［大規模科学計算システム］

SX-ACE でのプログラミング（並列化編）

— 共有並列化と分散並列化 —

林 康晴
日本電気株式会社

SX-ACE は、4 個の CPU コアを備えたノードを、ノード間スイッチ IXS（Inter-node X-bar Switch）により接続したスーパーコンピュータシステムです。SX-ACE のハードウェア性能を一つのプログラムから十分に引き出すには、プログラムの並列化により、複数の CPU コアを有効に活用する必要があります。SX-ACE は、並列処理環境としてプログラミング言語 Fortran、C、及び C++、並びに 通信ライブラリ MPI を用意しています。本稿は、第 1 章で並列処理の基本事項をご説明した後、第 2 章では SX-ACE の Fortran 95 コンパイラ FORTRAN90/SX（以後、単にコンパイラと記します）による共有並列化、第 3 章では SX-ACE の MPI ライブラリ MPI/SX による分散並列化を中心にご紹介します。

1. 並列処理

並列処理とは、一つの仕事を複数の小さな仕事に分割し、それらの仕事を複数同時に実行することです。例えば、2 重ループを四つの仕事に分割し、4 個の CPU コア上で並列実行すると、図 1.1 のようになります。この場合、外側の do j のループの 100 回の繰返しを 4 等分し、各 CPU コア上で並列に実行しています。

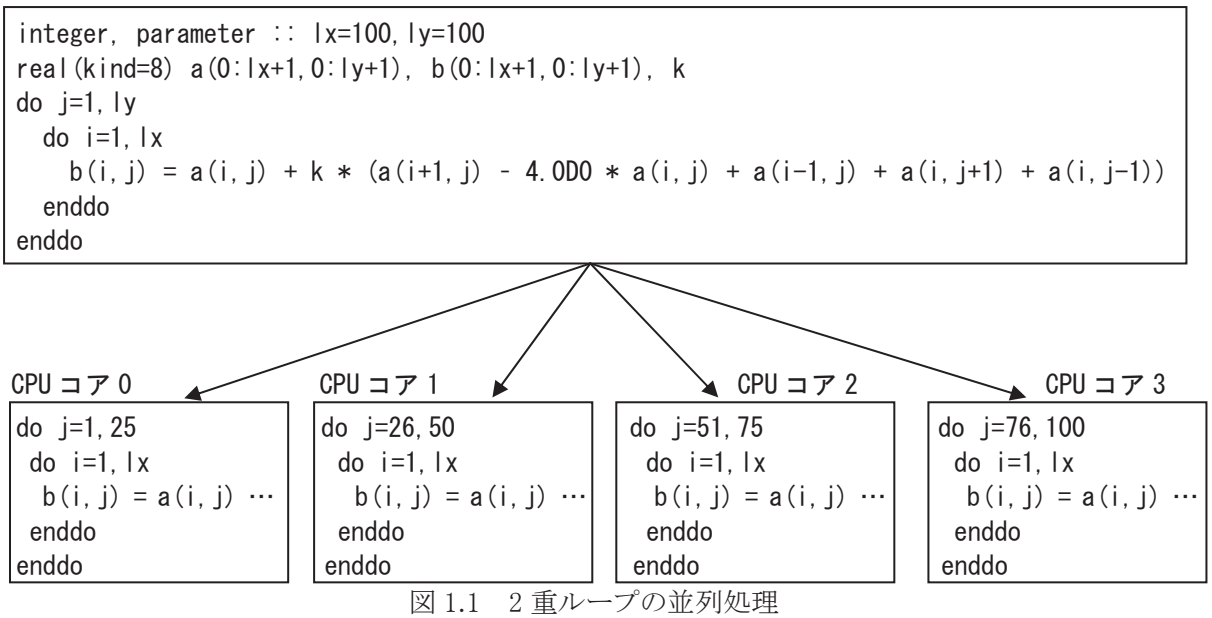

図 1.1　2 重ループの並列処理

1.1　並列化可能な条件

並列実行中、複数の仕事は、一般には実行タイミングの調整（同期）なしで、それぞれ独立に実行されます。そのため、複数の仕事内の各処理の間には、定まった実行順序はなく、一般には実行毎に異なる順序で実行されます。従って、ループが並列実行可能であるためには、ループの繰返しをどのような順序で実行しても実行結果が変わらないことが必要です。例えば、図 1.2 の二つのループ(a)、(b)は、いずれもループ中で確定される各配列要素 a(i) 及び a(i,1), (i=2,3,…,n) が、各ループ(a)、(b)の全繰返し中でそれぞれ 1 度しか出現しません。そのため、ループの繰返しをどのような順序で実行しても結果は同じ

になるので、並列実行可能です。一方、図 1.3 の(a)、(b)の場合、いずれもループの k 回目の繰返しで値が確定される配列要素 a(k)が、(a)の場合 k+1 回目の繰返しで、(b)の場合 k-1 回目の繰返しでそれぞれ引用されます。そのため、ループの繰返しの実行順序に依存して、確定前の値を引用するか、確定後の値を引用するかが変わってしまい、並列化することはできません。図 1.3 の(c)の場合も、ループの実行終了時に変数 ifound に残る値が、ループの繰返しの実行順序に依存して変わってしまうので、並列化することはできません。このように、ループを並列化するためには、一つの繰返しで確定したデータ要素は、他の繰返しでは確定も引用もしないようにループを記述する必要があります。図 1.3 の(d)のように、ループ外への飛越しを含むループも、ループの繰返しの実行順序に依存して飛越しのタイミングが変わってしまうので、並列化することはできません。

図 1.4 の二つのループ(a)、(b)は、スカラ変数 t, s のそれぞれを各ループの全繰返しで確定するので、図 1.3 の条件に該当し、一見並列化できないように思えます。しかし(a)の場合、変数 t は、ループの繰返し毎に新たに確定した値を、その繰返しの中だけで引用しています。そのため、ループ実行後に変数 t の値を引用しないのであれば、仕事毎に作業変数を確保して、並列実行中は、変数 t の代わりにその作業変数を参照すれば並列化可能です。また(b)のように、同一の変数に同一の演算を繰返し適用するパターンは集計計算と呼ばれます。集計計算の場合も、やはり仕事毎に作業変数を確保して、並列実行中はその作業変数に各仕事の局所的な計算結果を格納し、並列実行終了時に全仕事の値を集計することにより並列化できます。

多重ループの場合、並列化できるかどうかはループ毎に判断します。図 1.5 の内側の do i のループは並列化可能ですが、外側の do j のループは、左辺の a(i,j)と右辺の a(i,j-1)との間に図 1.3(a)と同様の依存関係があるので並列化できません。

```
integer, parameter :: n=100
real(kind=8), dimension(n) :: a, b
do i=2,n
  a(i) = b(i) + b(i-1)
enddo
```

(a)

```
integer, parameter :: n=100
real(kind=8), dimension(n,n) :: a
do i=2,n
   a(i,1) = a(i,2) + a(i-1,2)
enddo
```

(b)

図 1.2　並列化できるループ

```
integer, parameter :: n=100
real(kind=8), dimension(n) :: a
do i=2,n
  a(i) = a(i) + a(i-1)
enddo
```

(a) 依存

```
integer, parameter :: n=100
real(kind=8), dimension(n) :: a
do i=1,n-1
   a(i) = a(i) + a(i+1)
enddo
```

(b) 逆依存

```
integer, parameter :: n=100
real(kind=8), dimension(n) :: a
integer ifound
do i=1,n
  if(a(i).ge.0) ifound = i
enddo
```

(c) 出力依存

```
integer, parameter :: n=100
real(kind=8), dimension(n) :: a
do i=1,n
  if(a(i).ge.0) goto 100
enddo
100 continue
```

(d) 制御依存

図 1.3　並列化できないループ

```
integer, parameter :: n=100
real(kind=8) :: a(n), b(n), t
do i=1,n-1
  t = a(i) + a(i+1)
  b(i) = t
enddo
```

(a) 繰返し内作業変数

```
integer, parameter :: n=100
real(kind=8) :: a(n), s
do i=1,n
  s = s + a(i)
enddo
```

(b) 集計計算

図 1.4　作業変数の使用により並列化できるループ

```
integer, parameter :: n=100
real(kind=8), dimension(n,n) :: a
do j=2,n
  do i=1,n
    a(i,j) = a(i,j-1) + a(i,j)
  enddo
enddo
```

図 1.5　内側のみ並列化できる多重ループ

1.2 並列化の効果

一つのプログラムを N 個の CPU コア上で並列実行した場合、最大 N 倍のスピードアップ(実行時間の短縮)が期待できます。このスピードアップを最大にするには、まず、どれだけ多くの部分を並列化できたか(並列化率)が重要です。これは、図 1.6 のように逐次実行時間の 10 %分の仕事が並列化できない場合、残りの仕事をどれだけ多くの小さな仕事に分割して並列化しても、スピードアップは 10 倍未満になってしまうことからも分かります。

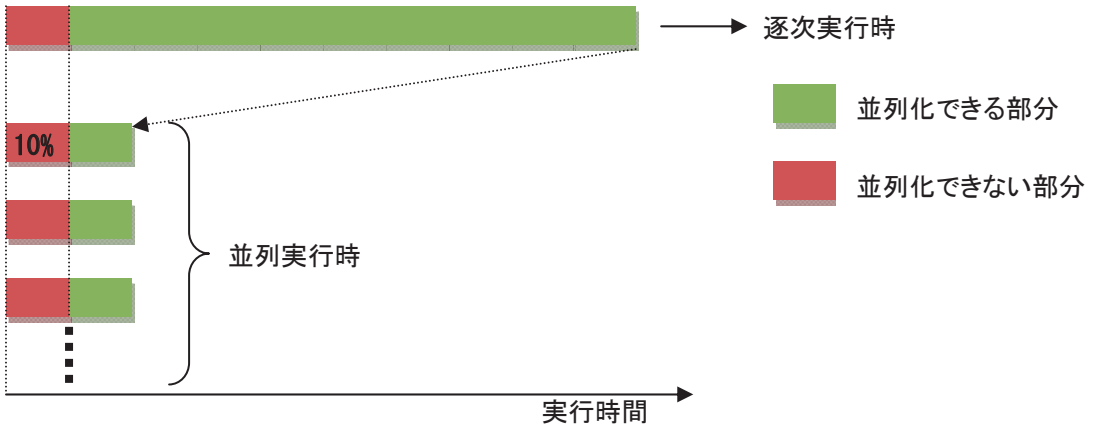

図 1.6　並列化率とスピードアップ(並列化率 90 %・スピードアップは 10 倍未満)

また、仕事を複数の小さな仕事に分割するための処理や、仕事間のデータのやり取り(通信)及び同期といった逐次実行時には必要なかった処理(オーバヘッド)をできるだけ小さくすることも重要です。これは、図 1.7 のように逐次実行時間の 10 %分のオーバヘッドが並列化により発生すると、スピードアップはやはり 10 倍未満になってしまうことからも分かります。特に、実行時間(粒度)の小さなループを並列化するような場合、オーバヘッドの方が大きすぎ、並列化によりかえって遅くなる場合もあります。オーバヘッドを削減するには、できるだけ外側のループで並列化する、といった方法により、各仕事の粒度をできるだけ

大きくすると効果的です。

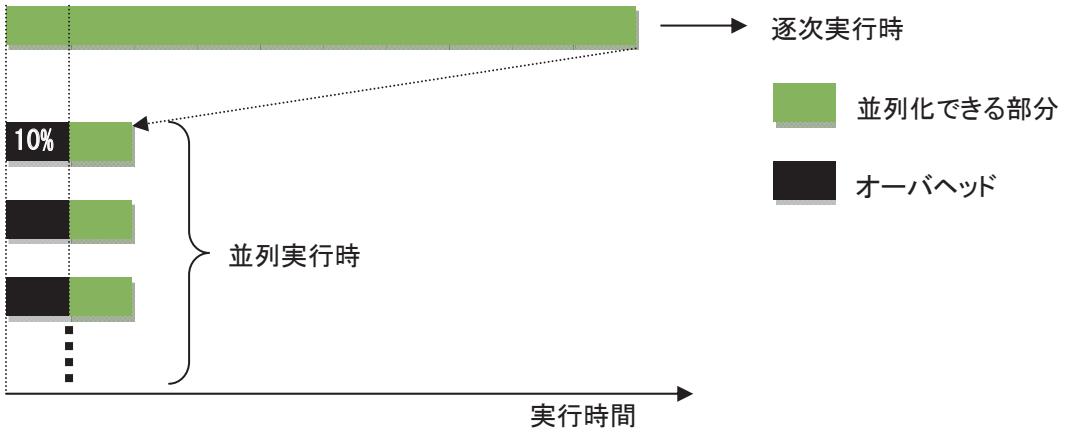

図 1.7　オーバヘッドとスピードアップ（オーバヘッド 10 %・スピードアップは 10 倍未満）

さらに、各仕事の実行時間のバランス（負荷バランス）をできるだけ均等化することも重要です。これは、図 1.8 のように複数の仕事のうち一つの仕事の実行に逐次実行時の 10 %分の時間がかかってしまったとすると、残りの仕事をどれだけ高速化しても、スピードアップは 10 倍以下になってしまうことからも分かります。

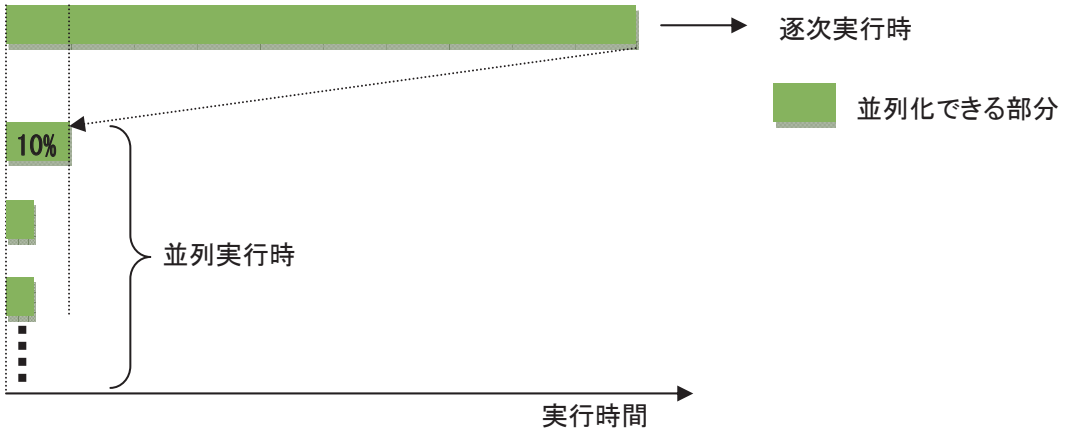

図 1.8　負荷バランスとスピードアップ（スピードアップは 10 倍以下）

このように、並列プログラムにおいて高いスピードアップを達成するには、高い並列化率・低いオーバヘッド・均等な負荷バランスの 3 点が重要です。

1.3　経過時間と CPU 時間

並列処理は、一つの仕事を複数の小さな仕事に分割して、複数の CPU コア上で同時に実行することにより、仕事の実行時間（経過時間）を減らします。一方、全仕事を実行するための全 CPU コアの処理時間の総計（CPU 時間）を減らすことはできません。実際には、一つの仕事を複数の小さな仕事に分割するための処理や仕事間の同期などのオーバヘッドも発生するので、CPU 時間は並列化によりむしろ増加することに注意してください。

1.4　経過時間とチューニング

並列化の主たる目的であるプログラム実行の経過時間短縮のためには、アルゴリズムの工夫、高速な

ライブラリの使用、及びベクトル化の促進などにより、逐次プログラムとしての性能を並列化の前に十分チューニングしておくことが重要です。並列化によるスピードアップは、使用する CPU コアの個数が上限となりますが、逐次プログラムのチューニングによる高速化は、場合により数 10～数 100 倍に達することもあり、さらに CPU 時間も削減できるからです。

2. 共有並列化

一般に、コンピュータのプログラムは、プロセスにより実行されます。実行中のプロセスは、そのメモリ空間にプログラムのデータを格納しており、現在の実行状態をもっています。プロセスは、複数のタスクから構成することもできます。一つのプロセス中の複数のタスクは、それぞれ独自の実行状態をもち、別々の仕事を実行できますが、プロセスのメモリ空間は全タスクが共有します。

共有並列化とは、一つのプロセス中の複数のタスクによる並列処理のことです。SX-ACE の各ノード内では、主記憶装置を共有する 4 個の CPU コア上で実行されるタスクに仕事を割り当て、共有並列化を行うことができます。

例として、図 1.1 の 2 重ループの共有並列化を考えます。第 1 章と同様に、外側の do j のループを四つの仕事に分割して別々のタスクに割り当てます。この場合、データ領域に関しては、図 2.1 のように、2 次元配列 a, b を do j のループに対応する 2 次元目で分割し、分割された各部分領域の処理を各タスクが担当することになります。ここで、例えば j=25 の繰返しを担当するタスク 0 は、タスク 1 の担当領域である配列要素 a(i,26), (i=1,2,…,lx) の値を引用する、といったように、配列の分割境界部分の処理では、自タスクの担当領域外の配列要素を参照する必要があります。しかし、各タスクは、メモリ上に配置された 2 次元配列 a, b の全領域を直接参照できるので、特に問題はありません。一方、DO 変数 i, j については注意が必要です。並列実行中、各タスクは変数 i, j をそれぞれ独自のタイミングで参照します。そのため、配列 a, b と同様に、変数 i, j の領域を全タスクが共有すると、あるタスクが確定した変数 i, j の値を別のタスクが引用してしまう可能性があり、実行が正しく行えません。従って、並列実行中に各タスクが値を確定する変数は、図 2.2 のようにそれぞれのタスクが専用の作業領域を割り付けて参照する必要があります。2 次元配列 a, b のような全タスクがメモリ領域を共有するデータをタスク間共有データ、DO 変数 i, j のような各タスクが専用のメモリ領域を割り付けるデータをタスク固有データと呼びます。

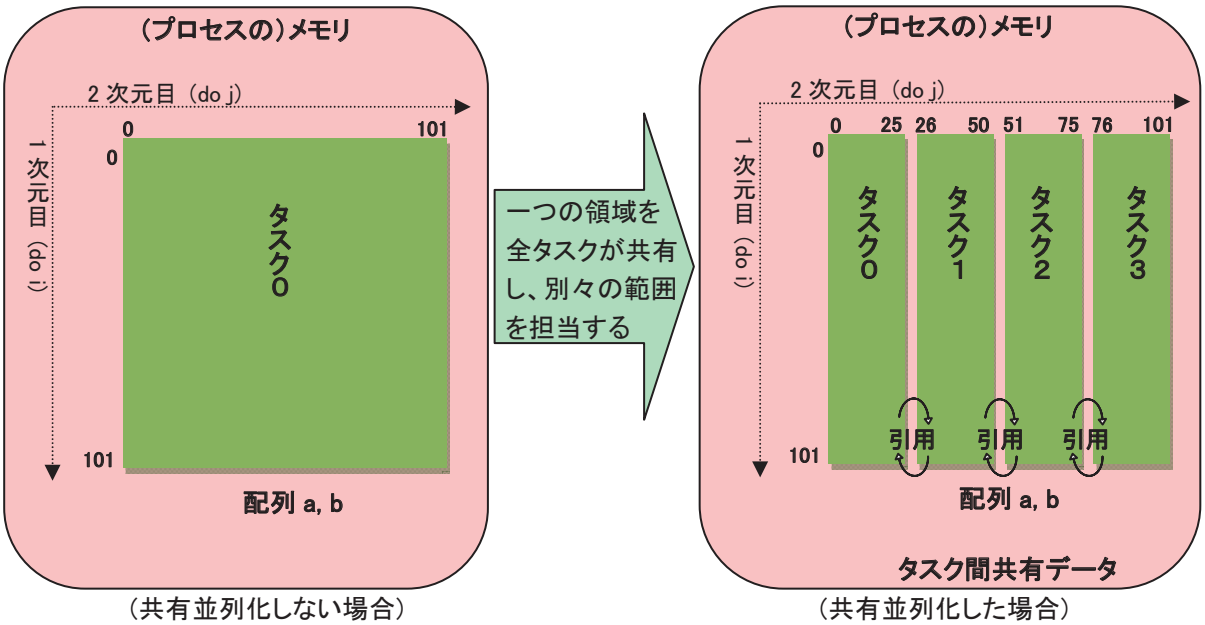

図 2.1　各タスクが担当する配列 a, b の領域

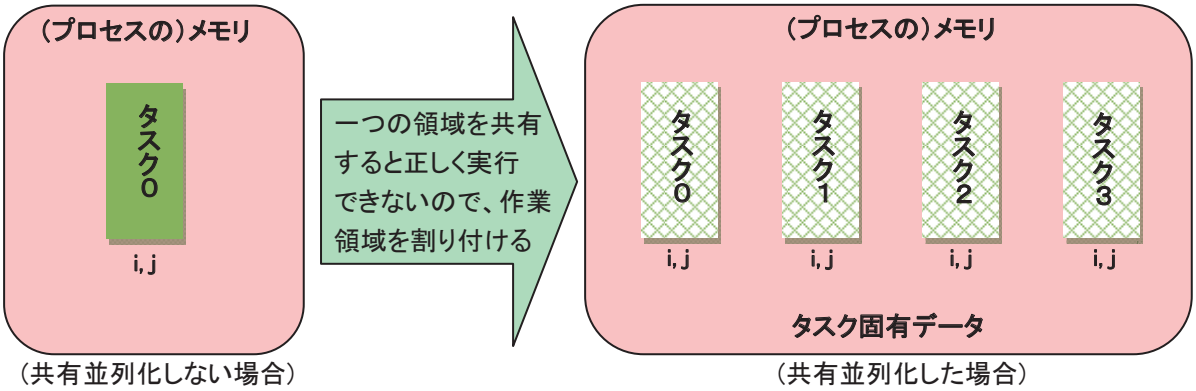

図 2.2　各タスク専用の DO 変数 i, j 用の作業領域

2.1 自動並列化

並列化を行う場合、並列実行しても結果が不正にならないことを保証するために、通常はデータの依存関係の解析を行い、細心の注意を払ってプログラムの変形や指示文の挿入を行う必要があります。しかし、コンパイラの自動並列化機能を使用すると、コンパイラオプション-P auto を指定するだけで、それらの作業をコンパイラが自動的に行います。

コンパイラは、まずプログラムを解析して並列実行可能なループや文の集まりを抽出し、次にそれらを複数の仕事に分割してタスクに割り当て、経過時間を短縮します。また、タスク間共有データ・タスク固有データの割付けも自動的に行います。コンパイラの自動並列化対象は、図 2.3 のとおりです。

対象となる構文	DO ループ、配列式
対象ループ中に書ける文	代入文、IF 構文、GOTO 文、CONTINUE 文、CALL 文、CASE 構文
対象となる演算	四則演算、べき乗演算、論理演算、関係演算、型変換、組込み関数

図 2.3 コンパイラの自動並列化対象

2.2 コンパイラの最適化

ここでは、共有並列化の効果を高めるためにコンパイラが行う最適化をいくつかご紹介します。

2.2.1 ループ変形

ループ融合・ループ一重化などのループ変形による最適化が可能な場合、最適化後に共有並列化を行い、共有並列実行時の仕事の粒度を最大化します。図 2.4 にループ融合の例を示します。

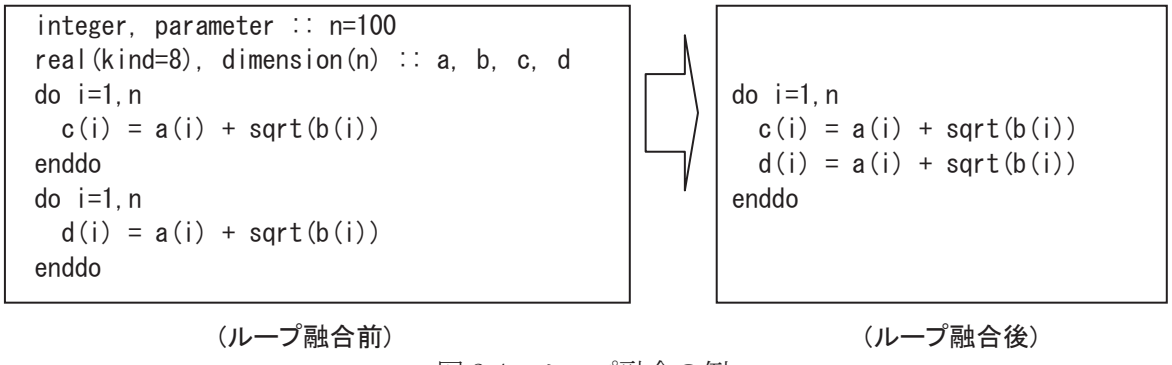

図 2.4　ループ融合の例

2.2.2 条件並列化

並列化できるかどうか又は並列化により高速化できるかどうかが翻訳時に判断できない場合、実行時に依存関係やループ長を調べて、共有並列化するかどうかを選択できるよう条件並列化を行います。図2.5では、条件並列化後のIF構文の論理式 nx*ny > n によって、共有並列化により高速化するのに十分なループ長があるかどうかを判定しています。この際、ループ中の各演算の演算コストに基づいて、共有並列化の効果が期待できる値(n)を自動的に算出し、条件並列化を行います。また、論理式 id-ic==0 .or. abs(id-ic)>=nx によって、ループ中で確定される配列要素 y(ic+i)と y(id+i)の領域に重なりがなく、並列化可能であることを判定しています。

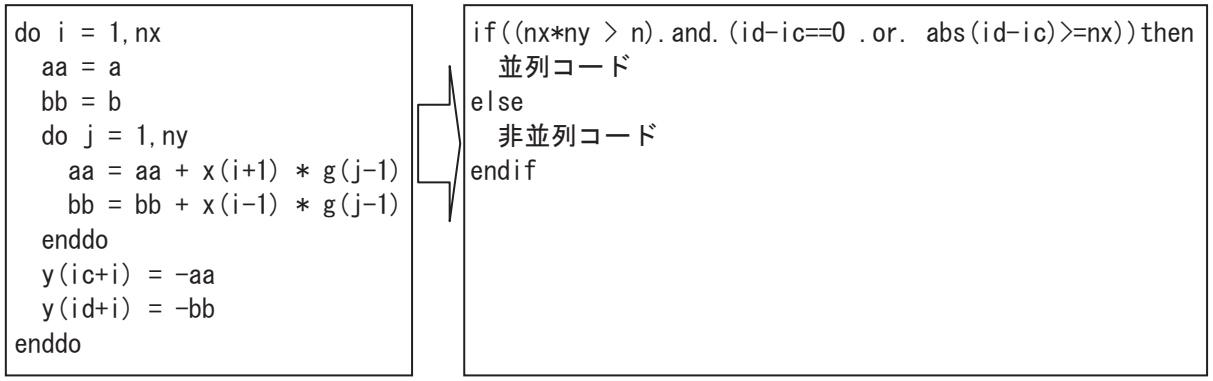

図 2.5　条件並列化の例

2.3 自動並列化促進のためのプログラミング

コンパイラオプション-P auto の指定だけでは自動並列化できない場合でも、コンパイラ指示行の挿入やプログラムの修正により自動並列化できる場合もあります。ここでは、主な並列化用指示行とプログラムの修正例をご紹介します。

2.3.1 並列化指示行

コンパイラ指示行の書式は、図2.6のとおりです。

Fortran の場合：

```
!CDIR コンパイラ指示オプション
```

C・C++の場合：

```
#pragma  cdir コンパイラ指示オプション
```

図 2.6　コンパイラ指示行の書式

自動並列化のために用意されている主なコンパイラ指示オプションは、次のとおりです。

- concur [{ by=整数 | for=整数 }] / noconcur
  直後のループを自動並列化の対象とする / しないを指定します。コンパイラ指示オプション concur に、省略可能なサブオプション by=整数 を指定すると、各タスクに割り当てるループの繰返し回数を指定できます。省略可能なサブオプション for=整数 を指定すると、ループをいくつのタスクに分割するかを指定できます。コンパイラ指示オプション noconcur は、粒度が小さく共有並列化すると

かえって性能が低下するループに指定します。

- inner / noinner
  内側ループ又は一重ループを自動並列化の対象とする / しないを指定します。多重ループの内側ループは、既定値では自動並列化の対象にならないので、共有並列化したい場合、コンパイラ指示オプション inner を指定します。一重ループは、共有並列化の効果が翻訳時に不明の場合、自動並列化の対象になりませんが、コンパイラ指示オプション inner を指定することにより自動並列化の対象とすることができます。

- nosync
  ループ中で参照される配列に依存がないことを指定します。依存関係が翻訳時には分からないので自動並列化できないときでも、依存がないことを利用者が分かっている場合、コンパイラ指示オプション nosync によりそれをコンパイラに教えてやることによって、自動並列化できる場合があります。

- cncall
  Fortran の組込み関数以外の手続引用を含むループは、既定値では自動並列化の対象になりませんが、並列化しても問題ないことを利用者が分かっている場合、コンパイラ指示オプション cncall を指定すると自動並列化の対象とすることができます。自動並列化されたループ中で引用される手続の仮引数は、原則としてタスク間共有データとなるので、そのような仮引数を手続中で確定した結果、図 1.3 のような依存関係が発生すると結果が不正となることに注意してください。

2.3.2 並列化のためのソース変形

図 2.7(a)の 2 重ループは、外側の do j のループには依存があるので並列化できませんが、コンパイラ指示オプション inner を内側の do i のループに指定すると、内側ループで自動並列化されます。ただし、内側ループを並列化すると、並列化のオーバヘッドが外側ループの繰返し回数分発生してしまいます。このような場合、図 2.7(b)のように内側ループと外側ループを利用者が交換すると、外側ループで自動並列化され並列化のオーバヘッドを削減できます。

```
integer, parameter :: n=10000,m=10000
real(kind=8), dimension(n,m) :: a
do j=1,m-1
   do i=1,n
      a(i,j) = a(i,j+1) + a(i,j)
   enddo
enddo
```

(a) ループ交換前

```
do i=1,n
 do j=1,m-1
   a(i,j) = a(i,j+1) + a(i,j)
 enddo
enddo
```

(b) ループ交換後

図 2.7　ループ交換の例

仮引数は原則としてタスク間共有データとなるので、図 2.8(a)のループネストは、タスク間共有データである仮引数 w に関するデータ依存により、外側の do j のループを自動並列化できません。このような場合、図 2.8(b)のように仮配列引数 w を次元拡張し、do j のループの繰返し毎に異なる要素を参照するように修正すれば、自動並列化可能となります。この際、仮引数 w に対応する実引数にも同様の修正が必要となることに注意してください。また、手続 sub の引用元が仮引数 w に対応する実引数を参照しておらず、配列 w は単なる作業変数として利用されている場合は、図 2.8(c)のように仮引数 w を手続 sub の局所変数

に変更すると、コンパイラによりタスク固有データとして割り付けられるので、自動並列化できます。

```
subroutine sub(a,b,w)
parameter(n=100,m=100)
real(kind=8) a(n,m),b(n,m),w(0:n)
do j=1,m
  do i=1,n
    w(i) = a(i,j) / dble(n)
  enddo
  do i=1,n
    b(i,j) = b(i,j) + (w(i-1)+w(i))/2.0
  enddo
enddo
```

(a) 仮引数 w に関する依存

```
subroutine sub(a,b,w)
parameter(n=100,m=100)
real(kind=8) a(n,m),b(n,m),w(0:n,m)
do j=1,m
 do i=1,n
   w(i,j) = a(i,j) / dble(n)
 enddo
 do i=1,n
   b(i,j) = b(i,j)+(w(i-1,j)+w(i,j))/2.0
 enddo
enddo
```

(b) 次元拡張後

```
subroutine sub(a,b)
parameter(n=100,m=100)
real(kind=8) a(n,m),b(n,m),w(0:n)
do j=1,m
  do i=1,n
    w(i) = a(i,j) / dble(n)
  enddo
  do i=1,n
    b(i,j) = b(i,j) + (w(i-1)+w(i))/2.0
  enddo
enddo
```

(c) 局所変数化後

図 2.8　仮引数の修正により自動並列化可能となる例

2.4 自動並列化の際のメモリ消費量

局所変数は、コンパイラによりタスク固有データとして割り付けられます。そのため、巨大な局所配列を宣言すると、その局所配列がタスク数個分割り付けられ、メモリ不足によりプログラム実行が失敗する場合があります。大きな配列は大域変数(Fortranの場合、モジュール変数又は共通ブロック変数)としてください。

クロス環境において、sxsize コマンドを、オプション-l に実行タスク数を指定した上で、実行ファイルに対して適用すると、図 2.9 のように、各タスクに対して確保されるスタックである logical task region のメモリ量を表示させ、確認することができます。

```
%> sxsize -f -l 4 a.out
11256208(.text) + … + 9714896(logical task region) * 4 = …
```

図 2.9　sxsize コマンド実行例 (4 タスク実行の場合)

2.5 OpenMP による共有並列化

コンパイラは、OpenMP による手動並列化も用意しています。OpenMP は、the OpenMP Architecture Review Board (ARB)[1] によって策定された共有並列化のためのオープンな API であり、共有並列化のデファクト標準として広く使用されています。OpenMP では、主として、ソースプログラムに OpenMP 指示文を挿入することによって共有並列化を行います。利用者は、OpenMP 指示文に必要に応じて指示句を追加し、並列化方法を詳細に制御できます。

SX-ACE で OpenMP を使用する場合、コンパイラオプション-P openmp を指定してください。

2.5.1 OpenMP 使用時の注意点

OpenMP と自動並列化は併用できないので、共有並列化したい部分には全て OpenMP 指示文を指定してください。

自動並列化の場合、コンパイラがプログラムを解析し並列ループ間の不必要な同期を自動的に抑制します。一方 OpenMP による共有並列化時には、nowait 指示句を指定しない限り並列ループの直後に毎回同期が生成されます。同期が必要ない場合には、nowait 指示句を指定すると共有並列化のオーバヘッドを削減できます。

また自動並列化の場合、図 1.4 のような作業変数や集計変数は、コンパイラによって自動的に認識され適切に処理されます。一方 OpenMP による共有並列化時には、DO 変数以外の作業変数及び集計変数をそれぞれ private 指示句及び reduction 指示句中に指定する必要があります。

OpenMP の parallel do 指示文をループに指定して共有並列化した例を図 2.10 に示します。繰返し内作業変数 t を private 指示句中に、集計演算子+ 及び集計変数 s を reduction 指示句中に指定していることに注意してください。なお、DO 変数 i は、この場合既定値でタスク固有データとなるので、private 指示句を指定する必要はありません。

```
integer, parameter :: n=100
real(kind=8) :: a(n), t, s
!$omp parallel do private(t) reduction(+: s)
do i=1,n-1
  t = a(i) + a(i+1)
  s = s + t
enddo
```

図 2.10　OpenMP による並列化例

2.6 性能解析

2.6.1 プログラム特性情報（PROGINF）

逐次（ベクトル）プログラムの場合と同様に、共有並列プログラムの場合も、プログラム特性情報（PROGINF）が使用できます。PROGINF は、プログラム実行時に環境変数 F_PROGINF（Fortran の場合）又は C_PROGINF（C・C++の場合）を、YES 又は DETAIL と設定することによって採取できます。図 2.11 に、環境変数 F_PROGINF を DETAIL に設定して共有並列実行した場合の出力例を示します。ここで図中の※は、共有並列実行時に固有の情報です。

共有並列プログラムの性能を分析する場合、PROGINF 中の Conc. Time に着目してください。図 2.11 の場合、プログラムは 4 個の CPU コア上でそれぞれ約 12.6 秒ずつ実行されたことが分かります。Conc. Time が CPU コア間でほぼ均等となっているので、うまく共有並列化されていると判断できます。逆に図 2.12 のように、Conc. Time (>= 1)だけ実行時間が大きく残りが小さい場合は、プログラムが十分共有並列化できていないか、又は負荷バランスが悪い、と判断できます。このようなプログラムは、より多くのループ

を共有並列化できるよう修正するか、又はコンパイラ指示オプション concur by=整数 などを指定して負荷バランスを均等化すると、経過時間を短縮できる可能性があります。

```
    ******  Program Information  ******
Real Time (sec)                :          12.763395
User Time (sec)                :          50.708166
Sys  Time (sec)                :           0.074710
Vector Time (sec)              :          40.632103
Inst. Count                    :       11790881218.
V. Inst. Count                 :        5299726673.
V. Element Count               :     1356647707664.
V. Load Element Count          :       24425310348.
FLOP Count                     :      687380352458.
MOPS                           :       26882.038333
MFLOPS                         :       13555.614543
MOPS   (concurrent)            :      107208.756514  ※
MFLOPS (concurrent)            :       54061.398206  ※
A. V. Length                   :         255.984467
V. Op. Ratio (%)               :          99.523808
Memory Size (MB)               :        4544.000000
Max Concurrent Proc.           :                4.  ※
  Conc. Time(>= 1) (sec)       :          12.714809  ※ 1個以上のCPUコアで実行した時間
  Conc. Time(>= 2) (sec)       :          12.713175  ※ 2個以上のCPUコアで実行した時間
  Conc. Time(>= 3) (sec)       :          12.688490  ※ 3個以上のCPUコアで実行した時間
  Conc. Time(>= 4) (sec)       :          12.591691  ※ 4個以上のCPUコアで実行した時間
Event Busy Count               :                0.  ※
Event Wait (sec)               :           0.000000  ※
Lock Busy Count                :                0.  ※
Lock Wait (sec)                :           0.000000  ※
Barrier Busy Count             :                0.  ※
Barrier Wait (sec)             :           0.000000  ※
MIPS                           :         232.524308
MIPS (concurrent)              :         927.334513  ※
I-Cache (sec)                  :           0.001616
O-Cache (sec)                  :           0.015225
Bank Conflict Time
  CPU Port Conf. (sec)         :           0.005777
  Memory Network Conf. (sec)   :           0.106487
ADB Hit Element Ratio (%)      :          66.689853
```

図 2.11　共有並列実行時の PROGINF

```
Max Concurrent Proc.           :                4.
  Conc. Time(>= 1) (sec)       :          74.154168
  Conc. Time(>= 2) (sec)       :           8.549322
  Conc. Time(>= 3) (sec)       :           8.292376
  Conc. Time(>= 4) (sec)       :           8.071275
```

図 2.12　Conc. Time の例

2.6.2 簡易性能解析機能(ftrace 機能)

逐次(ベクトル)プログラムの場合と同様に、共有並列プログラムの場合も、ftrace 機能によって手続単位又は利用者が指定した区間単位の詳細な性能情報が取得できます。

基本的な使用方法は逐次プログラムの場合と同様ですが、コンパイラは、共有並列化する各ループネストを切り出して、それらを独立した手続に変形します。そのため共有並列化された各ループネストは、ftrace の表示上は独立した一つの手続として表示されることに注意してください。切りだされた各ループネストは、ソースプログラム中の出現順に、元の手続名に加えて$1, $2,…とサフィックスが付いた名前の手続として表示されます。図 2.13 に、共有並列実行時の ftrace の表示例を示します。$のついた手続

gauss$1 及び cauchy$2 は、それぞれ手続 gauss の最初に出現する共有並列化されたループネスト、手続 cauchy の 2 番目に出現する共有並列化されたループネストの性能情報です。-micro1, -micro2, … は、各タスクの性能情報です。gauss のような$のついていない手続名は、手続 gauss 中の共有並列化されなかった部分の性能情報です。

```
PROG.UNIT FREQUENCY EXCLUSIVE      AVER.TIME   MOPS MFLOPS V.OP  AVER.  I-CACHE O-CACHE…
                    TIME[sec](%)   [msec]                 RATIO V.LEN  MISS    MISS  …
gauss$1      7872   2.536( 94.6)     0.322   4487.1 2099.2 99.41 233.7  0.0009 0.2196 …
 -micro1     1968   0.644( 24.8)     0.337   4306.4 2014.4 99.40 233.6  0.0000 0.0574 …
 -micro2     1968   0.616( 23.0)     0.313   4605.5 2154.8 99.42 233.8  0.0009 0.0541 …
 -micro3     1968   0.628( 23.4)     0.319   4522.0 2115.6 99.41 233.8  0.0000 0.0541 …
 -micro4     1968   0.628( 23.4)     0.319   4527.5 2118.2 99.41 233.8  0.0000 0.0540 …
cauchy$2        4   0.104(  3.9)    26.104    230.5   38.4  0.08 228.4  0.0000 0.0000 …
 -micro1        1   0.026(  1.0)    26.138    230.2   38.3  0.08 228.4  0.0000 0.0000 …
 -micro2        1   0.026(  1.0)    26.092    230.6   38.4  0.08 228.4  0.0000 0.0000 …
 -micro3        1   0.026(  1.0)    26.093    230.6   38.4  0.08 228.4  0.0000 0.0000 …
 -micro4        1   0.026(  1.0)    26.093    230.6   38.4  0.08 228.4  0.0000 0.0000 …
gauss           1   0.033(  1.2)    33.191   1246.9  514.9 96.90 226.3  0.0011 0.0030 …
                :
```

図 2.13　共有並列実行時の ftrace

3. 分散並列化

分散並列化とは、それぞれ独自のメモリ空間をもつ複数のプロセスによる並列処理のことです。SX-ACE では、一つ又は複数のノードの CPU コア上で実行されるプロセスに仕事を割り当て、分散並列化を行うことができます。

例として、図 1.1 の 2 重ループの分散並列化を考えます。第 1 章と同様に、外側の do j のループを四つの仕事に分割して別々のプロセスに割り当てます。この場合、データ領域に関しては、図 3.1 のように、2 次元配列 a, b を do j のループに対応する 2 次元目で分割し、分割された各部分領域の処理を各プロセスが担当することになります（分割配置・1 次元分割）。ここで、例えば j=25 の繰返しを担当するプロセス 0 は、プロセス 1 の担当領域である配列要素 a(i,26), (i=1,2,…,lx) の値を引用する、といったように、配列の分割境界部分の処理では、自プロセスの担当領域外の配列要素を参照する必要があります。ところが各プロセスは、他のプロセスに割り付けられた配列の領域を直接参照できないので、通信を行って必要な配列要素の値を取得しなければなりません。このように分散並列化の場合、共有並列化と異なり、他のプロセスの担当範囲のデータを参照するために、必要に応じて通信が必要です。一方 DO 変数 i, j については、図 3.2 のように各プロセスがそれぞれ専用の領域を割り付ける（重複配置）と、並列実行中、各プロセスは変数 i, j をそれぞれ独立に参照できるので通信の必要はありません。すなわち、分散並列実行時にどのような通信が必要となるかは、各データ要素をどのようにプロセスに割り付けるか（データマッピング）及び一つの仕事をどのように複数の小さな仕事へと分割してプロセスに割り当てるか（計算マッピング）に依存します。このように分散並列化においては、データマッピング、計算マッピング、及び通信の 3 点を全て考慮してプログラムする必要があります。

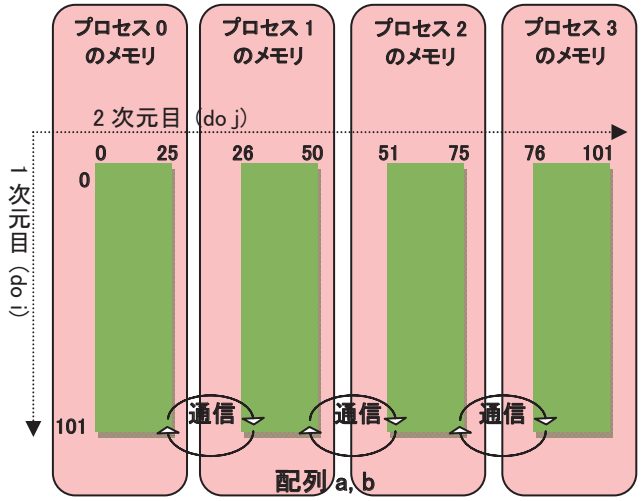

図 3.1　配列 a, b の分割配置（1 次元分割）

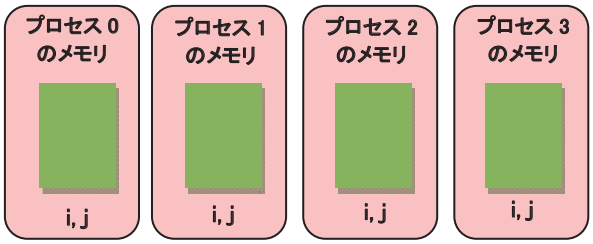

図 3.2　DO 変数 i, j の重複配置

3.1 データマッピングと性能

再び図 1.1 の 2 重ループの分散並列化を考えます。今度は図 3.3 のように、外側・内側のループをそれぞれ二つの仕事に分割してみます。この場合、データ領域に関しては、図 3.4 のように 2 次元配列 a,b を1次元目・2 次元目共に分割し、分割された各部分領域の処理を各プロセスが担当することになります（分割配置・2 次元分割）。ここでも、例えば j=50 の繰返しを担当するプロセス 0 は、プロセス 2 の担当領域である配列要素 a(i,51), (i=1,2,…,50) の値を引用する、といったように、配列の分割境界部分の処理では、自プロセスの担当領域外の配列要素を参照するために通信が必要です。

これを前節の外側ループだけによる分散並列化（1 次元分割）と比較してみます。通信に関しては、1 次元分割の場合、送受信回数はそれぞれ 2 回（2 次元目の上下方向）、送受信量はそれぞれ 1 次元目の寸法である lx*2 となります。一方 2 次元分割の場合、送受信回数はそれぞれ 4 回（1・2 次元目それぞれの上下方向）、送受信量はそれぞれ (lx/1 次元目分割数+ly/2 次元目分割数)*2 となります。従って、1 次元分割と比べると、2 次元分割の方が通信回数は多いですが、通信量はプロセス数が増えるほど少なくなります。1 回の通信時間は、典型的にはセットアップ時間+通信量/通信バンド幅と表せます。そのため、通信量が少なく（配列の寸法が小さく）セットアップ時間の影響が大きい場合には 1 次元分割の方が通信コストは少なく、逆に通信量が多く（配列の寸法が大きく）通信量の影響が大きい場合には 2 次元分割の方が通信コストは少なくなると考えられます。一方、各プロセスの演算性能を比較すると、ベクトル長は内側の do i のループに対応する配列の 1 次元目の寸法に依存します。そのため、1 次元目をプロセス間に分割することによってベクトル長が短くなりすぎると、ベクトル演算の効率が低下します。従って、配列の寸法がそれほど大きくない場合、2 次元目だけを分割したほうが演算時間は短い可能性があります。このように、これら二つの分散並列化方法のどちらが経過時間を短縮できるかは、プロセス数や配列の寸法に依存して変わります。分散並列化時には、通信コストや並列化後の演算性能を考慮して、データマッピングを決定することが重要です。

```
integer, parameter :: lx=100, ly=100
real(kind=8) a(0:lx+1,0:ly+1), b(0:lx+1,0:ly+1), k
do j=1,ly
  do i=1,lx
    b(i,j) = a(i,j) + k * (a(i+1,j) - 4.0D0 * a(i,j) + a(i-1,j) + a(i,j+1) + a(i,j-1))
  enddo
enddo
```

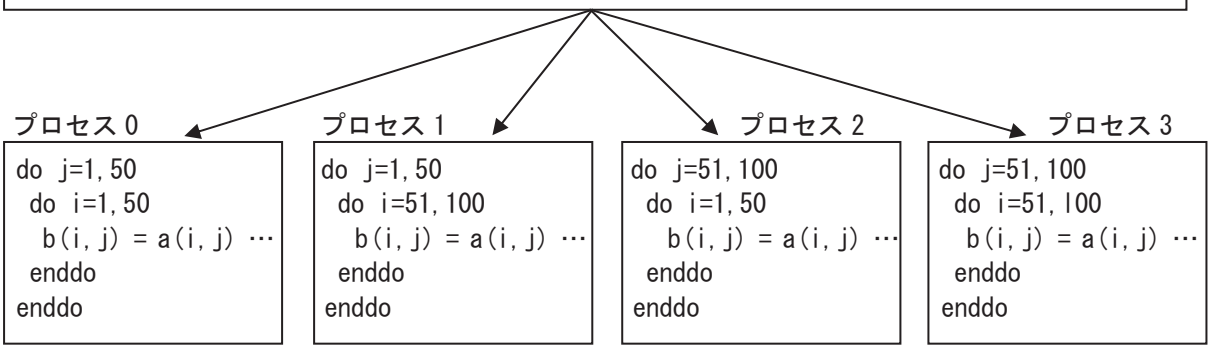

図 3.3　二つのループを共に並列化する例

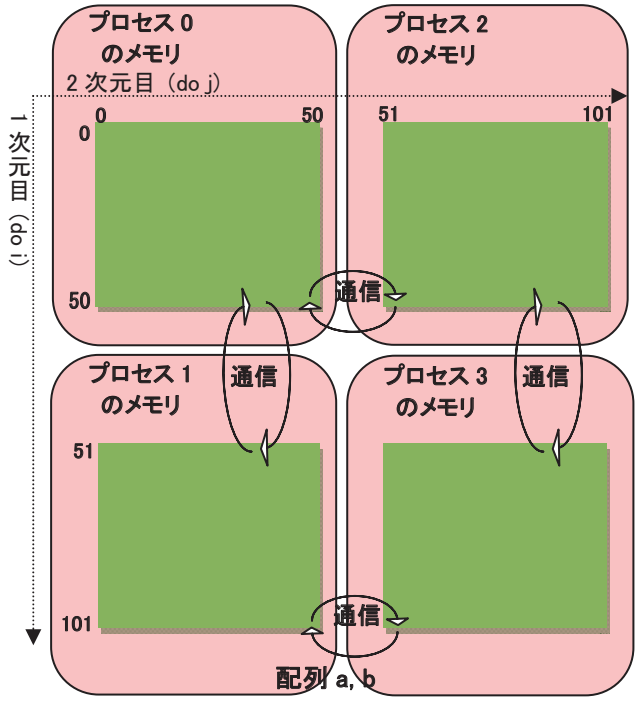

図 3.4　配列 a, b の分割配置(2 次元分割)

3.2 MPI

3.2.1 MPI 概要

MPI (Message Passing Interface) は、Message Passing Interface Forum[2] によって策定された通信ライブラリのオープンな仕様であり、分散並列プログラミングのデファクト標準として広く使用されています。MPI を使用した分散並列化では、図 3.1 や図 3.4 中の通信部分を、MPI 手続の引用によってプログラムすることになります。

分散並列化時に高いスピードアップを達成するには、通信コストの削減が特に重要です。マルチノード

システムの場合、一般に、ノード内のメモリアクセスコストと比較して、ノード間の通信コストは圧倒的に大きいからです。

SX-ACEは、異なるノードのメモリをIXSの命令により直接参照できるハードウェア機能(グローバルメモリ)などを活用した高速な通信環境を提供するため、MPI ライブラリ MPI/SX を用意しています。MPI/SXの1対1通信は、通信サイズに応じて最適な通信プロトコルを選択し、IXSの性能を最大限引き出しています。また MPI/SX の集団通信は、プロセスのノードへの配置方法や通信サイズに応じて最適な通信アルゴリズムを選択し、各ノードの共有メモリやIXSのバリア同期機構を利用した高速な通信を実現しています。

3.2.2 MPIプログラミングの基本

MPI プログラミングでは、まず MPI 仕様で規定されている各種定数などが定義されたヘッダファイルmpif.h (Fortranの場合) 又は mpi.h (C・C++の場合)をインクルードします(Fortranの場合、ヘッダファイルをインクルードする代わりに、システムモジュール mpi を引用することもできます)。次に、手続MPI_INIT 又はMPI_INIT_THREAD を引用して初期化を行い、手続MPI_FINALIZEを引用してMPIの使用を終了します。利用者は、この二つの引用の間で、1対1通信、集団通信、又は単方向通信により通信を行うことができます。

通信を行うプロセスの集合は、コミュニケータと呼ばれる引数を MPI 手続に渡すことにより指定します。プログラム開始時の全プロセスに対応するコミュニケータは、MPI の定数 MPI_COMM_WORLD で指定できます。各プロセスは、ランクと呼ばれるコミュニケータ中で一意な整数値をもちます。ランクの値は、0 以上、プロセス数-1 以下です。手続 MPI_COMM_SIZE 及び MPI_COMM_RANK により、それぞれプロセス数及び自プロセスのランクを取得でき、これらを利用して各プロセスが行う処理をプログラムできます。

MPI プログラムの典型的な例として、図 1.1 の 2 重ループを外側ループで四つの仕事に分割し、分散並列化したプログラムを図3.5に示します。MPIプログラミングでは、プログラム全体のデータ・処理ではなく、各プロセス用に分割した後のデータ・処理をプログラムします。そのため、定数 ly の値を 25 に変更して(文番号100)、配列の宣言(文番号200)とループ長(文番号300)を分割後の範囲に縮小しています。この際、通信時に受信したデータを配置する領域が必要なので、図3.6のように各プロセス上の配列aの領域を大きめに割り付けていることに注意してください。このような隣接プロセスとの通信のための領域は袖領域と呼ばれます。並列ループ(文番号300)実行前に、分割境界の処理に必要な配列aの要素を、1対1通信手続MPI_SENDRECVにより隣接プロセス間で送受信(文番号400)し、並列ループ実行中は、隣接プロセス上のデータの代わりに自プロセス上の袖領域を参照しています。両端のプロセスには片側の隣接プロセスが存在しないので、通信相手を設定する IF 構文(文番号 500)において、通信相手を示す変数 men, mep に、該当するプロセスが存在しないことを示す MPI の定数 MPI_PROC_NULL を設定しています。送受信先として定数 MPI_PROC_NULL を指定すると、その通信は実行されません。

```
      include "mpif.h"
100   integer, parameter :: lx=100, ly=25   ! ly は分割後の寸法
200   real(kind=8) a(0:lx+1,0:ly+1),b(0:lx+1,1:ly), k
   ! 初期化
      call mpi_init(ierr)
   ! プロセス数を変数 nproc に取得する
      call mpi_comm_size(mpi_comm_world,nproc,ierr)
   ! 自プロセスのランクを変数 me に取得する。
      call mpi_comm_rank(mpi_comm_world,me,ierr)
                :
   ! 通信相手の設定：両端のプロセスは特別処理を行う。
500   if(me .eq. 0)then
        men = mpi_proc_null  ! 下端のプロセスの下隣は存在しない
      else
        men = me - 1  ! 下隣のプロセスのランク
      endif
```

```
      if(me .eq. nproc-1)then
        mep = mpi_proc_null  ! 上端のプロセスの上隣は存在しない
      else
        mep = me + 1  ! 上隣のプロセスのランク
      endif
  ! 隣接プロセス間の通信：上方向 → 下方向
400   call mpi_sendrecv(a(1,ly),lx,mpi_double_precision,mep,0,                &  ! 上方向
     &     a(1,0),lx,mpi_double_precision,men,0,mpi_comm_world,mpi_status_ignore,ierr)
      call mpi_sendrecv(a(1,1),lx,mpi_double_precision,men,0,                 &  ! 下方向
     &    a(1,ly+1),lx,mpi_double_precision,mep,0,mpi_comm_world,mpi_status_ignore,ierr)
  ! 並列化後のループ
300   do j=1,ly
        do i=1,lx
          b(i,j) = a(i,j) + k * (a(i+1,j)-4.0D0*a(i,j)+a(i-1,j)+a(i,j+1)+a(i,j-1))
        enddo
      enddo
          :
  ! 後始末処理
      call mpi_finalize(ierr)
      end
```

図 3.5　MPI プログラム例

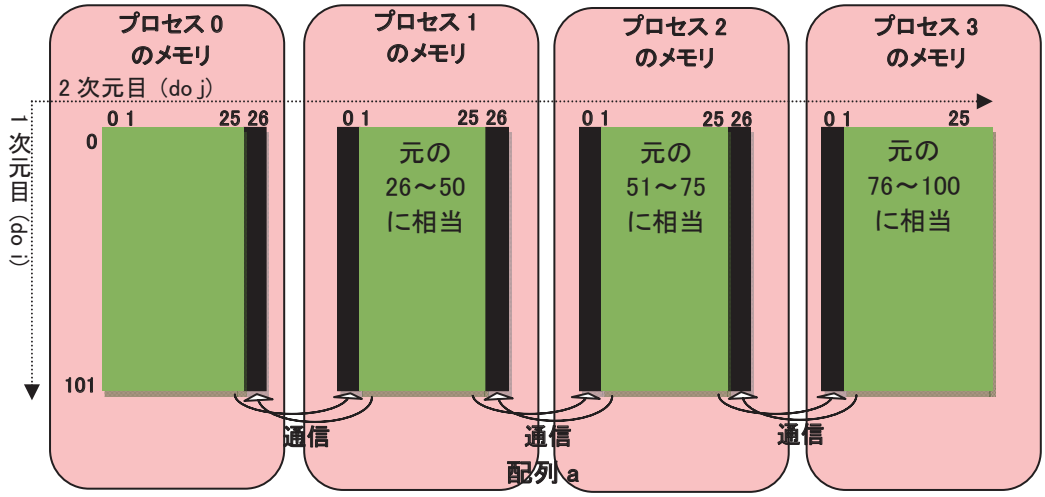

図 3.6　1 次元分割時の配列 a の割付け（黒色は袖領域）

3.3 MPI/SX の活用方法と注意点

3.3.1 MPI プログラムへの標準入力

MPI プログラムに標準入力から入力データを与える場合、図 3.7 のように、実行ファイルに対して直接入力データを与えると、正常に実行できない場合があります。

```
mpirun -np 4 -nn 4 /execdir/a.out < /datadir/inputfile
```

図 3.7　MPI プログラムへの不正な標準入力の例

このような場合、図 3.8 のように、コンパイラの環境変数 F_FF05 に入力ファイル名を設定して事前接続し、さらに MPI の環境変数 MPIEXPORT にその環境変数名 F_FF05 を設定した上で、MPI プログラムを実行してください。なお、MPI プログラム実行時にコンパイラの環境変数を使用するには、このようにコン

パイラの環境変数を設定した上で、環境変数 MPIEXPORT にその環境変数名を設定する必要があります。

```
setenv F_FF05 /datadir/inputfile
setenv MPIEXPORT "F_FF05"

mpirun -np 4 -nn 4 /execdir/a.out
```

図 3.8　MPI プログラムへの標準入力

3.3.2 プロセス毎に別々のファイルを対象にした入出力

環境変数 MPIRANK は、手続 MPI_COMM_RANK に引数 MPI_COMM_WORLD を渡して引用することにより得られるランクと同一の値をもっています。これを利用して、図 3.9 のようなシェルスクリプトを作成し（シェルスクリプトのファイル名を、例えば/execdir/run.sh とします）、図 3.10 のように、mpirun コマンドでそのシェルスクリプトを実行すると、各プロセスのランクと入出力ファイル名とを一対一に対応させ、プロセス毎に別々のファイルを対象にした入出力が可能です。

```
#!/bin/csh
setenv F_FF05 infile.$MPIRANK
setenv F_FF06 outfile.$MPIRANK
setenv MPIEXPORT "F_FF05 F_FF06"
exec /execdir/a.out
```

図 3.9　環境変数 MPIRANK を利用した入出力のためのシェルスクリプト例

```
mpirun -np 4 -nn 4 /execdir/run.sh
```

図 3.10　シェルスクリプトを介した MPI プログラムの実行例

プログラム中で接続したファイルに対して入出力する場合は、手続 MPI_COMM_RANK により取得したランクの値を利用してファイル名を決めると、やはりプロセス毎に別々のファイルを対象にした入出力が可能です

3.3.3 プロセス毎に別々のファイルへの標準出力・標準エラー出力

複数の MPI プロセスが同時に出力を行った場合、それらは入り交じって表示される場合があります。各プロセスの標準出力及び標準エラー出力を別々のファイルに容易に出力できるように、MPI/SX はシェルスクリプト/usr/lib/mpi/mpisep.sh を用意しています。このスクリプトを使用して、図 3.11 のように MPI プログラムを実行すると、環境変数 MPISEPSELECT の値に応じて、次のように各プロセスの標準出力及び標準エラー出力が別々のファイルに出力されます。なお下記の u は通常 0 となり、r は環境変数 MPIRANK の値となります。

- 環境変数 MPISEPSELECT の値が 1 の場合
  各プロセスの標準出力が、ファイル stdout.u:r に出力されます。

- 環境変数 MPISEPSELECT の値が 2 の場合（既定値）
  各プロセスの標準エラー出力が、ファイル stderr.u:r に出力されます。

- 環境変数 MPISEPSELECT の値が 3 の場合
  各プロセスの標準出力及び標準エラー出力が、それぞれファイル stdout.u:r 及び stderr.u:r に出力されます。

- 環境変数 MPISEPSELECT の値が 4 の場合
  各プロセスの標準出力及び標準エラー出力が、共にファイル std.u:r に出力されます。

```
mpirun -np 4 -nn 4 /usr/lib/mpi/mpisep.sh  /execdir/a.out
```

図 3.11　シェルスクリプト /usr/lib/mpi/mpisep.sh を使用した実行例

3.3.4 グローバルメモリを利用した高速化

転送元・転送先のデータを共にグローバルメモリ上に割り付け、MPI の基本データ型を使用することによって、通信を高速化できます。ただし、転送元・転送先データの先頭アドレス及び転送バイト数が、全て 8 の倍数である必要があります。

利用者のデータをグローバルメモリ上に割り付けるには、コンパイラオプション-gmalloc を指定した上で ALLOCATE 文を実行するか（Fortran の場合）、又は手続 MPI_ALLOC_MEM を引用して（C・C++の場合）メモリを割り付けてください。これらの手段で割り付けたメモリ領域の先頭アドレスは、8 の倍数であることが保証されています。

単方向通信（MPI_PUT 又は MPI_GET）において、転送元・転送先のデータが前述の条件を満たす場合、手続 MPI_WIN_FENCE の引数 assert に MPI/SX 独自の最適化情報 MPI_NEC_MODE_GETPUTALIGNED を指定すると、同期処理の高速化が可能です。また手続 MPI_WIN_CREATE によるウィンドウ生成時に、MPI/SX 独自キー"use_gbc"を"true"に設定した info 実体を引数 info に指定すると、同期処理をさらに高速化できます。

3.3.5 MPI プログラムデバッグ機能

MPI/SX は、実行時でなければ見つけにくい MPI プログラム特有のバグを検出する機能を用意しています。コンパイラオプション-mpiverify を指定すると、MPI 手続に対する引数のプロセス間の不整合や、配列引数への領域外アクセスの可能性などの情報が実行時に出力されます。

3.3.6 Fortran からの MPI 使用時の注意点

Fortran プログラム中で引用する非ブロッキングな MPI 手続の実引数として、部分配列、配列式、配列ポインタ、又は形状引継ぎ配列を指定すると、引数がメモリ上連続であることを保証するために、コンパイラが一時配列を割り付けて実引数をコピーする場合があります。このとき、実際の引数としては、その一時配列が MPI 手続に渡されます。しかし、その一時配列は、非ブロッキングな MPI 手続の完了以前に解放されてしまうので、プログラムが正常に実行されない可能性があります。非ブロッキングな MPI 手続の実引数には、メモリ上連続な配列を指定してください。配列ポインタや形状引継ぎ配列のように、翻訳時にはメモリ上連続であることが分からない場合、利用者がメモリ上連続であることを保証できるなら、コンパイラオプション-Wf"-cont"を指定すると、コンパイラによる一時配列の生成を抑制できます。

3.4 ハイブリッド並列化

複数のプロセスだけによるプログラムの分散並列化をフラット並列化と呼びます。SX-ACE では、分散並列化と第 2 章でご説明した共有並列化を併用することもできます。これをハイブリッド並列化と呼びます。

ハイブリッド並列化は、MPI プログラムを自動並列化又は OpenMP により共有並列化するだけで使用可能です。

図 3.5 では、図 1.1 の 2 重ループを外側ループで四つの仕事に分割し、4 プロセスに割り当てフラット並列化しました。これを、それぞれ 2 タスクからなる二つのプロセスに割り当てハイブリッド並列化するには、図 3.5 中の定数 ly の値を 50 に変更した上で、自動並列化のためのコンパイラオプション-P auto 及びタスク数を指定するコンパイラオプション-reserve=2 を指定し共有並列化します。フラット並列化の場合、配列 a を各プロセス上に図 3.6 のように割り付けました。一方、ハイブリッド並列化の場合、配列 a を各プロセス上に図 3.12 のように割り付けることになります。一つのプロセス中の複数のタスクは、そのプロセスの全データ領域を直接参照できるので、タスク間で通信を行う必要はなく、従ってタスク間には袖領域も必要ないことに注意してください。

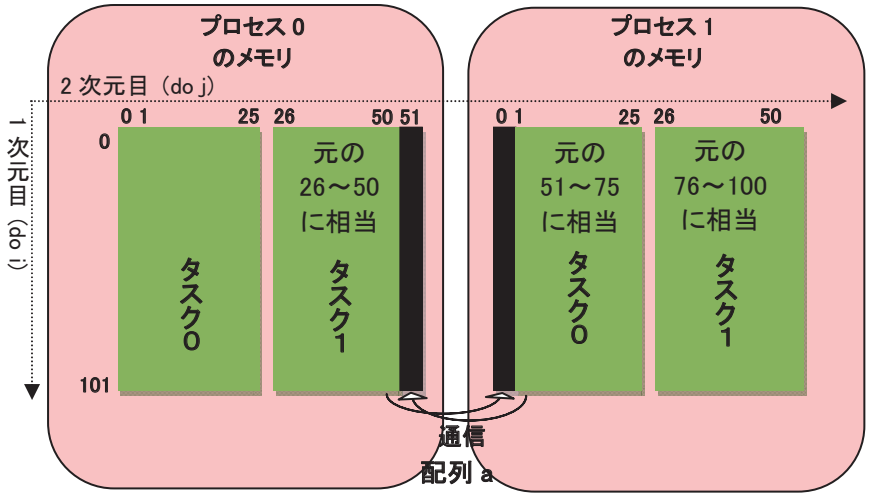

図 3.12　ハイブリッド並列化時の配列 a の割付け（黒色は袖領域）

一つの仕事を複数の小さな仕事に分割して、複数のプロセスに割り当てる場合（フラット並列化）と、各プロセス内の複数のタスクに割り当てる場合（ハイブリッド並列化）とを比較すると、ハイブリッド並列化によりプロセスの総数が削減されます。そのため、プロセス数に依存してコストが増大する通信を含むようなプログラムは、ハイブリッド並列化により経過時間を短縮できる場合があります。その反面、分散並列化と共有並列化の 2 重のオーバヘッドが発生するので、プログラムによってはハイブリッド並列化により経過時間が増加する場合もあります。一方、コンパイラ・MPI の処理系内部のバッファや、全プロセスに重複配置するデータの個数は、プロセス数の減少とともに減少するので、ハイブリッド並列化の方がフラット並列化よりも使用メモリの総量を削減できる場合があります。

3.4.1　ハイブリッド並列化時の注意点

自動並列化の場合、既定値では各プロセスがノード内の全 CPU コアを使用して共有並列実行を行います。そのため、ハイブリッド並列化時に一つのノード上で二つ以上のプロセスを実行すると、CPU コアの個数がタスクの個数より少なくなり、大幅に性能が低下する可能性があります。ハイブリッド並列化時には、各ノードのプロセス数を 1 にするか、共有並列化の並列数を指定する環境変数 F_RSVTASK（Fortran の場合）若しくは C_RSVTASK（C・C++の場合）を指定するか、又はコンパイラオプション-reserve=共有並列数 を指定して、各ノードにおけるプロセス数とタスク数の積が 4 以下となるようにしてください。

MPI/SX のタスクサポートレベルは、MPI_THREAD_SERIALIZED です。従って、全タスクが MPI 手続を引用できますが、手動で共有並列化する場合、同一プロセス中の複数のタスクが同時に MPI 手続を引用しないように利用者がプログラムする必要があります。

3.5 性能解析

3.5.1 MPI プログラム特性情報（MPIPROGINF）

MPI プログラムの実行性能を分析するために、MPI プログラム特性情報（MPIPROGINF）が使用できます。MPIPROGINF は、プログラム実行時に環境変数 MPIPROGINF を、YES，ALL，DETAIL，又は ALL_DETAIL に設定することによって採取できます。図 3.13 に、環境変数 MPIPROGINF を DETAIL に設定して実行した場合の出力例を示します。経過時間が最小のプロセス（Min）、最大のプロセス（Max）、及び平均（Average）の情報が表示されており、プログラム全体の性能を簡便に把握できます。

```
MPI Program Information:
========================
Note: It is measured from MPI_Init till MPI_Finalize.
      [U,R] specifies the Universe and the Process Rank in the Universe.


Global Data of 4 processes  :          Min [U,R]          Max [U,R]       Average
============================
  Real Time (sec)           :       56.829 [0,3]       56.879 [0,0]        56.855
  User Time (sec)           :       15.153 [0,2]       18.308 [0,0]        16.238
  Sys  Time (sec)           :        0.728 [0,3]        0.983 [0,0]         0.850
  Vector Time (sec)         :        3.765 [0,3]       10.689 [0,0]         6.807
  Inst. Count               :    721497042 [0,0]   1091361840 [0,2]    917296494
  V. Inst. Count            :     12671162 [0,3]     48775850 [0,0]     29139941
  V. Element Count          :    537593371 [0,3]   7704138496 [0,0]   4113854075
  V. Load Element Count     :            0 [0,0]            0 [0,0]            0
  FLOP Count                :          737 [0,2]      8400513 [0,0]      2102273
  MOPS                      :      104.355 [0,3]       517.423 [0,1]      296.505
  MFLOPS                    :        0.000 [0,3]         0.459 [0,0]        0.115
  A. V. Length              :       41.177 [0,2]       182.593 [0,1]      106.037
  V. Op. Ratio (%)          :       33.308 [0,2]        91.969 [0,0]       62.558
  Total Memory Size (MB)    :     1344.000 [0,0]      1344.000 [0,1]     1344.000
    Memory Size (MB)        :     1280.000 [0,0]      1280.000 [0,0]     1280.000
    Global Memory Size (MB) :       64.000 [0,0]        64.000 [0,0]       64.000
  MIPS                      :       39.408 [0,0]        72.020 [0,2]       57.512
  I-Cache (sec)             :        0.365 [0,0]         0.906 [0,2]        0.679
  O-Cache (sec)             :        0.539 [0,0]         1.134 [0,3]        0.786
  Bank Conflict Time
    CPU Port Conf. (sec)    :        0.000 [0,0]         0.000 [0,0]        0.000
    Memory Network Conf. (sec):      0.000 [0,0]         0.000 [0,0]        0.000
  ADB Hit Element Ratio (%) :        0.000 [0,0]         0.000 [0,0]        0.000
```

図 3.13　MPIPROGINF

3.5.2 簡易性能解析機能（ftrace 機能・NEC Ftrace Viewer）

コンパイラオプション-ftrace を指定すると、簡易性能解析情報ファイル ftrace.out.u.r（u は通常 0、r は環境変数 MPIRANK の値）が各プロセスから実行時に出力されます。MPI プログラムの場合、コンパイラによる簡易性能解析情報に加えて、通信時間などの情報が表示できます。

SX-ACE が用意する NEC Ftrace Viewer[3] は、並列プログラムの性能を分析する際にも有効です。例として、14 プロセスにより並列実行したある MPI プログラムの負荷バランスを解析してみます。まず、簡易性能解析情報ファイルを読み込んだときに表示される Process Breakdown Chart を見ると、図 3.14 のように手続 iterate 内の実行時間（Exclusive Time）が大きいことが分かります。実行時間が大きな手続 iterate

の負荷バランスを把握するため、左上の「Chart」メニューから Function Statistics Chart を選択します。すると、図 3.15 中の折れ線グラフが示すように、手続 iterate の Exclusive Time の標準偏差が大きく、負荷バランスが悪いことが分かります。状況をより詳細に調査するため、「Chart」メニューから Process Metrics Chart を選択します。次に、右上の「Metrics Selection」メニューから、MPI COMM. TIME（MPI 通信時間）及び MPI IDLE TIME（MPI 待ち時間）を選択します。すると、図 3.16 のように、MPI 通信時間及び MPI 待ち時間を示す二つの折れ線グラフがほぼ重なって見えます。これは、MPI 通信時間のほとんどが MPI 待ち時間であることを意味しています。また、棒グラフで示される手続 iterate の Exclusive Time と、折れ線グラフの MPI 待ち時間とを比較すると、Exclusive Time に占める MPI 待ち時間が、両端のプロセスだけ短いことが分かります。従って、両端のプロセスの MPI 待ち時間以外の実行時間、つまり演算時間が他のプロセスよりも大きく、相対的に演算時間の小さな他のプロセスにおいて、MPI による通信時に待ちが発生していると推定できます。以上のことから、両端のプロセスの演算を他のプロセスに分配すれば、負荷バランスを改善できる可能性があることが分かります。このように、NEC Ftrace Viewer のグラフ機能を使用すると、負荷バランスなど複数の仕事間の関係を容易に把握できます。

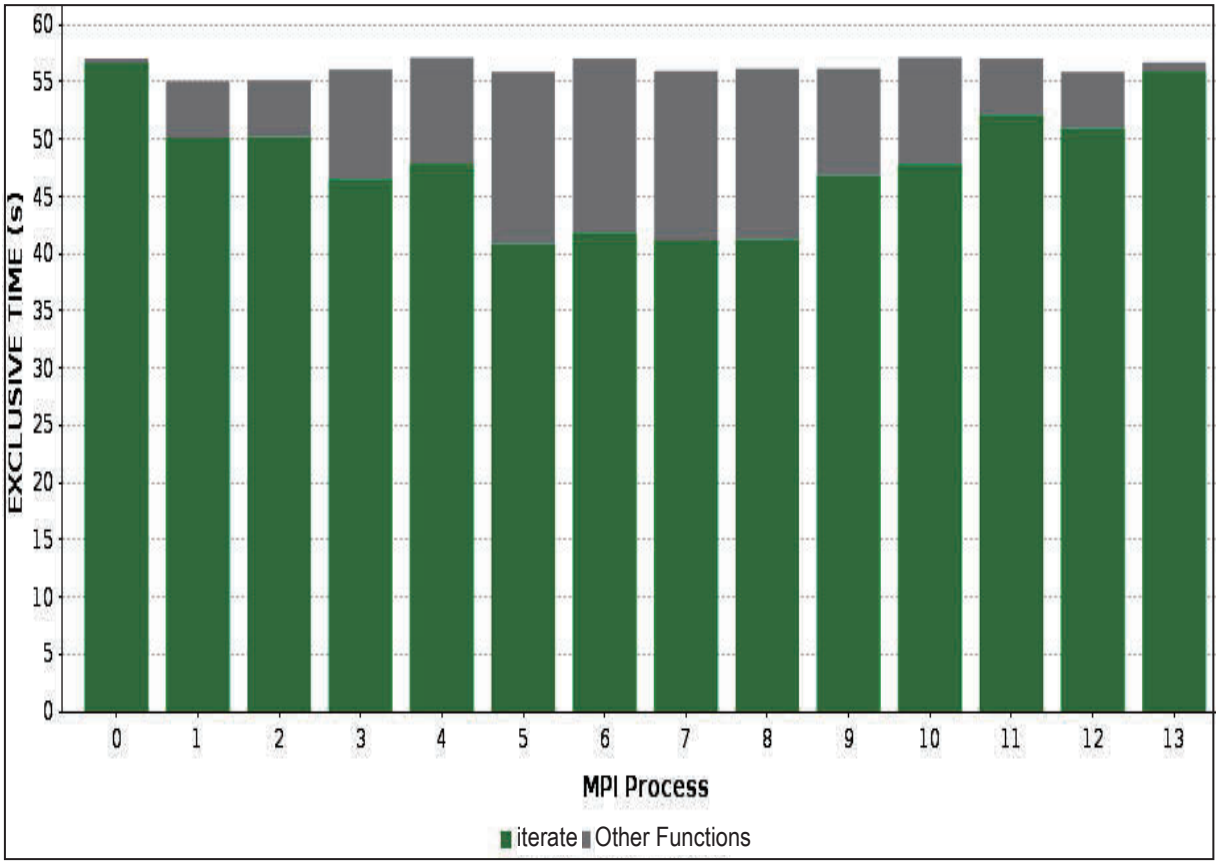

図 3.14　Process Breakdown Chart

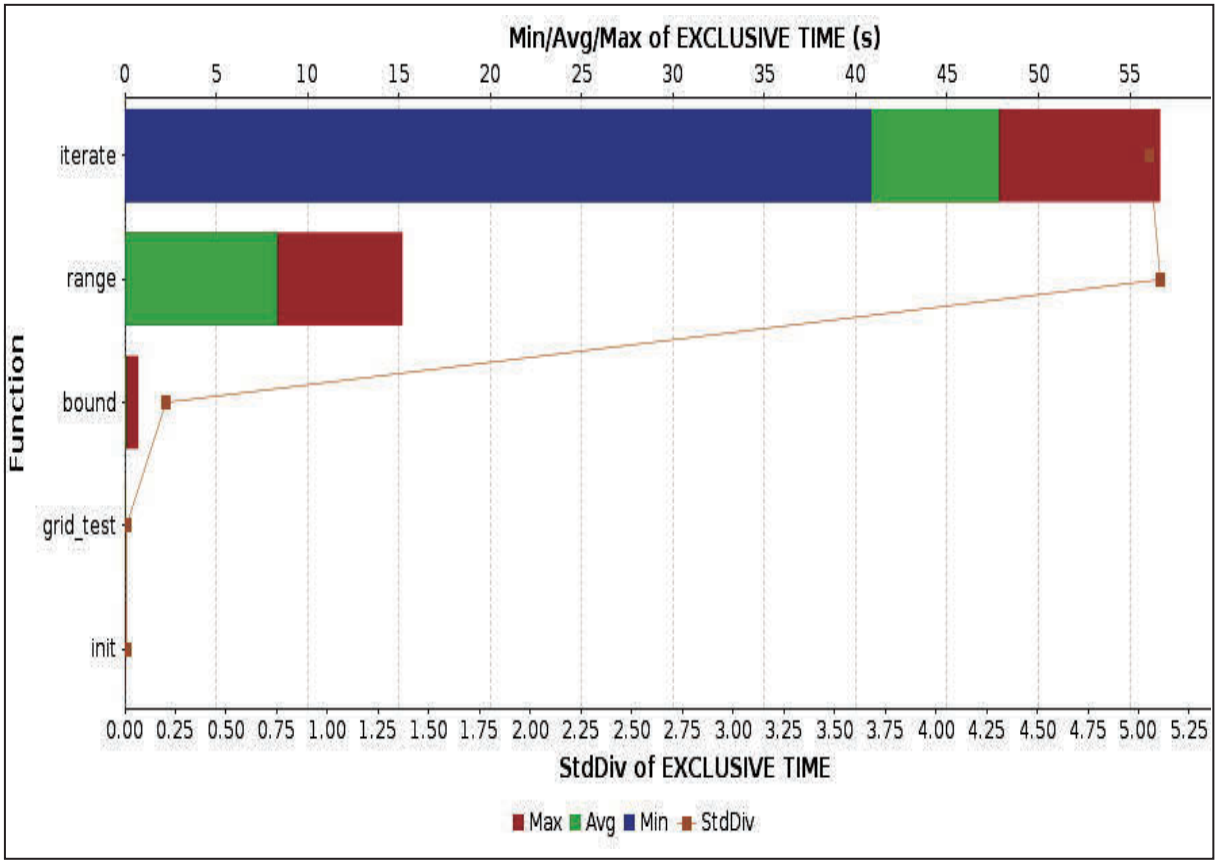

図 3.15　Function Statistics Chart

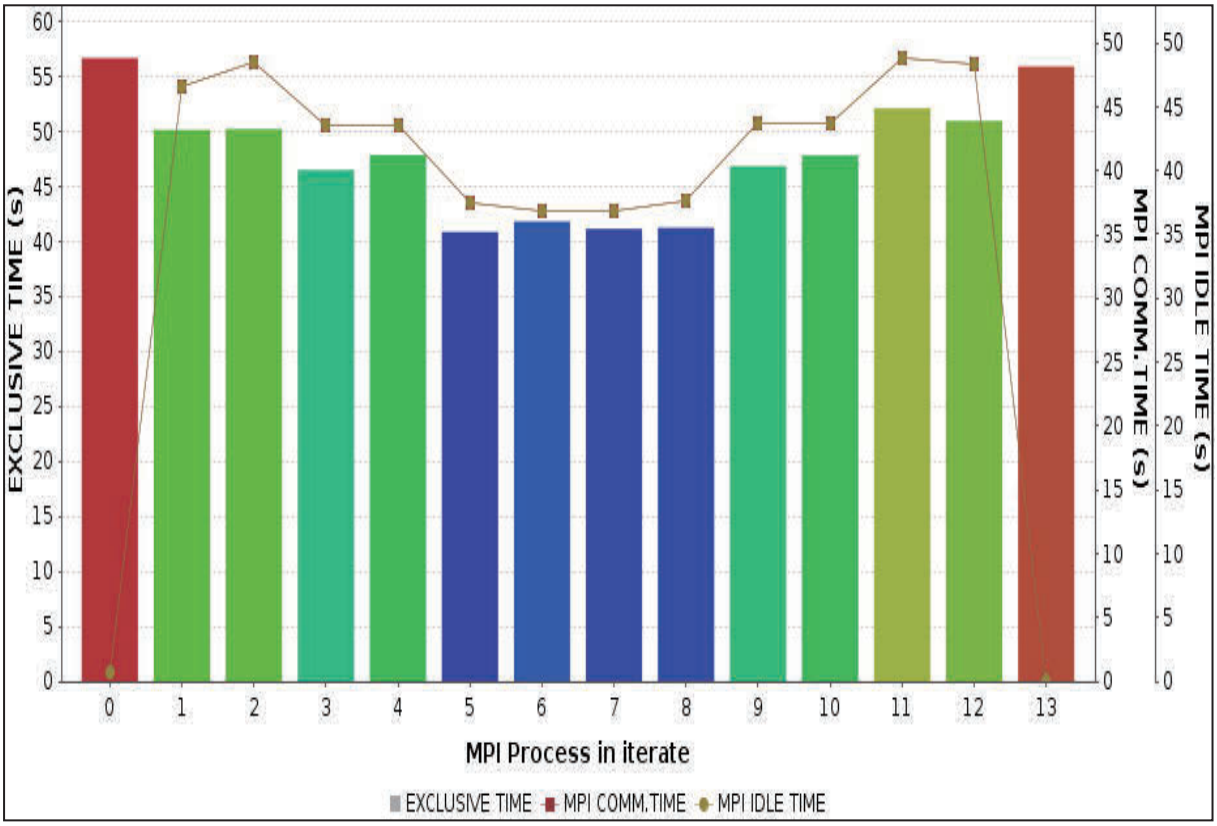

図 3.16　Process Metrics Chart

4. あとがき

本稿では、コンパイラの自動並列化機能及びMPI/SXを使用したプログラム並列化の基本的事項を中心にご紹介しました。SX-ACE が提供する並列処理環境の詳細な使用方法については、「FORTRAN90/SXプログラミングの手引」[4]、「FORTRAN90/SX並列処理機能利用の手引」[5]、「C++/SXプログラミングの手引」[6]、及び「MPI/SX 利用の手引」[7] をご覧ください。SX-ACE をご使用になる上で、本稿がお役に立てば幸いです。

参考文献

[1] The OpenMP Architecture Review Board (http://openmp.org/)
[2] Message Passing Interface Forum (http://www.mpi-forum.org/)
[3] NEC Ftrace Viewer 利用の手引，日本電気株式会社，G1AF33
[4] FORTRAN90/SX プログラミングの手引，日本電気株式会社，G1AF07
[5] FORTRAN90/SX 並列処理機能利用の手引，日本電気株式会社，G1AF08
[6] C++/SX プログラミングの手引，日本電気株式会社，G1AF28
[7] MPI/SX 利用の手引，日本電気株式会社，G1AF09