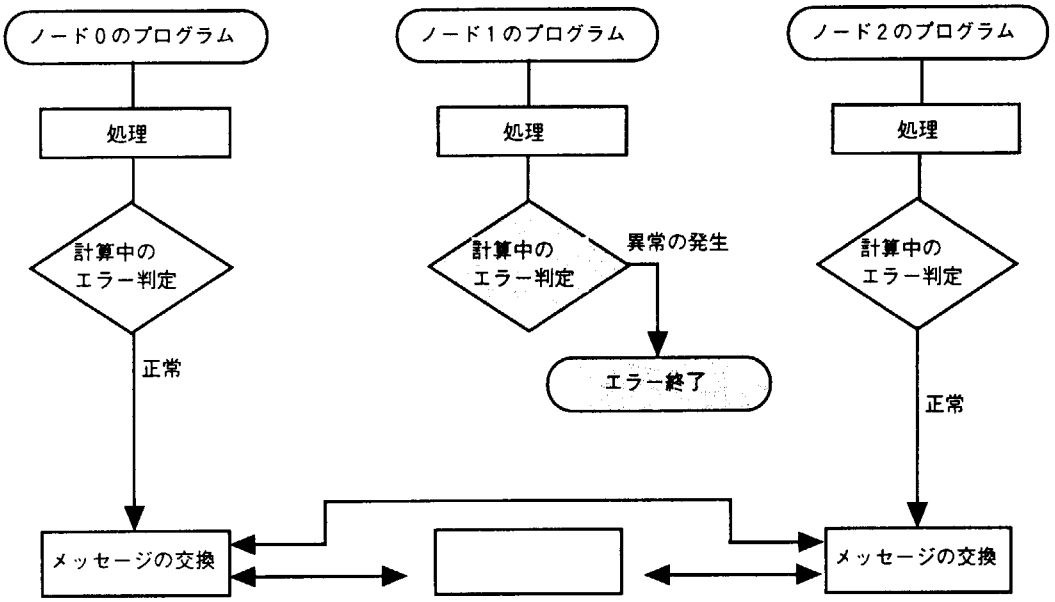
このための対策の一つは、プログラム中の適当な場所で全てのノードがエラー情報を交換しあうことである（Fig.A5.2）。これによって、あるノードで発生したエラーを全てのノードが察知してジョブを終了させることができる。しかしプログラム中にこのための処理を多数回加えると、プログラムが複雑になる上、エラー情報の交換に要する通信時間のために処理速度が損なわれる。特にすでに停止してしまっているノードがあった場合、これを判定する唯一の方法は、所定時間応答がなかった場合にエラーと判定する方法だけである。このための待ち時間は処理速度を損なうと予想される。

Intel Paragonのnxライブラリでは、割り込み型のメッセージパッシングの手続きが定義されていて、これを用いるとより柔軟なエラー処理が行えると考えられる。割り込み処理とは、ハードウエアの信号やソフトウエアのある特定の命令（割り込み）を用いて、現在実行されているプログラムを一時中断させて別のプログラムを実行することである。

割り込みが発生した時に実行されるプログラムを割り込みハンドラと呼ぶ。Intel Paragonでは送り側ノードが割り込み型のメッセージを送ると、受け側ノードでどのような処理行われているかに関わらず割り込みハンドラを起動されるようにすることができる。プログラム本体は、エラーを検出した任意の時点で割り込み型のメッセージを発行するだけで良く、割り込みハンドラはプログラムを終了させる処理のみを行えば良い（Fig.A5.3）。この方法ではエラー情報の交換のための待ち時間もないので高速な処理が行えると考えられる。

割り込みを活用するとより多様な処理を行うことができる。例えば、割り込みハンドラにその時点のデータを書き出す処理を加えると、エラーの原因の解析に便利である。長時間のジョブの場合、書き出されたデータをもとにリスタートできればCPU時間を有効に活用することもできるだろう。反面、複雑なプログラムには、メモリアロケーションやpfsへの読み書きなど中断が許されない処理（クリティカルセクション）が存在し、このような状況下で割り込み処理を活用するにはより高度なプログラム技術が必要になる。

MPIを用いる場合は、エラーが発生したノードでMPI_ABORT関数を呼び出すことにより、単純にすべてのノードで実行されているプログラムを停止させ、ジョブを終了させることができる。



(Fig.A5.1) 一つのノードでのエラーによるプログラムハングの例

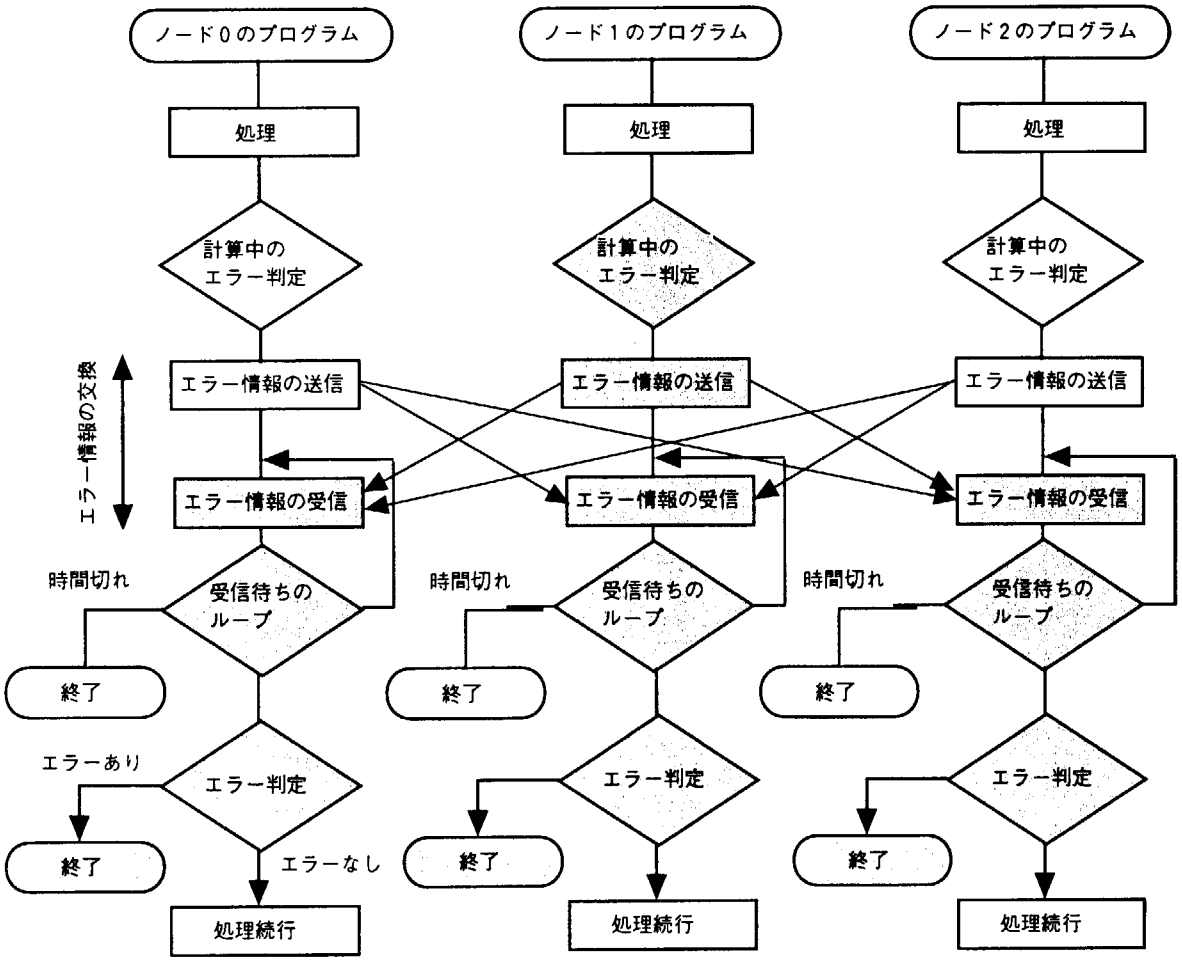


Fig.A5.2 エラー情報の交換を加えたプログラム

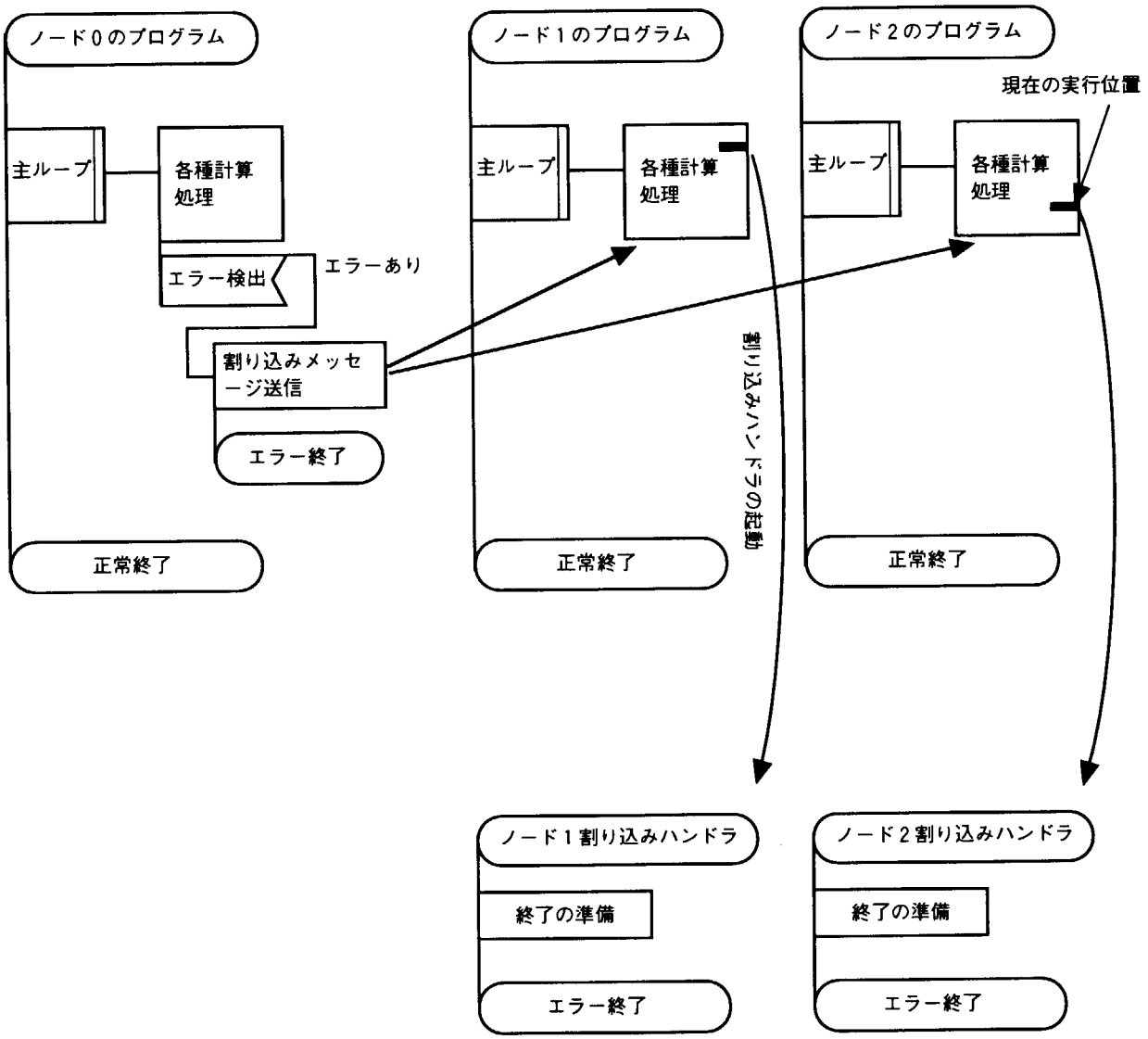


Fig.A5.3 割り込みによるエラー処理

This is a blank page.

国際単位系 (SI) と換算表

表 1　SI 基本単位および補助単位

量	名　称	記　号
長　さ	メートル	m
質　量	キログラム	kg
時　間	秒	s
電　流	アンペア	A
熱力学温度	ケルビン	K
物質量	モル	mol
光　度	カンデラ	cd
平面角	ラジアン	rad
立体角	ステラジアン	sr

表 2　SI と併用される単位

名　称	記　号
分，時，日	min，h，d
度，分，秒	°，′，″
リットル	l，L
トン	t
電子ボルト	eV
原子質量単位	u

$1\ \mathrm{eV} = 1.60218 \times 10^{-19}\ \mathrm{J}$

$1\ \mathrm{u} = 1.66054 \times 10^{-27}\ \mathrm{kg}$

表 5　SI 接頭語

倍数	接頭語	記　号
10^{18}	エクサ	E
10^{15}	ペタ	P
10^{12}	テラ	T
10^{9}	ギガ	G
10^{6}	メガ	M
10^{3}	キロ	k
10^{2}	ヘクト	h
10^{1}	デカ	da
10^{-1}	デシ	d
10^{-2}	センチ	c
10^{-3}	ミリ	m
10^{-6}	マイクロ	μ
10^{-9}	ナノ	n
10^{-12}	ピコ	p
10^{-15}	フェムト	f
10^{-18}	アト	a

表 3　固有の名称をもつ SI 組立単位

量	名　称	記号	他の SI 単位による表現
周波数	ヘルツ	Hz	s^{-1}
力	ニュートン	N	$m \cdot kg/s^2$
圧力，応力	パスカル	Pa	N/m^2
エネルギー，仕事，熱量	ジュール	J	N·m
工率，放射束	ワット	W	J/s
電気量，電荷	クーロン	C	A·s
電位，電圧，起電力	ボルト	V	W/A
静電容量	ファラド	F	C/V
電気抵抗	オーム	Ω	V/A
コンダクタンス	ジーメンス	S	A/V
磁束	ウェーバ	Wb	V·s
磁束密度	テスラ	T	Wb/m^2
インダクタンス	ヘンリー	H	Wb/A
セルシウス温度	セルシウス度	℃	
光束	ルーメン	lm	cd·sr
照度	ルクス	lx	lm/m^2
放射能	ベクレル	Bq	s^{-1}
吸収線量	グレイ	Gy	J/kg
線量当量	シーベルト	Sv	J/kg

表 4　SI と共に暫定的に維持される単位

名　称	記　号
オングストローム	Å
バーン	b
バール	bar
ガル	Gal
キュリー	Ci
レントゲン	R
ラド	rad
レム	rem

$1\ \text{Å} = 0.1\ \mathrm{nm} = 10^{-10}\ \mathrm{m}$

$1\ \mathrm{b} = 100\ \mathrm{fm}^2 = 10^{-28}\ \mathrm{m}^2$

$1\ \mathrm{bar} = 0.1\ \mathrm{MPa} = 10^5\ \mathrm{Pa}$

$1\ \mathrm{Gal} = 1\ \mathrm{cm/s}^2 = 10^{-2}\ \mathrm{m/s}^2$

$1\ \mathrm{Ci} = 3.7 \times 10^{10}\ \mathrm{Bq}$

$1\ \mathrm{R} = 2.58 \times 10^{-4}\ \mathrm{C/kg}$

$1\ \mathrm{rad} = 1\ \mathrm{cGy} = 10^{-2}\ \mathrm{Gy}$

$1\ \mathrm{rem} = 1\ \mathrm{cSv} = 10^{-2}\ \mathrm{Sv}$

（注）

1. 表 1－5 は「国際単位系」第 5 版，国際度量衡局 1985 年刊行による。ただし，1 eV および 1 u の値は CODATA の 1986 年推奨値によった。
2. 表 4 には海里，ノット，アール，ヘクタールも含まれているが日常の単位なのでここでは省略した。
3. bar は，JIS では流体の圧力を表わす場合に限り表 2 のカテゴリーに分類されている。
4. EC 閣僚理事会指令では bar，barn および「血圧の単位」mmHg を表 2 のカテゴリーに入れている。

換　算　表

力	N(=10^5dyn)	kgf	lbf
	1	0.101972	0.224809
	9.80665	1	2.20462
	4.44822	0.453592	1

粘　度　1 Pa·s(N·s/m^2)=10 P(ポアズ)(g/(cm·s))

動粘度　1 m^2/s=10^4St(ストークス)(cm^2/s)

圧力	MPa(=10 bar)	kgf/cm^2	atm	mmHg(Torr)	lbf/in^2(psi)
	1	10.1972	9.86923	7.50062×10^3	145.038
	0.0980665	1	0.967841	735.559	14.2233
	0.101325	1.03323	1	760	14.6959
	1.33322×10^{-4}	1.35951×10^{-3}	1.31579×10^{-3}	1	1.93368×10^{-2}
	6.89476×10^{-3}	7.03070×10^{-2}	6.80460×10^{-2}	51.7149	1

エネルギー・仕事・熱量	J(=10^7 erg)	kgf·m	kW·h	cal(計量法)	Btu	ft·lbf	eV
	1	0.101972	2.77778×10^{-7}	0.238889	9.47813×10^{-4}	0.737562	6.24150×10^{18}
	9.80665	1	2.72407×10^{-6}	2.34270	9.29487×10^{-3}	7.23301	6.12082×10^{19}
	3.6×10^6	3.67098×10^5	1	8.59999×10^5	3412.13	2.65522×10^6	2.24694×10^{25}
	4.18605	0.426858	1.16279×10^{-6}	1	3.96759×10^{-3}	3.08747	2.61272×10^{19}
	1055.06	107.586	2.93072×10^{-4}	252.042	1	778.172	6.58515×10^{21}
	1.35582	0.138255	3.76616×10^{-7}	0.323890	1.28506×10^{-3}	1	8.46233×10^{18}
	1.60218×10^{-19}	1.63377×10^{-20}	4.45050×10^{-26}	3.82743×10^{-20}	1.51857×10^{-22}	1.18171×10^{-19}	1

1 cal = 4.18605 J（計量法）

= 4.184 J　（熱化学）

= 4.1855 J　（15 °C）

= 4.1868 J（国際蒸気表）

仕事率　1 PS（仏馬力）

= 75 kgf·m/s

= 735.499 W

放射能	Bq	Ci
	1	2.70270×10^{-11}
	3.7×10^{10}	1

吸収線量	Gy	rad
	1	100
	0.01	1

照射線量	C/kg	R
	1	3876
	2.58×10^{-4}	1

線量当量	Sv	rem
	1	100
	0.01	1

（86 年 12 月 26 日現在）

Paragon上でのスカラー超並列プログラム開発ガイド