非均質計算環境向け並列化コンパイラ *hetero-TINPAR*

hetero-TINPAR: A Parallelizing Compiler for Heterogeneous Computing Environment

後藤 慎也 (*GOTO Shin-ya*)

平成9年2月14日

処理能力の異なる計算機が接続された非均質計算機環境では計算機の能力に応じたタスク分散が必要となる。本論文では，データを非均質に分散させることで計算能力に応じたタスク分散を行い，非均質計算機環境向けの並列コード生成を行う並列化コンパイラ *hetero-TINPAR* について述べる。

従来のような均質な並列計算機に対する並列化では，均等かつ規則的に分割したデータを各プロセッサに分散配置することにより，負荷の均衡を図ることができた。このため，分割されたデータを所有するプロセッサの判定や，各プロセッサにおいて実行すべきループのイタレーションの範囲を求めることは，プロセッサ番号を用いて容易に行えたため，高速な並列実行が可能であった。一方，非均質環境ではデータを各プロセッサの処理能力に応じて分散させなければならない。したがって，データの所有判定やイタレーション範囲の決定は複雑になり，単純な実装では大きなオーバヘッドをもたらしてしまう。

この問題を解決する方法として，本論文では均質計算環境である抽象プロセッサを用いたデータの非均質分散を提案する。この方法ではまず最初に抽象プロセッサに対して，従来と同様の規則的なデータ分散を行う。次に，実際のプロセッサに対して，抽象プロセッサを性能に比例した個数だけ割り付ける。データは抽象プロセッサに対しては均質に分散されているため，データから抽象プロセッサへの写像は従来のように簡単な式によって行うことができる。したがって抽象プロセッサから実プロセッサへの写像のためのテーブルを用意するだけで，非均質なデータ分散にもとづくデータの所有判定などが可能となる。

さらに本手法では非均質実行のオーバヘッドを，ループの実行をループスケジューリング部とループ実行部に分割することで大幅に削減する。ループスケジューリング部は，データのプロセッサに対する分散が決定した時点で各プロセッサが実行しなければならないイタレーションを求めリストの形にする。ループ実行部ではこのスケジューリングによって得られたリストに従って必要なイタレーションのみを実行する。本手法により，各プロセッサでは必要のないイタレーションが実行されず，またループ内ではデータの所有判定を行わないため高速な実行が可能である。またループスケジューリングも，抽象プロセッサに対してデータが均質に分散されていることを利用して少ないコストで実現している。

また，*hetero-TINPAR* では動的負荷分散機能を実装した。プログラムの実行中に各プロセッサの処理能力を測定することで理想的なデータの分散比率を予測し，この予測にとづいてデータの実プロセッサへの分散比率を動的に変化させる。このためユーザはプログラム起動時に各プロセッサの能力を把握している必要がなくユーザに対して非均質性の隠蔽が可能である。また，マルチユーザ環境下において他のユーザプロセスのために動的にプロセッサの処理能力が変動するような時にもデータ分散比率を負荷に応じて変化させることで柔軟かつ高速な実行を可能にしている。

性能評価の結果，本手法による非均質実行のオーバヘッドは従来の均質環境のための手法と比較してほとんど差がないことがわかった。また，このオーバヘッドは抽象プロセッサの粒度にはほとんど関係しないものであることも分かった。さらに，分散させるデータ量を計算機の能力に応じて分散させることで均質に分散させた時と比較して実行時間を約 2/3 に短縮することができた。的負荷分散機能により他のプロセスの影響を考慮した柔軟な実行ができ，負荷分散を行わない時と比較して 6 ∼ 25%の性能向上が示された。

In heterogeneous computing environment which consists of the processors with different computing power, tasks must be distributed in proportion to the computing power of processors. This paper presents *hetero-TINPAR*, a parallelizing compiler which generates optimized codes for heterogeneous computing environment.

In homogeneous environment, we can divide data elements regularly and distribute equally onto processors, therefore owner processor of data elements and iterations to be executed can be calculated with low overhead. On the other hand, in heterogeneous environment data elements must be distributed unequally and irregularly for load balancing, and such distribution causes large overheads in calculations of owner and iterations to be executed.

This paper presents a compiling technique to generate codes to execute efficiently with irregular and unequal distribution by use of "Abstract Processors," virtual homogeneous parallel processors. In this technique, data elements of arrays are once distributed onto homogeneous abstract processors. And then these processors are mapped onto real processors so that several abstract processors are assigned to a real processor in proportion to the available computing power of the real processor. Because data elements are regularly distributed onto abstract processors, projection from data elements to the abstract processors is performed with a simple calculation. Therefore, only a projection table from the abstract processors to the real processors is needed to decide owner of data elements.

To reduce overheads of executing loops in a heterogeneous environment, *hetero-TINPAR* divides a loop into two sections: *loop scheduler* and *loop executor*. Loop scheduler makes lists of iterations to be executed for each loop when the processor mapping is decided. By referring the iteration lists, loop executor executes necessary iterations. Since the loop executor never executes useless iterations and owner condition statements, loops can be executed with low overheads. Furthermore, utilizing regularity and homogeneousness of data distribution on abstract processors, loop scheduling can be done with small cost.

This paper also presents a dynamic load balancing technique, which is provided as a *dynamic load balancing library* in *hetero-TINPAR*. During the computation, available computing power of each processor is evaluated periodically and abstract processors are remapped to balance the computational loads among real processors. This library enables users to execute programs without any parameters which specify the computing power of heterogeneous processors. And it also makes programs flexible with dynamic load unbalance in multiuser environment.

As a result of the performance evaluation, it is concluded that the generated codes are effectively executed in the heterogeneous environment. The overheads dealing with heterogeneous data distribution could be kept in the same level with the codes generated by a compiler for homogeneous environment. The results also demonstrated that these overheads have no correlation with the number of abstract processors. The heterogeneous data distribution which reflects the computing power of processors reduced the execution time 33 ∼ 35% as compared with the homogeneous distribution. And as an effect of the dynamic load balancing library, the 6 ∼ 25% speed up of execution time is achieved in the loaded environment by other processes.

目次

1 はじめに

近年，ワークステーションやパーソナルコンピュータなどの安価で高性能な計算機が普及してきており，またこれらの計算機どうしは Ethernet やさらに高速な LAN によって接続されている。こういった計算機群の全体または一部を用いて仮想的な並列計算機を構築する試みがなされている。このようなシステムはスーパコンピュータと比較しても比較的安価に高性能の環境を構成できるといった利点がある。そこで，ユーザが容易にこのようなネットワーク環境を最大限利用できるような並列化コンパイラおよび並列プログラム実行環境の研究が進められている。

ところがこのような LAN 環境の多くが，さまざまな機種によって構成された非均質計算環境 (以下非均質環境と略す) であるため，アーキテクチャが異なったり，各プロセッサの性能が異なるといった問題がある。さらにマルチユーザ環境下では他ユーザプロセスなどの影響で各計算機の利用可能な計算能力が動的に変化する。

現在までの並列化コンパイラは，各プロセッサの性能が同一である均質な並列計算機を対象とするものが多い。均質環境では，コンパイラは全プロセッサに対し同量のタスクを処理させるようなコードを生成することで計算機の能力を引き出すことができた。しかし非均質環境では，動的に変化する各計算機の能力に応じたタスク量を計算機に対して与える必要があり，今までのような均等なタスク分配では最適な並列実行が望めない。そのためコンパイラに対して各計算機の負荷・性能の動的変化を考慮に入れた並列化が要求される。

本研究では，このような非均質環境において各プロセッサの処理能力に応じたタスク分配を行うための並列化コード手法を提案し，それにもとづく並列化コンパイラ *hetero-TINPAR*[1] を開発した。*hetero-TINPAR* は抽象プロセッサを介したデータ分散とループスケジューリングの最適化により，負荷とデータの非均質な分散に伴うオーバヘッドが極めて小さいコードを生成する。さらにこのコードに対して動的負荷分散を行うためのライブラリをリンクすることでプログラムを負荷変動に柔軟に対応させる。

以下，第 2 章では背景として本研究が対象とする非均質環境について述べ，さらに従来の均質計算環境向けのコード生成手法とその問題点を指摘する。第 3 章では *hetero-TINPAR* の概要について述べ，第 4 章において *hetero-TINPAR* の並列化手法，第 5 章で *hetero-TINPAR* による負荷分散機能について述べる。第 6 章で評価を行い第 7 章でまとめる。

2 背景

本章では，本研究でコンパイラのコード生成の対象とした非均質計算環境についてその特徴を述べる。次に従来の均質環境向けコンパイラの並列化手法を概説し，従来の並列化手法を非均質環境向けに適用した時の問題点を指摘する。

2.1 非均質計算環境

従来の並列計算機の多くは，全く同じ構成の要素プロセッサが格子やツリーなどの規則的なネットワークによって接続された均質計算環境 (Homogeneous Computing Environment) であった。しかし LAN 環境においてさまざまな計算機を接続した並列計算環境は，非均質計算環境 (Heterogeneous Computing Environment) と呼ばれる環境となることが多い。非均質環境は均質環境と異なり以下のような特徴がある。

プロセッサの非均質性　並列計算機を構成している各プロセッサ (計算機) が非均質である。具体的には以下のような非均質性が存在する。

- プロセッサ・アーキテクチャの非均質性:

  非均質環境では異なるプロセッサによって構成されることがある。特に実行オブジェクト形式やバイトオーダが異なる時には，プロセッサ間でのバイナリの互換性が保たれない。また，非均質環境を構成しているプロセッサ自体が並列計算機やベクトル計算機である場合には，レベルの異なる並列性を扱う必要がある。

- 記憶容量の非均質性:

  個々のプロセッサが所有している主記憶容量やキャッシュ容量が異なることがある。このような場合，同じ CPU であっても性能に差異を生ずる。

- 基本性能の非均質性:

  各プロセッサのアーキテクチャが異なれば当然計算能力は異なる。また同じアーキテクチャであってもクロック速度や記憶容量の違いによって性能が非均質になることがある。

ネットワークの非均質性　プロセッサ間を接続するネットワークが格子などのような規則的な網とならない。また，異なるネットワークを使用した場合にはネットワークのバンド幅，レイテンシが異なる。また，Ethernet などによって接続されている場合には，ネットワークの混雑度によりスループットが大幅に変動する。

マルチユーザ環境　従来の並列計算機の多くがシングルユーザ環境であったため，プログラム実行中はプロセッサの計算能力を占有することが可能であった。しかしネットワーク上の計算機の多くはマルチユーザ環境に対応した OS が動作する。このような環境では多くのプロセスが並行に動作しており，実行すべきプログラムはプロセッサ能力を他のプロセスと分けあわなければならない。またプロセスの生成や終了は動的であり，利用可能な計算能力は時間的に変動する。

本論文では特に，LAN 環境下で複数の性能の異なるワークステーションを接続したワークステーションクラスタを対象とした並列化の議論を行う。

ワークステーションクラスタを対象とした並列処理の研究としては [2, 3] などがあるが，用いられているワークステーションクラスタは同一機種のプロセッサが専用の高速ネットワークによって接続しており均質な並列計算機といってよく，従来の並列計算機と同様な並列化アプローチによって高速実行可能である。

本研究ではこのような特殊なワークステーションクラスタではなく，教室・研究室内などの比較的小規模な LAN 環境において異機種ワークステーションを接続したクラスタを対象とする。これは本研究の目的が，近年高性能になりつつあるワークステーションを用いて安価な並列計算環境の構築を行い，また負荷の低いワークステーションを並列計算機の要素プロセッサとして有効利用することにあるためである。LAN 環境では同機種のワークステーションによってクラスタが構成されていることは少なく，様々な世代・アーキテクチャのワークステーションによって接続された非均質環境となる。また，LAN 環境ではネットワークはイーサネットなど，ある程度均質な場合が多く，ネットワークの非均質性と比較してプロセッサ性能の非均質性の方がより性能を左右する要因となる。そこで本稿では特にプロセッサの非均質性の隠蔽および，マルチユーザ環境への対応を目的とした並列化手法について述べる。

2.2　非均質環境向けの並列化アプローチ

一般にプログラムを並列化する場合，次のような作業を行わなければならない。

1. プログラムの中から並列化が可能な部分を見つける
2. 並列化対象部分をいくつかのタスクに分割する
3. 各々のタスクをプロセッサに割り付ける

非均質環境では各プロセッサの処理能力が異なるため，特に 3. の各タスクのプロセッサに対する割り付け手法が最も大きな問題となる。

ユーザがこのような作業を行うことは一般に容易ではないため，並列性を陽に記述できる並列プログラミング言語の利用や，逐次プログラムから自動的または半自動的に並列化を行うための並列化コンパイルシステムの利用などのアプローチがとられる。

前者の例としては均質環境向けには OCore[4] などがある。また非均質環境向けには HeNCE[5] がある。HeNCE ではユーザがプログラムをタスクグラフとして記述し，処理系が実行時に計算機の負荷を考慮しながら各タスクのプロセッサに対する割り付けを行う。このようなアプローチでは，タスクが独立した形で表されており，各プロセッサは自律的に実行可能なタスクを性能に応じて処理することでタスクの非均質な分散が行われる。

後者の例としては科学技術演算分野を対象とした逐次プログラムの並列化がある。数値処理分野では Fortran などの逐次言語で書かれたプログラムが長年にわたり蓄積されており，また数値処理プログラムは並列性をある程度容易に抽出できるものが多いため，コンパイラを利用したアプローチに対する研究が盛んになされている。均質環境向け並列化コンパイラの例としては Fortran D コンパイラ [6] や多くの HPF[7] コンパイラおよびわれわれが開発している並列化コンパイラ TINPAR[8] などがある。

本論文で述べる並列化コンパイラ *hetero-TINPAR* も，後者のアプローチにもとづき非均質環境において高速な並列実行を行うためのコード生成を行うことを目的とする。並列化コンパイラによってタスクを並列化し非均質環境向けのコードを生成する際には，各プロセッサに適切にタスクを割り付ける処理が均質環境と比較して最も大きな問題となる。そこで以降では，タスクの割り付け処理を中心に従来の均質環境向け並列化手法について考察し，非均質環境に対し従来の手法を適用した場合の問題点を明確にする。

2.3　均質環境のための並列化手法

均質環境では並列計算機を構成する全てのプロセッサが同一の性能を持つ。このためデータ並列性を持つ数値計算などのアプリケーションでは，コンパイラは各プロセッサに割り付けられるデータ量を均等に分散させて実行するようなコードを生成することで，全プロセッサの性能を最大限引き出すことができる。以下，均質環境における並列化コンパイル手法について概説する。

プロセッサに対するタスク割り付け手法　数値処理プログラムでは，実行時間の大部分がループ中の配列演算に費される。このような演算では多数の配列要素に対して規則的に同一かつ独立した演算を行うといったデータ並列性を見いだすことができる。

そこでコンパイラはプログラムのデータ領域を複数のデータ領域に分割し，この分割されたデータ領域に対する処理を一つのタスクとする。分割されたデータは全プロセッサに対して分散配置され，Owner Computes Rule を用いてタスクのプロセッサに対する割り付けを行う。

Owner Computes Rule 分割された配列データを所有するプロセッサがそのデータに対する参照を行う文を実行するような SPMD(Single Program/Multiple Data) コードを生成する。代入文では原則として左辺のデータを所有するプロセッサに演算が割り付けられる。

データ分割方法　プログラム中の配列は図 1 に示すようなブロック分割やサイクリック分割といった手段により分割される。これらの分割法は，データ領域を各プロセッサに対して均等に分配できるため，均質環境においてプロセッサ性能を引き出すことが比較的容易である。ブロック分割は，全データ領域をプロセッサ数で等分する分割法であり，隣接した配列の要素間の処理が同一のプロセッサによって行われるため通信量が低減できる。サイクリック分割は，処理が配列の一部分に集中するような演算においてプロセッサ負荷の均衡化のために用いられる。

分割データの所有判定　Owner Computes Rule に従って並列化する場合には，分割された配列データを所有するプロセッサがどれであるのかを特定する必要があるため，添字からデータを所有するプロセッサを求める関数 $OWNER()$ が使用される。配列 $X()$ の分割次元の配列サイズを N，プロセッサ数を N_{PE} とするとブロック分割，サイクリック分割ではそれぞれ

$$OWNER_{block}(X(k)) = \left\lfloor \frac{k}{B} \right\rfloor \quad (\text{ただし } B = \left\lceil \frac{N}{N_{PE}} \right\rceil) \tag{1}$$

$$OWNER_{cyclic}(X(k)) = k \bmod N_{PE} \tag{2}$$

として求めることができる。OWNER 関数は規則的分割の下ではこのように 1 回の除算または剰余演算により簡単に求まり，並列化の上で非常に有利である。

通信はデータの送受信相手プロセッサをこの関数により求め，

```
send( OWNER( a(i) ), b(i) )
```

などのように行う。この例では `a(i)` を所有するプロセッサに `b(i)` を送信する。多くの場合で，通信相手はコンパイル時に求まり，またコンパイル時に決定できなくとも実行時に 1 回の除算または剰余演算を用いるだけで求めることができ，実行時のオーバヘッドは小さい。

ループ並列化　各プロセッサは，並列化対象ループの全てのイタレーションを実行する必要はなく，自分が所有するデータが参照されるイタレーションのみを実行すればよい。そこでループ変数の初期値・終了値を変更するなどしてできるだけデータ所有判定をしないようなループ並列化を行う。

ループの並列化の例を図 2 に示す。図 2(a) のループは，`NPE` 台のプロセッサ間でブロック分割されたデータに対して参照・代入を行う。

配列 a(),b() は `N / NPE` 要素づつ各プロセッサに分散される。まず，各プロセッサはこのループのうち自分が所有するデータを参照するようなイタレーションを求める。配列 `a(i)` をローカルに参照するイタレーションの集合を I_a，配列 `b(i+1)` をローカルに参照するイタレーションの集合を I_b とするとこれら 2 つのイタレーションの組み合わせにより，

1. a(i),b(i+1) を共に ローカル参照できるイタレーション $(I_a \cap I_b)$
   →a(i) に対する b(i+1) のローカルな代入
2. a(i) のみローカル参照可能なイタレーション $(I_a \cap \overline{I_b})$
   → $OWNER$(b(i + 1)) からの b(i+1) の受信および a(i) に対する代入
3. b(i+1) のみローカル参照可能なイタレーション $(\overline{I_a} \cap I_b)$
   → $OWNER$(a(i)) に対する b(i+1) の送信
4. a(i), b(i+1) ともにローカルには参照しないイタレーション $(\overline{I_a} \cap \overline{I_b})$
   →(何もしない)

```
real a(N),b(N)
processors PROC(NPE)
distribute a(BLOCK) onto PROC
distribute b(BLOCK) onto PROC
for i = 0, N-1 do
  a(i) = a(i)+b(i+1)
endfor
```

(a) 逐次コード

```
real a(N),b(N)

if( PID != 0 )then                               /*$\overline{I_a}\cap I_b$*/
  i = (N/NPE)×PID-1
  send( OWNER(a(i)), b(i+1) )
endif

for i = (N/NPE)×PID, (N/NPE)×(PID+1)-2 do /*$I_a\cap I_b$*/
  a(i) = a(i)+b(i+1)
endfor

if( PID != NPE-1 )then                           /*$I_a\cap\overline{I_b}$*/
  i = (N/NPE)×(PID+1)-1
  recv( OWNER(b(i+1)), tmp )
  a(i) = a(i) + tmp
endif
```

(b) 並列化されたコード

図 2: 逐次ループの並列化

といった文が生成される。ブロック分割の場合，配列要素 $a(si+t)(s,t$ は整定数$, s \neq 0)$ をプロセッサ `PID` 上でローカルに参照するイタレーションは

$$\frac{(\mathtt{N}\,/\,\mathtt{NPE}) \times \mathtt{PID} - t}{s} \leq i < \frac{(\mathtt{N}\,/\,\mathtt{NPE}) \times (\mathtt{PID} + 1) - t}{s} \quad (i \text{ は整数}) \tag{3}$$

と求めることができる [9]。したがって I_a，I_b はそれぞれ

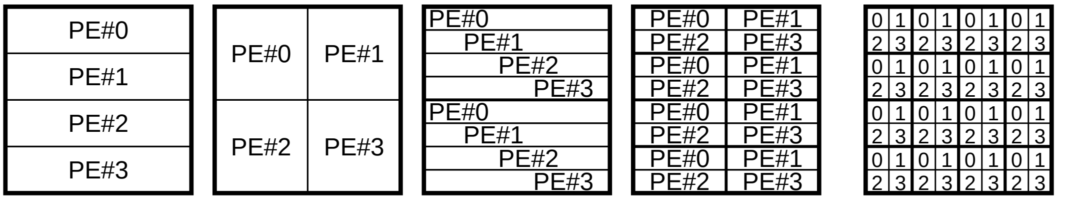
$$\begin{aligned} I_a &= \{i_a | i_a \text{は整数}, \\ &\quad (\mathtt{N}\,/\,\mathtt{NPE}) \times \mathtt{PID} \leq i_a < (\mathtt{N}\,/\,\mathtt{NPE}) \times (\mathtt{PID} + 1), \\ &\quad 0 \leq i_a < N\} \\ I_b &= \{i_b | i_b \text{は整数}, \\ &\quad (\mathtt{N}\,/\,\mathtt{NPE}) \times \mathtt{PID} - 1 \leq i_b < (\mathtt{N}\,/\,\mathtt{NPE}) \times (\mathtt{PID} + 1) - 1, \\ &\quad 0 \leq i_b < N\} \end{aligned}$$

と求まり，これらから

1. $\overline{I_a} \cap I_b = \{i | i = (\mathtt{N}\,/\,\mathtt{NPE}) \times \mathtt{PID} - 1\}$
2. $I_a \cap I_b = \{i | (\mathtt{N}\,/\,\mathtt{NPE}) \times \mathtt{PID} \leq i < (\mathtt{N}\,/\,\mathtt{NPE}) \times (\mathtt{PID} + 1) - 2\}$
3. $I_a \cap \overline{I_b} = \{i | i = (\mathtt{N}\,/\,\mathtt{NPE}) \times (\mathtt{PID} + 1) - 1\}$

と決定できる。求めたイタレーション集合によってループを分割し，ループ変数の初期値と終了値を適切に定めることにより図 2(b) のコードが得られる。

均質環境でデータを規則的に分割した場合，分散データを参照するようなイタレーションの集合が式 (3) のようにプロセッサ番号を用いた式としてコンパイル時に求められる。このため実行時には自分のプロセッサ番号をループの初期値・終了値の式に代入すれば，必要なイタレーションのみを実行することができる。したがってデータの所有判定のための計算はごくわずかであり，実行時のオーバヘッドを小さく抑えた高速なループ実行が可能となる。



(BLOCK,*) (BLOCK,BLOCK) (CYCLIC,*) (CYCLIC,BLOCK) (CYCLIC,CYCLIC)

図 1: データ分割法 (プロセッサ数 4)

```
real a(N/NPE),b(N/NPE)

if( PID != 0 ) then
  i = (N/NPE)×PID-1
  send( OWNER(a(i)), b(LOCAL_INDEX(i+1)) )
endif

for i = (N/NPE)×PID, (N/NPE)×(PID+1)-2 do
  a(LOCAL_INDEX(i)) = a(LOCAL_INDEX(i))
                          +b(LOCAL_INDEX(i+1))
endfor

if( PID != NPE-1 ) then
  i = (N/NPE)×(PID+1)-1
  recv( OWNER(b(i+1)), tmp )
  a(LOCAL_INDEX(i)) = a(LOCAL_INDEX(i)) + tmp
endif
```

図 3: アドレッシングの変更

アドレッシングの変更　図 2(b) のコードでは各プロセッサが全ての配列領域を確保しているが，実際には所有しているデータの分だけ領域を確保するだけでよい。均質環境ではデータが全プロセッサに等分されているため，各プロセッサが所有する配列領域を (1/プロセッサ数) となるように変更する。また，配列の参照も正しく変更されるよう，配列の添字から自プロセッサにおけるローカルなインデックスを求める関数 $LOCAL\_INDEX()$ を用いる。割次元の配列サイズを N，プロセッサ数を N_{PE} とするとブロック分割，サイクリック分割ではそれぞれ

$$LOCAL\_INDEX_{block}(k) = k \bmod \lceil N/N_{PE} \rceil$$

$$LOCAL\_INDEX_{cyclic}(k) = \lfloor k/N_{PE} \rfloor$$

という簡単な式により求めることができる。これを利用して図 2(b) のコードから図 3 のコードを生成する。

2.4　非均質計算環境における並列化の問題点

均質環境では全プロセッサの処理能力は同一であったためにタスク量も均等に分散することができた。このためデータをブロック分割やサイクリック分割といった分散手法で均等に割るだけで高いパフォーマンスを得ることができた。

非均質環境では各プロセッサの処理能力が異なるため，高速な並列実行を行うためにはプロセッサの処理能力に合わせて与えるタスク量を変化させ，プロセッサの非均質性を隠蔽する必要がある。つまり Owner Computes Rule にもとづいたタスク並列化を行う場合にはデータを各プロセッサの能力に比例した分散比で分散させなければならない。しかしデータを非均質に分散させるような並列化は，従来の均質環境向けの並列化と比較して以下のような問題が生ずる。

```
for i = 0, n - 1 do
  if( OWNER(a(i)) = PID ) then
    if( OWNER( b(i+1) ) = PID ) then
      /* a(i) も b(i+1) も所有 $(I_a \cap I_b)$*/
      a(i) = a(i) + b(i+1)    /* ローカルな代入 */
    else
      /* a(i) のみ所有 $(I_a \cap \overline{I_b})$*/
      recv( OWNER( b(i+1) ), tmp)
      a(i) = a(i) + tmp    /* b(i+1) を受信し代入 */
    endif
  else
    if( OWNER(b(i+1)) = PID ) then
      /* b(i+1) のみ所有 $(\overline{I_a} \cap I_b)$*/
      send( OWNER(a(i)), b(i+1) )   /* b(i+1) を送信 */
    endif
  endif
endfor
```

図 4: 非均質環境向けの単純なループ並列化

- 非均質なデータ領域を扱うことによる実行時のオーバヘッド: データを非均質に分散させる場合，均質環境において適用できた規則的な分散が不可能となる。このため，データとデータを所有するプロセッサの関係を求める OWNER 関数は均質環境のように簡単な式では求まらない。

  従って各プロセッサの性能比がコンパイル時に決定していたとしても，実行すべきイタレーションの特定や通信相手プロセッサの決定をコンパイル時に行い，生成コードに反映させることは非常に困難である。また，実行時に各プロセッサの分散比を変化させようとする場合には，静的にこれらを決定することは不可能である。

  非均質環境のための単純な並列化方式としてテーブル参照を用いる手法がある。各プロセッサに配列と同じ大きさのテーブルを持たせ，コンパイル時または実行開始時に各配列要素ごとにそれを所有するプロセッサ番号を与えておく。実行時にはこのテーブルを参照しながら実行すべきイタレーションや通信相手プロセッサの特定を行う。

  しかしこのような方法では，プログラム中のデータアクセスのたびにテーブル参照が必要となり実行時のオーバヘッドが問題となる。特にイタレーションの実行は，図 4 のように分割されたデータに対する参照の前にプロセッサがそのデータを所有するかどうかのテーブル参照を行わなければならないため，実質的に全てのイタレーションを実行してしまい，データを参照しないイタレーションの実行コストが全てオーバヘッドとなる。

- プロセッサに割り付けるタスク量の決定の困難性: 非均

質環境において各プロセッサの処理能力比があらかじめ判っていれば，各プロセッサに対して割り付けるタスク量を実行開始時に決定することができる。しかし各プロセッサに対して与えるべきタスク量をあらかじめ求めるためには，ベンチマークの実行や様々なタスク量による試行を行わなくてはならず，ユーザにとって非均質環境が利用しやすいものとはならない。

- 動的な負荷変動: マルチユーザ環境では各プロセッサの利用可能な計算能力が動的に変化する。このため実行開始時に静的にプロセッサのタスク処理能力比を見積り，与えるタスク量を固定してしまうことは適切ではなく，従来のような固定的なデータ分散を行うことができない。
- アーキテクチャ・通信インタフェースの非互換性: 非均質環境下では，そのアーキテクチャおよび通信インタフェースが異なることがある。このため並列プログラムを作成する時に各プロセッサ毎にアーキテクチャに依存したプログラミング手法や通信ライブラリを使用してやらなければならず，コード作成上の障害となる。また，アーキテクチャにより扱えるデータ型やバイトオーダが異なることがあり，通信時にデータが変化しないようにしなければならない (Data Coercion)[10]。

以上述べたうち，動的な負荷変動問題に関してはマルチユーザ環境下の均質環境においても問題となり，また通信インタフェースの非互換性に関しても複数の均質計算機に対してコンパイラを開発する際に指摘される問題でもある。しかし，その他の問題点は非均質環境に特有であり [11]，非均質環境向けコンパイラを開発する上で適切な対応が求められる。

3 *hetero-TINPAR* の概要

前章では非均質環境における並列化の問題点について述べた。本研究ではこれらの問題を解決し，非均質環境において高速な並列実行を行うような並列コードを生成するための並列化コンパイラ *hetero-TINPAR* の開発を行った。

本章では，最初に *hetero-TINPAR* の設計目標を示し，次に非均質環境向けに実行を高速化するために抽象プロセッサおよび抽象プロセッサを用いた実行方式を提案し，最後に *hetero-TINPAR* の内部構成を示す。

3.1 *hetero-TINPAR* の設計目標

hetero-TINPAR は，非均質環境において高速実行が可能となる並列コード生成を行うためのコンパイル手法・実行環境の研究プラットフォームとして位置付けられる。*hetero-TINPAR* では第 2.4 節で述べた非均質環境における並列化の問題に対応するべく以下の基本方針を採用した。

プロセッサの非均質性/動的負荷変動に柔軟に対応する

プロセッサの処理能力が異なるため，各プロセッサに対して処理能力に比例したタスク量が分散できるようにし処理能力の非均質性に対応する。また，マルチユーザ環境下で動的に変動しうる処理能力にも対応できるよう動的負荷分散機能を実装する。また，ユーザが実行開始時にプロセッサの処理能力が分からなくても自動的に最適な分散比を予測し非均質実行を行う機能も実装する。

タスク分散の非均質性に起因するオーバヘッドを低く抑える

各プロセッサに対して非均質にタスクを分散させることにより，均質環境と比較して非均質性に起因した実行時オーバヘッドが生ずることになる。本手法では，このオーバヘッドをできるだけ低く抑えるようなコード生成を実現する。

均質環境向き並列化コンパイラの最適化技術を利用する

現在までに均質環境において並列化手法および多くの実行・通信最適化手法が提案されてきた。これらの最適化手法のなかには非均質環境においても十分な効果をもたらすものが多い。そこで *hetero-TINPAR* ではこのような最適化手法を有効に利用することができるよう，従来の並列化コンパイラを拡張した形で実装することとした。このため，均質環境向け並列化コンパイラが多く採用してきた Owner Computes Rule にもとづくタスク並列化方針を採用し，出力コードも SPMD 型の並列コードとする。

本研究では特に，われわれが開発してきた均質環境向け並列化コンパイラ TINPAR の拡張という形で設計を行う。

任意に分割された配列データを任意のプロセッサへ分散できるようにする

各計算機に対して処理能力に比例したデータを分散させることで非均質なタスク分散を行うが，データ空間を任意の比率・形状で分割しプロセッサに分散できるように設計を行い，動的負荷分散時のタスクの量変更に伴うデータ移動を容易にする。

以上の設計目標を達成するためには，非均質環境において効率よくタスクを分散させる手法が必要となる。そこで，従来の並列化手法からの容易な拡張性および低コストで非均質実行を実現するために抽象プロセッサによる並列実行方式を提案する。

3.2 抽象プロセッサ

hetero-TINPAR では，異なる処理能力を持つプロセッサに対してその処理能力に適した量のタスクを分散させることで負荷の均衡化を図り高速実行を行う。このため均質環境の時のような規則的な分割によるプロセッサ割り付けはできない。

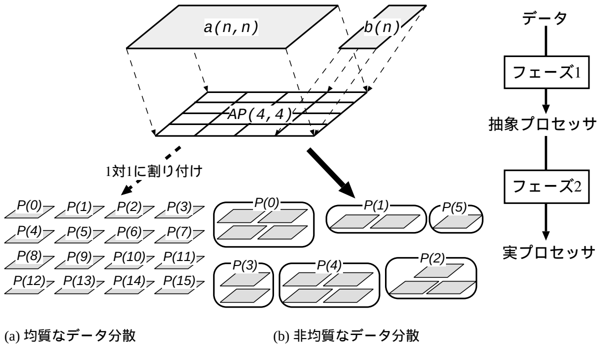
ところがブロック分割・サイクリック分割以外の任意の分割方法を許した時にはデータを所有するプロセッサの判定が簡単な式では求まらず，並列化が困難になる可能性がある。本手法では均質環境におけるデータ分散の規則性を非均質環境においても利用するべく，仮想的な抽象プロセッサ (Abstract Processors) を定義する。また抽象プロセッサに対し，最終的なコード生成の対象となる実プロセッサも定義する。

抽象プロセッサ: タスク処理能力が等しい要素プロセッサが多次元格子状に配列した仮想的並列計算機を想定し，その要素プロセッサを抽象プロセッサと呼ぶ。データは全てこの抽象プロセッサ上に均等に割り当てられる。抽象プロセッサ数および次元数はユーザがディレクティブの形で与える。

実プロセッサ: コンパイラが最終的にコード生成の対象とする物理的なプロセッサとする。実プロセッサのタスク処理能力に応じた個数の抽象プロセッサが割り付けられる。

hetero-TINPAR では抽象プロセッサの概念を用いることにより，非均質環境における並列化を

1. データを均質な抽象プロセッサに対して分散させる$\cdots$(フェーズ 1)



(a) 均質なデータ分散 (b) 非均質なデータ分散

図 5: 抽象プロセッサによる非均質分散

2. 均質な抽象プロセッサを非均質な実プロセッサに割り付ける···(フェーズ 2)

という 2 段階に分割して考える (図 5)。すなわちプログラムの並列実行の単位であるタスクを均質なものとし，均質なタスクを性能に応じて非均質なプロセッサに割り付けることで非均質実行を行う。本手法では，プロセッサの非均質性はタスクの割付けにのみ影響し，データのタスクへの分割は非均質性を考慮しなくてよいため，データを直接非均質なプロセッサに分散させる手法と比較して並列化の複雑度を下げているといえる。

また，フェーズ 1 に関しては均質環境向けの並列化を行うだけであるため従来の均質環境向きコンパイル手法をそのまま適用できる。フェーズ 2 についても，従来の均質環境は非均質環境の特殊ケース (抽象プロセッサ 1 個が実プロセッサ 1 個に割り付けられる) であるとみなすことができ，両者を一元的に取り扱うことができるといった利点がある。

抽象プロセッサ数および次元の決定は逐次コードに対するディレクティブ

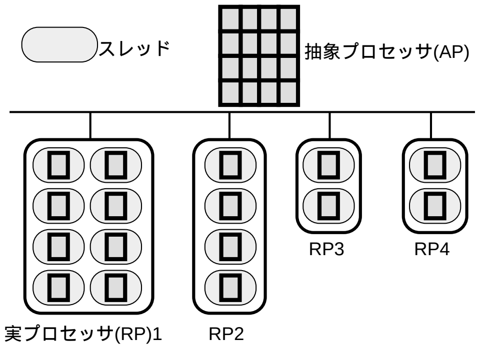
processors AP(N_{AP_1} , N_{AP_2} , ..., N_{AP_N})

としてユーザが与える形となっており，HPF における抽象プロセッサ (Abstract Processors) 宣言と同一のものとしている。本手法では抽象プロセッサが並列実行の最小単位となるため，抽象プロセッサ数によってタスクの粒度をコントロールすることができる。抽象プロセッサ数を配列サイズと同じにした場合には，完全に非均質な実行を行うとができ，逆に抽象プロセッサ数を実プロセッサ数に合わせた場合，同じタスク量が全実プロセッサに対して割り付けられてしまい非均質実行が不可能となる。

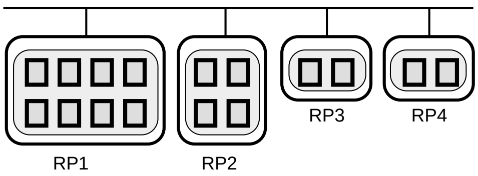
本手法ではさらに，実プロセッサに対して任意の抽象プロセッサを割り付けることができるよう設計した。抽象プロセッサの割り付け手法の自由度を高くすることで，多次元の抽象プロセッサでも実プロセッサに対して性能比に正確な割り付けができ，また動的負荷分散も容易に行うことができる。

3.3 実プロセッサにおける抽象プロセッサの実行戦略

hetero-TINPAR では，複数の抽象プロセッサが一つの実プロセッサに対して割り付けられるため，各実プロセッサは複数の抽象プロセッサに対するタスクを処理する。このとき，各タスクを独立して扱うかどうかによってマルチスレッド方式とシングルスレッド方式という 2 つのタスク処理方式が考えられる。本節ではこれら 2 方式の得失について議論し，最後に本手法で採用した実行方式について述べる。



(a)マルチスレッド方式



(b)シングルスレッド方式

図 6: 実プロセッサへの分散方式

マルチスレッド方式 図 6(a) のように，個々の抽象プロセッサをスレッドとして実プロセッサ上に分散させる。各実プロセッサはそれぞれの抽象プロセッサを独立して扱い，OS または実行処理系によるスケジューリングによって抽象プロセッサのタスクを実行する。

同じ実プロセッサ内の複数の抽象プロセッサどうしの通信は，スレッド間の通信として実現する。この方式では各抽象プロセッサの独立性が高く，各スレッドは均質な抽象プロセッサの処理を行うことができるため，コンパイラは均質な処理を行うようなスレッドを生成すればよい。

シングルスレッド方式 図 6(b) のように，各実プロセッサに対してスレッドを一つだけ起動し，各スレッド内で複数の抽象プロセッサに対する処理を行うためのコードを生成する。シングルスレッド方式では，同じ実プロセッサ内の複数の抽象プロセッサ間通信はスレッド間通信を用いなくともよい。また実プロセッサは実行時にスレッド切り替えを行う必要がなく，低オーバヘッドの実行が可能となる。しかし，スレッド内で各抽象プロセッサの実行すべきデータ領域に対する処理のスケジューリングを適切に行わないとデッドロックが生ずる可能性がある。このためコンパイラは不規則なデータ領域を動的に扱えるようなスレッドの生成が必要である。

本手法では非均質実行スケジューリング方式として，シングルスレッド方式を採用した。これは次の理由による。

- 既存の OS をそのまま利用する場合，シングルスレッド方式によって 1 スレッド内で非均質なデータ量を処理するためのオーバヘッドは，OS の介在するマルチスレッド方式のスケジューリングのオーバヘッドと比較して一般に小さい。本手法では，さらにループを適切にスケジューリングすることで非均質なデータを処理することにより生ずるオーバヘッドを低減している。ループスケジューリングについては 4.2 節で述べる。

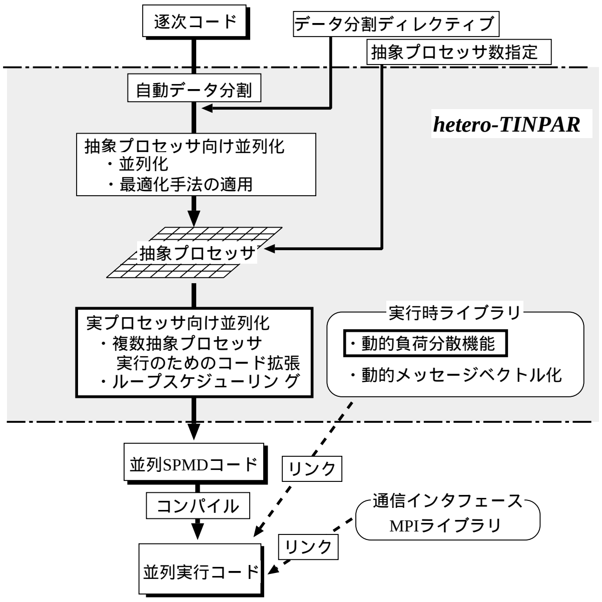


図 7: *hetero-TINPAR* の構成

- マルチスレッド方式では，同一の実プロセッサに割り付けられた複数の抽象プロセッサ間のデータ移動はスレッド間通信を用いなければならず，ローカルな代入と比較して余分な時間がかかる。特に抽象プロセッサの粒度を小さくしたときにはスレッド間通信量が増加してしまう。ところが，シングルスレッドではこのような移動は全て代入文とすることができるため非均質分散の粒度に関わらずオーバヘッドがおさえられる。

3.4 *hetero-TINPAR* の内部構成

hetero-TINPAR の内部構成を図 7 に示す。*hetero-TINPAR* は逐次コードを並列化するコンパイラ本体と動的負荷分散機能を実現するための実行時ライブラリから構成された，統合的な並列化コンパイラシステムとして開発されている。

入力逐次コードは TINPAR と同様 Fortran タイプの言語である Tiny Language[12] に対してデータ分割ディレクティブやプロセッサ指定ディレクティブを加えたものである。ただし *hetero-TINPAR* ではプロセッサ指定ディレクティブは実プロセッサの指定ではなく，抽象プロセッサの次元数およびプロセッサ数の指定がなされているものとして扱う。また，実行中に動的再分散機能を行うためのディレクティブも記述される。

コンパイラは出力としてプロセッサに依存しない SPMD コードを生成する。実際のオブジェクトコードの生成は，この SPMD コードを各アーキテクチャ毎にコンパイルさせ，同じくアーキテクチャ毎にコンパイルされた実行時ライブラリとのリンクによって行われる。

コンパイラの処理手順　*hetero-TINPAR* では，コード生成は以下の手順により行われる。

1. データ分割の決定

   逐次コードに対しディレクティブまたは自動データ分割決定 [13] によってデータ分割方法が与えられる。このデータ分割は抽象プロセッサに対して分散させるためのものであり実プロセッサは全く考慮に入れない。

2. 抽象プロセッサに対する並列化

   抽象プロセッサに対する分割が与えられると均質環境と同様なプロセスで並列化処理やコードの最適化処理が行われる。最適化処理は均質環境向け TINPAR において開発された手法 [9] のうち，通信最適化に関する以下の手法が用いられる。

   - 不要な通信の削除および 1 対多通信の `broadcast` 文への変更 [14]:

     データフロー解析を行い実際には使用されないデータを送受信する通信文を求め，削除することで通信回数を削減する。また，1 プロセッサから他の全プロセッサに対する `send/recv` はブロードキャストに変更し高速化を図る。

   - `send` 文のリスケジューリング:

   - `send/recv` 文をともに含むループ分配:

     以上 2 種は，`send` 文と `recv` 文の間の時間的距離を長くするために，`send` 文の実行をできるだけ先行させるようなコードを生成するための最適化手法である。

   - `msg_buffer_flush` 文の挿入:

     TINPAR では動的メッセージベクトル化ライブラリによって動的なメッセージベクトル化を行う [15]。この動的にベクトル化されたメッセージを最適なタイミングで送信させるための命令をコードに対して挿入する。

   なお，TINPAR による最適化手法のうち第 2.3 節で述べたループ並列化に用いられるものに関しては本手法では適用しない。これは本手法ではループ並列化が従来と全く異なった方針で行われるためである。

   本手法では以上述べたように，この段階で従来の均質環境向け並列化処理や多くの最適化手法が適用できるため，従来の並列化コンパイラからの拡張性が非常に高いものとなっている。

3. 実プロセッサに対する並列化

   抽象プロセッサに対する並列化処理がなされた後，複数の抽象プロセッサを実プロセッサに対して割り付けるための並列化処理がなされる。抽象プロセッサに対して生成された均質環境向けコードに対して以下の処理を行う。

   (a) データを所有するプロセッサ判定の非均質環境向け拡張

   (b) 非均質データをアクセスするループのスケジューリング

   (c) 非均質に分散されたデータに対するアドレッシングの変更

   これらの詳細については第 4 章で述べる。また，動的負荷分散を行うためのコードの挿入も行う。

実行時ライブラリ　*hetero-TINPAR* では実行時ライブラリとして動的負荷分散ライブラリを提供する。動的負荷分散ライブラリでは，非均質環境において動的に変動する負荷をモニタリングし性能予測および理想的なデータ分散比を求め，これに従ったデータ移動を行う。この動的負荷分散機能の実装によりプロセッサの動的負荷に柔軟な実行が可能である。動的負荷分散機能については第 5 章で述べる。

また，TINPAR において開発されたメッセージの動的ベクトル化機能を実現した通信ライブラリも実行時にリンクされる。

メッセージ通信ライブラリ　実際のメッセージ通信のためのライブラリとして MPI[16] を採用した。MPI を使用することにより Data Coercion も含めて全てのプロセッサのメッセージ通信インタフェースを統一でき，計算機間の通信インタフェースの非互換性の問題が解消できる。また，個々の計算機における通信の高速化・最適化はそれぞれの機種の MPI ライブラリの最適化にまかせられるため，*hetero-TINPAR* ではより一般的な通信最適化手法を適用するだけでよい。

以下，第 4 章では本節で述べた *hetero-TINPAR* の機能のうち，核となる非均質環境向けコード生成手法について述べる。さらに第 5 章では，*hetero-TINPAR* の大きな特徴である動的負荷分散を実現するための動的負荷分散機能について詳細を述べる。

4　並列コード生成手法

本章では非均質環境において不規則に分散されたデータを高速に並列実行するためのコード生成手法について述べる。第 3.4 節で述べた通り，非均質データを扱うための並列コード生成は，抽象プロセッサに対して並列化および一般的な最適化手法が適用された後のコードを用いて行われる。以下，次に示すような非均質な抽象プロセッサ割り付けに対応させるためのコード生成手法について述べる。

1. データを所有するプロセッサ判定の非均質環境向け拡張
2. 非均質データをアクセスするループのスケジューリング
3. 非均質に分散されたデータに対するアドレッシングの変更

4.1　データの所有プロセッサ判定

4.1.1　概要

hetero-TINPAR では各実プロセッサに対して複数の抽象プロセッサを割り付けるため，分割されたデータを所有する実プロセッサ番号は

1. アクセスすべきデータを所有する抽象プロセッサを求める
2. その抽象プロセッサが割り付けられている実プロセッサを求める

という 2 段階の処理によって求める。データは抽象プロセッサに対して均質環境と同様に規則的な分割がなされているため，1. の段階では従来の $OWNER()$ 関数 (以降，実プロセッサに対するデータ分散と区別するため $AP\_OWNER()$ とする) を用いて容易に行うことができる。抽象プロセッサ番号から実プロセッサ番号を求める 2. の処理は，不規則な割り付けが行われているためテーブル参照を用いる。

一般に不規則なデータ分散では各データがどのプロセッサに所有されているかが簡単な式では求まらないため，全ての配列要素あるいは配列要素の部分集合から実プロセッサへのインデックスを設け，インデックス参照によりプロセッサを特定しなければならない。しかし本手法では抽象プロセッサという均質なプロセッサを導入することでインデックスサイズを抽象プロセッサ数のオーダにまで抑え，記憶領域の節減や間接参照によるオーバヘッドの低減がなされている。

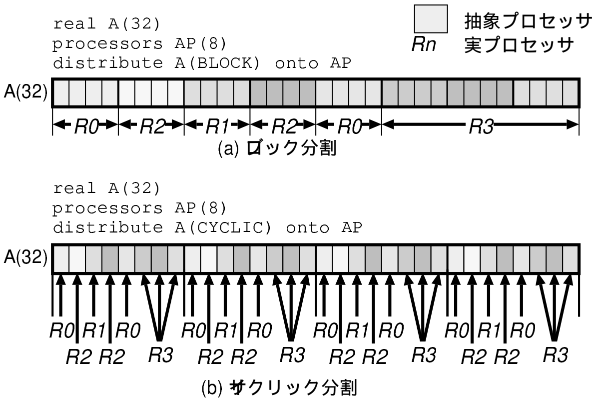


図 8: 抽象プロセッサによる実プロセッサ割り付け

さらにこのインデックスの内容はコンパイル時に決定する必要はなく，このインデックスの値の変更により自由に実プロセッサ割り当てを変更することができる[1]ため，これを利用して動的負荷分散が実現されている。

4.1.2　手順

抽象プロセッサとそれが割り付けられた実プロセッサとの対応関係を，インデックス配列 $APtoRPindex()$ として全抽象プロセッサに持たせる。例えば

$$APtoRPindex() = \{0, 2, 1, 2, 0, 3, 3, 3\}$$

であれば，実プロセッサ 0 番は $0, 4$ 番の抽象プロセッサ，実プロセッサ 2 番は $1, 3$ 番の抽象プロセッサといった形で割り付けられていることを示す (図 8)。このインデックスを用いることで特定のデータ $data$ を所有する実プロセッサ番号 $RP\_OWNER(data)$ を，

$$RP\_OWNER(data) = APtoRPindex(AP\_OWNER(data)) \quad (4)$$

として求める。具体的には，抽象プロセッサ向けに並列化されたコードに対して以下の処理を行う。

文の実行判定　抽象プロセッサ AP_ID 上でデータ a(i) に対する演算を行うかどうかの判定のための条件式

```
if( AP_OWNER(a(i)) = AP_ID ) then
   a(i) = ...
endif
```

は，該当抽象プロセッサが割り付けられた実プロセッサ上で処理を行えるよう実プロセッサ番号 RP_ID および式 (4) を用いた

```
if( RP_OWNER(a(i)) = RP_ID ) then
   a(i) = ...
endif
```

---

[1] 実際には実プロセッサ間のデータ移動および第 4.2 節で述べるループスケジューリングも同時に行う。

という形に変更する。なお，ループ中に含まれる実行判定は第 4.2 節で示すループスケジューリングにより行う。

通信相手の特定 抽象プロセッサが割り付けられた実プロセッサと通信を行うように通信相手を変更する。例えば，データ`a(i+1)`をデータ`a(i)`を所有する抽象プロセッサに対する送信コード

`send(` $AP\_OWNER($`a(i)`$)$`, a(i+1) )`

に対して式 (4) を用いて

`send(` $RP\_OWNER($`a(i)`$)$`, a(i+1) )`

と変更を行う。

4.2 ループスケジューリング

4.2.1 概要

多くのプログラムでは実行時間の大部分がループの実行となるため，ループスケジューリング手法が実行性能に大きな影響を及ぼす。

従来の均質環境では実プロセッサに対して規則的なデータ分割を行うことができたため，プロセッサ番号から各イタレーションにおけるデータアクセスの有無が容易に求められ，ループ実行のオーバヘッドをかなり低く抑えられた。

hetero-TINPAR ではシングルスレッド方式による非均質なタスク実行を行う。このため各実プロセッサは複数の抽象プロセッサの所有するデータに対する処理を 1 つのスレッドとして連続的に処理する。すなわち，1 つのスレッドで複数の抽象プロセッサの実行すべきイタレーションを矛盾なく実行する必要がある。また，抽象プロセッサの実プロセッサに対する割り付けは負荷分散により動的に変化してしまう。このため配列データを参照するループでは，各プロセッサはどのイタレーションを実行すればよいか，各イタレーションにおいてどのような通信を実行するかはループの実行時にならなければ確定しない。しかし図 4 のようにループ中のデータ参照の度に文の実行判定を行うと，実際には実行する必要のないイタレーションの実行が行われたり，実行するイタレーション内でも多くの条件判定が行われたりするため実行オーバヘッドが非常に大きい。

hetero-TINPAR では 1 つのループを，ループ内で実行すべきイタレーションを求めるループスケジューリング部と，実際にループの実行を行うループ実行部に分離し，実行時のデータ所有判定をループスケジューリング部でまとめて行うことで実行時オーバヘッドの低減を図る。

ループスケジューリング部では，抽象プロセッサの実プロセッサに対する割り付けが決定した時点であらかじめ必要なイタレーションおよび内部で実行される文をリストの形で求めておく。ループ実行部では，このリストに含まれるイタレーションのみ実行し，不要なイタレーションの実行や条件判定を防ぎ高速実行可能なコードを生成する。本手法は，データ参照が不規則になるようなループを並列化する手法として従来より提案されてきた Inspector/Executor 手法 [17] を，実プロセッサに対し不規則に割り付けられた抽象プロセッサの参照に応用した手法ともいえる。

4.2.2 基本的なコード生成方針

非均質に分散されたデータを参照するループを高速に実行するためには各実プロセッサ上で次の情報が必要となる。

1. 実プロセッサ上で実行するイタレーション

```
makeExecList(){
  ExecList = φ;
  for i=0, n-1 do
    $ExecElem.condition = \phi$
    if( RP_OWNER( a(i) )=RP_ID ) then   /*  a(i) を所有*/
      ExecElem.conditionに $I_a$を追加
    endif
    if( RP_OWNER( b(i+1) )=RP_ID ) then /*b(i+1) を所有*/
      ExecElem.conditionに $I_b$を追加
    endif
    if( $ExecElem.condition \neq \phi$ ) then
                                               /*実行する必要がある*/
      $ExecElem.iteration = i$
      ExecListに ExecElemを追加
    endif
  endfor
  return( ExecList )
}
```

(a) ループスケジューリング部

```
/*ループ実行部*/
for ExecElem in ExecList do
  i = ExecElem.iteration
  switch ExecElem.conditon
    case $I_a \cap I_b$:              /*a(i) も b(i+1) も所有*/
      a(i) = a(i)+b(i+1)
    case $I_a$:                      /*a(i) のみ所有*/
      recv( RP_OWNER( b(i+1) ), tmp)
      a(i) = a(i) + tmp
    case $I_b$:                      /*b(i+1) のみ所有*/
      send( RP_OWNER( a(i) ), b(i+1) )
  endswitch
endfor
```

(b) ループ実行部

図 9: 本手法によるループ実行の概念

2. 実行するイタレーションで参照されるデータに関する通信の有無

ループスケジューリング部はこれらの情報を，自プロセッサが所有する抽象プロセッサの情報から解析し実行イタレーションリストとして生成する。ループ実行部では生成された実行イタレーションリストにもとづいて適切なデータ参照を行う。

図 9 に本手法により図 2(a) を並列化したループを示す。図中では実行イタレーションリストを *ExecList*とし，イタレーションリスト内の要素を *ExecElem*としている。*ExecElem*は構造体データであり，実行すべきイタレーション番号 *iteration*と，そのイタレーションでのデータ通信の有無のパターンを表す識別子 *condition*からなる。

ループスケジューリング部 (a) では，ループ実行に先立って実プロセッサに分散されているデータ領域を解析し *ExecList*を生成する。この例では，逐次コード中に含まれる 2 つのデータアクセス`a(i)`および`b(i+1)`について，各イタレーションにおいて参照が行われるかどうかを求める。いずれかのデータをローカルに参照できるイタレーションの情報は，どのデータをアクセスするかの情報 I_a, I_bとともに *ExecList*に追加する。実行イタレーションリストの作成に必要な変数は，全実プロセッサが持つインデックステーブル *APtoRPindex*() のみであり，ループスケジューリング部は他の実プロセッサとの通信なしに実行できる。

ループ実行部 (b) ではループスケジューリング部によって求めた *ExecList*を参照しながら必要なイタレーションを実行する。リストに含まれないイタレーションは全く実行されないため，不要なイタレーション実行によるオーバヘッドは生じない。

また，各実行イタレーションにおいてどのデータに対する参照が行われるかはループスケジューリング部で求めているため，イタレーション内での条件判断を switch 構文 1 つにでき，図 4 と比較して実行時のオーバヘッドが小さい。

4.2.3 スケジューリングおよび実行コストの削減

図 9 に示した方針のままでも実行時のオーバヘッドをある程度削減することが可能である。しかしこのコードではループスケジューリング部で $ExecList$ を求めるために全イタレーションを実行しており効率が悪い。ループ実行部でも実行される全イタレーションで switch 構文による条件判断をしておりオーバヘッドが問題となる。そこで次に示すような手法によってスケジューリングコストおよびループ実行コストの削減を行う。

スケジューリングコストの削減は，抽象プロセッサ内の規則性を利用し，非均質性を抽象プロセッサ外に限定することで可能となっている。

ブロック分割を扱うループにおけるスケジューリングコスト削減　データが抽象プロセッサに対してブロック分割されている場合，各抽象プロセッサは規則的に連続した範囲のデータ領域を所有している。このため，実プロセッサは自分に割り付けられた抽象プロセッサ番号から，自分の所有するデータ領域を簡単な式によって求めることができる。また，実プロセッサがデータをアクセスするようなイタレーションの範囲も同様に容易に求まる。実プロセッサが抽象プロセッサ AP_p を所有している時，データ $X(ai+b)$ をローカルに参照できるイタレーション i は式 (3) より

$$\frac{(N/N_{AP}) \times AP_p - b}{a} \leq i < \frac{(N/N_{AP}) \times (AP_p + 1) - b}{a}$$

と求められる。このように，抽象プロセッサに対しブロック分割されたデータを扱うループでは全イタレーションを実行することなくイタレーションリストが作成でき，大幅にスケジューリング時間を短縮できる。

サイクリック分割を扱うループにおけるスケジューリングコスト削減　データがサイクリック分割によって抽象プロセッサに割り付けられている場合，データの分割次元は周期 N_{AP} で同じ抽象プロセッサに所有される。実プロセッサには抽象プロセッサ単位でデータが割り付けられており，データは周期 N_{AP} で同じ実プロセッサに配置されることも同時に保証される。

このため，イタレーション $i = i_0 + nN_{AP}(n:\text{整数})$ においてアクセスすべきデータ $X(ai+b)$ とそれを所有する実プロセッサの関係は

$$RP\_OWNER(X(a(i_0+nN_{AP})+b)) = RP\_OWNER(X(ai_0+b)) \tag{5}$$

であることが成り立つ[2]。本手法ではこの性質を利用し，サイクリック分割されたデータを扱うループは大きさ N_{AP} でストリップマイニングを行うことでスケジューリングコストを削減する。図 10 に例を示す。(a) のループはサイクリック分割されたデータを参照するループであるため，ループを大きさ N_{AP} でストリップマイニングし (b) のループに変形する。(b) ではデータを所有するかどうかの判定は外側ループに関係なく i'' のみを参照するだけでよい。このためスケジューリングも i'' の実行範囲である 0 から $N_{AP}-1$ までで行えばよく，リスト長およびスケジューリングコストを (N/N_{NP}) にまで削減できる。

```
real x(N)
processors AP(N_AP)
distribute x(CYCLIC) onto AP
for i = 0, N-1 do
  x(ai+b) = 5
endfor
```

(a) 逐次コード

```
for i'=0, N-1, N_AP do
  for i''=0, N_AP - 1 do
    i=i' + i''
    if( RP_OWNER( x(ai''+b) )=RP_ID )then
      x(ai+b) = 5
    endif
  endfor
endfor
```

(b) ストリップマイニング後

図 10: サイクリック分割データを扱うループの処理

スケジューリング情報の再利用　イタレーションリストは自プロセッサに対して分散されているデータ領域が判明した時点で求めることができる。さらにこの値は抽象プロセッサの実プロセッサに対する割り付けが変更されない間は変化しないため，データ再分散が生ずる直前までは同一のループに対し分散直後に求めたリストをそのまま用いることができる。(図 11)。このため反復法などでは分散決定時にスケジューリングを行うだけでよく，全体の実行時間と比較してスケジューリング時間を無視できる。

また，使用する配列のサイズおよび分割手法，ループ内部において用いられる配列の添字表現が同一であれば他のループとイタレーションリストを共有でき，スケジューリング回数を減少できる。

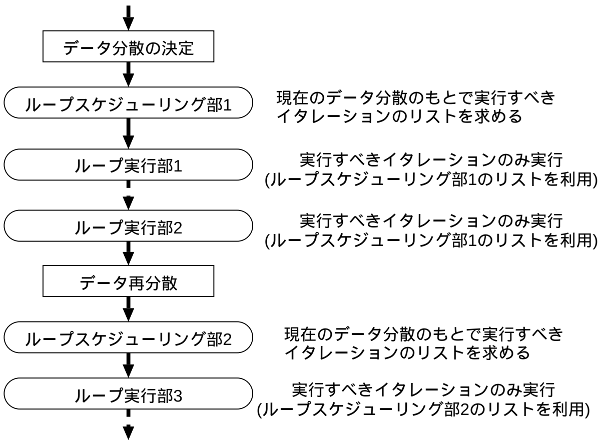


図 11: ループスケジューリング情報の再利用

ループ実行コストの削減　図 9(b) のループ実行部では各イタレーション毎にそのイタレーションでどのようなデータ参照を実行するかを判定している。抽象プロセッサに対しブロック分割

---

[2] 式 (2)(4) より

$$\begin{aligned}&RP\_OWNER_{cyclic}(X(a(i_0+nN_{AP})+b))\\&\quad = APtoRPindex((a(i_0+nN_{AP})+b) \bmod N_{AP})\\&\quad = APtoRPindex((ai_0+b) \bmod N_{AP})\\&\quad = RP\_OWNER_{cyclic}(X(ai_0+b))\end{aligned}$$

```
for $ExecElem$ in $ExecList$ do
  /*参照パターン $condition$を持つイタレーションの開始 */
  $l$ = $ExecElem.start$

  /*参照パターン $condition$を持つイタレーションの終了値 */
  $u$ = $ExecElem.end$

  switch $ExecElem.condition$
    case $I_a \cap I_b$:               /*a(i) も b(i+1) も所有*/
      for i = $l$, $u$                 /* $l$から $u$まで連続実行*/
        a($i$) = a($i$)+b($i$+1)
      endfor
    case $I_a$:                       /*a(i) のみ所有*/
      for i = $l$, $u$
        recv( RP_OWNER( b($i$+1) ), tmp)
        a($i$) = a($i$) + tmp
      end
    case $I_b$:                       /*b(i+1) のみ所有*/
      for i = $l$, $u$
        send( RP_OWNER(a($i$)), b($i$+1) )
      endfor
  endswitch
endfor
```

図 12: ループ実行コストの削減 (ループ実行部)

されたデータを扱うような場合は連続したデータを所有しており，連続した数イタレーションが同じパターンのデータ参照を行うことが多い。そこで連続する同じ参照パターンのイタレーションを 1 つのイタレーションリストとして扱うことで，1 つのイタレーションリストで複数イタレーションを連続実行する (図 12)。図の例では，イタレーションリストには参照パターン $condition$とその参照パターンでの初期値 $start$，終了値 endを持たせておき，$start$から endまでを連続に実行し `switch` 構文の使用回数を削減している。

4.2.4 生成コード長に関する考察

ループ実行部では，ループ内で参照される各配列要素についてローカル参照可能かどうかの全ての組み合わせに関して `case` 文を作成する。このような方法だけでは，ループ内で参照する配列要素数が増加した時にコード長が爆発的に増加しうるため，生成するループ内の状態数に上限を設けるなど対応が必要となる。具体的には，ローカル参照可能な配列要素の組み合わせにより，イタレーションを

- 通信の全く生じないイタレーション (全データをローカルに参照可能)
- 通信が生ずるイタレーション

の 2 つに分け，通信が生ずるイタレーションは `case` 文の内部で条件判定を行う。図 13 に図 12 のループ実行部にこの処理を適用したコードを示す。このコードでは，通信が生ずる組み合わせに対しては条件判定のオーバヘッドが生ずるが，このオーバヘッドは通信のオーバヘッドおよび受信待ち時間と比較して小さいため，イタレーション実行時間に大きく影響しないと考えられる。`case` 文の項の数にこのような制限を設けるかどうかはコンパイル時のパラメータとして指定することとしている。

```
for $ExecElem$ in $ExecList$ do
  $l$ = $ExecElem.start$
  $u$ = $ExecElem.end$
  switch $ExecElem.conditon$
    case $I_a \cap I_b$:                /*通信が不要*/
      for i = $l$, $u$
        a($i$) = a($i$)+b($i$+1)
      endfor
    default: /*($I_a \cup I_b$)        通信が必要*/
      for i = $l$, $u$
        if( RP_OWNER(b(i+1))= RP_ID ) then
          send( RP_OWNER(a($i$)), b($i$+1) )
        else if( RP_OWNER(a(i))= RP_ID ) then
          recv( RP_OWNER( b($i$+1) ), tmp)
          a($i$) = a($i$) + tmp
        endif
      endfor
  endswitch
endfor
```

図 13: 複数の条件判定の合成

4.2.5 スケジューリング処理の詳細

以下に本手法におけるループの並列コード生成手法の詳細について述べる。

最初に本手法が適用可能なループの条件について述べ，次にスケジューリング部からループ実行部に渡される実行イタレーションリストの内容，ループスケジューリング部・ループ実行部の処理手順について示す。

対象とするループ 本手法が適用可能なループは次の条件を満たす必要がある。なお，以下ではループ内でアクセスされる配列変数を総称して $X()$ とおく。

1. $X()$ はブロック分割またはサイクリック分割といった規則的な分割方法により抽象プロセッサに対して分散されている。分割次元および分割次元数に関しては制限しない。
2. $X()$ における全ての分割次元について，$X(i+j)$ などのように複数のループインデックスにより同時に一つの分割次元がアクセスされない。
3. $X()$ の分割次元に存在する，ループインデックス以外の変数はループ内で値が更新されない。これは，ループ内でその変数の値が更新された時点で実行すべきイタレーションが異なってくるためである。ただし，$X()$ をアクセスする全ループの外側において定義され，以降ループが終了するまで値が変化しないような変数を添字に用いている場合には，その変数が決定した時点で変数の値を定数として扱い，実行イタレーションリストを再生成することで本手法が適用できる。

$1 \sim 3$ の条件を満たすループに対し，初期値・終了値は変数を含め任意の整数をとるがステップ値が 1 となるように正規化を行うと，本手法が適用可能なループは図 14(a) のように表現できる。左辺は項 $x_0 = X(A_0\vec{i} + b_0)$ から，右辺は m 個の項 $x_k = X(A_k\vec{i} + b_k) \quad (k = 1, ..., m)$ からなる。ただし，A_kは サイズがループ次元数となる正方行列とし，各行に最大 1 個の 0 以外の整数が存在し，$A_k \neq 0$ とする。$\vec{b_k}$は整定数からなるベクタとする。$\circ$は任意の演算を示す。

以下の議論では，特にデータを 1 次元方向にのみ分割して 1 次元の抽象プロセッサに対して分散させた 1 重ループ (図 14(b)) について手順を述べるが，多次元分割の時には各次元にスケジューリングを行い，多重ループはループ変数をベクトルとして扱うことで拡張可能である。

実行イタレーションリスト ある実プロセッサ RPに関する実行イタレーションリスト $ExecList$は三つ組 $T = \langle l : u : C\rangle$ のリストで表現される。lと u は連続したイタレーションの開始

```
processors AP($\vec{N_{AP}}$)
$data\_type$ $X(\vec{N})$
distribute $X_1(\vec{D_1})$, ..., $X_m(\vec{D_m})$ onto AP

for $\vec{i}$ = $\vec{l}$, $\vec{u}$ do
  $X_0(A_0\vec{i}+\vec{b_0})$
        = $X_1(A_1\vec{i}+\vec{b_1})\circ X_2(A_2\vec{i}+\vec{b_2})\circ\cdots\circ X_m(A_m\vec{i}+\vec{b_m})$
endfor
```

(a) 一般的なループ表現

```
processors AP($N_{AP}$)
$data\_type$ X($N$)
distribute X($D$) onto AP

for $i$ = $l$, $u$ do
  X( $a_0 i + b_0$ ) =  X($a_1 i + b_1$)oX($a_2 i + b_2$)o$\cdots$oX($a_m i + b_m$)
endfor
```

(b) 議論に用いるループ

図 14: 本スケジューリング手法の適用可能なループ

値と 終了値$+1$ である。また Cは $l \leq i < u$ なるイタレーション i において，RPが所有するデータを参照するような項の集合である。また，$ExecList$はイタレーションの実行順にソートされており，同じイタレーションが複数の三つ組において重複しない。

例: $ExecList$=$\langle 3:5:\{x_0\}\rangle\langle 5:9:\{x_0,x_1\}\rangle$
イタレーション $i=3,4$ において項 x_0のみが RPのデータを参照し，イタレーション $i=5,6,7,8$ において項 x_0, x_1ともに RPのデータを参照する。それ以外のイタレーション i は全ての項に対する演算について RP が所有するデータは参照されない (イタレーションは実行しなくてよい)。

ループスケジューリング部　ループスケジューリング部では，ループ実行に必要となる情報を保持しておくイタレーションリストを生成する。

最初に各項 x_kについて，その項が RPのデータを参照するようなイタレーションの集合 L_kを求める。次に L_kを合成することにより実行イタレーションリスト $ExecList$を求める。具体的な手順は以下に示す。

1. 各項 x_kについて，その項が RPのデータを参照するようなイタレーションを求め，x_kに関するイタレーションリスト L_kを作成する。
   (a) RPが所有する抽象プロセッサ $AP_{l0},...,AP_{lp},...,AP_{lP'}$ $(0 \leq P' \leq N_{AP})$ についてイタレーションリスト L_{kp}を求める。データは各抽象プロセッサに対しては規則的な分割によって分散されているため，L_{kp}は均質環境と同様容易に求まる。
   項 $x_k = X(a_k i + b_k)$ を抽象プロセッサ AP_{lp}がアクセスするイタレーションの範囲はブロック分割・サイクリック分割でそれぞれ

$$\frac{(N/N_{AP})\times AP_{lp}-b_k}{a_k} \leq i_{block} < \frac{(N/N_{AP})\times(AP_{lp}+1)-b_k}{a_k} \qquad (6)$$

$$i_{cyclic} = \frac{AP_{lp}-b_k}{a_k} + nN_{AP} \quad (n=0,1,...,(N/N_{AP})-1) \qquad (7)$$

   となり実行イタレーションリスト L_{kp}は

$$L_{kp} = \begin{cases} \langle\frac{(N/N_{AP})\times AP_{lp}-b_k}{a_k} : \\ \quad \frac{(N/N_{AP})\times(AP_{lp}+1)-b_k}{a_k} : \{x_k\}\rangle \\ \quad \cdots(\text{ブロック分割}) \\ \langle(\frac{AP_{lp}-b_k}{a_k}) \bmod N_{AP} : \\ \quad (\frac{AP_{lp}-b_k}{a_k}) \bmod N_{AP}+1 : \{x_k\}\rangle \\ \quad \cdots(\text{サイクリック分割}) \end{cases} \qquad (8)$$

   として求まる。
   (b) 全抽象プロセッサの和集合を取ることにより，項 x_k について RP が所有するデータを参照するようなイタレーションリスト L_kを求める。ただし，2 つの連続した三つ組 $T_1 = \langle l_{kp}:u_{kp}:\{x_k\}\rangle, T_2 = \langle l_{k(p+1)}:u_{k(p+1)}:\{x_k\}\rangle$ について $u_{kp} = l_{k(p+1)}$であれば三つ組 $T' = \langle l_{kp}:u_{k(p+1)}:\{x_k\}\rangle$ に変換される。

$$\begin{aligned} L_k &= \bigcup_p L_{kp} \\ &= \langle l_{k1}:u_{k1}:\{x_k\}\rangle\langle l_{k2}:u_{k2}:\{x_k\}\rangle\cdots \\ &\qquad \langle l_{kP'}:u_{kP'}:\{x_k\}\rangle \end{aligned}$$

2. 全ての L_kを合成することにより実行イタレーションリスト $ExecList$ を生成する。

$$ExecList = \bigoplus_k L_k$$

   合成$\bigoplus$は，複数のイタレーションリストから各イタレーションでどの項の組み合わせがローカル参照するかを求め，一つのイタレーションリストを生成する。例えば，2 つのイタレーションリスト $L_1 = \langle 5:10:\{x_0\}\rangle, L_2 = \langle 7:20:\{x_1\}\rangle$ の合成 $L_1 \bigoplus L_2$は $L = \langle 5:7:\{x_0\}\rangle\langle 7:10:\{x_0,x_1\}\rangle\langle 10:20:\{x_1\}\rangle$ となる。実行イタレーションリストの合成$\bigoplus$のアルゴリズムは図 15 に示す。

ループ実行部　ループ実行部では，ループスケジューリング部で求めた実行イタレーションリスト $ExecList$ に従ったイタレーションの実行を行う。

ループ実行部は図 16 に示すようなコードが生成される。ループの最外部に実行イタレーションリストが 1 要素ずつ取り出されるようなループが生成される。その内側には，イタレーションがローカルに参照できる項の組み合わせ Cによってそれぞれ適切なアクセスを実行するよう条件文 (`switch`〜`case` 構文) が挿入される。さらに同じ参照条件を持つ複数のイタレーションを連続的に実行するための `for` 文が `case` 文の内側に置かれる。

`case` 文内の各項目内のコードは次のようにして作成される。

> 項 $\{x_0,x_1,...,x_m\}$ のすべての組み合わせ ($2^{(m+1)}$ 通り) について，ローカル参照の可/不可により適切にデータを扱うための文を生成する。具体的には，項の組み合わせ $C_k (k=0,1,...,2^{(m+1)}-1)$ について以下の要領で必要な文の挿入を行う。
>
> - $x_0 \in C_k$の時 (右辺が含まれる):

```
algorithm $\oplus$( $L_1$ , $L_2$ ){
入力:
  $L_1$=$\langle l_{11} : u_{11} : C_{11}\rangle\langle l_{12} : u_{12} : C_{12}\rangle \cdots \langle l_{1n} : u_{1n} : C_{1n}\rangle$
        $(l_{1k} < u_{1k}\ (1 \le k \le n),\ l_{1k} \le u_{1(k+1)}\ (1 \le k \le n-1))$
  $L_2$=$\langle l_{21} : u_{21} : C_{21}\rangle\langle l_{22} : u_{22} : C_{22}\rangle \cdots \langle l_{2n} : u_{2n} : C_{2n}\rangle$
        $(l_{2k} < u_{2k}\ (1 \le k \le n),\ l_{2k} \le u_{2(k+1)}\ (1 \le k \le n-1))$
アルゴリズム:
  $L \leftarrow \phi$   /* 返り値の初期化 */
  pop $T_1 = \langle l_1 : u_1 : C_1\rangle$ from $L_1$   /*$L_1$の先頭から 1 要素抽出*/
  pop $T_2 = \langle l_2 : u_2 : C_2\rangle$ from $L_2$   /*$L_2$の先頭から 1 要素抽出*/

  while( $(T_1 \ne \phi) \wedge (T_2 \ne \phi)$ ) do
    if( $u_2 < l_1$) then          /* $T_2$が必ず先に実行される */
      append $T_2$ to $L$
      pop $T_2$ from $L_2$
      continue
    endif
    if ( $u_1 < l_2$) then          /* $T_1$が必ず先に実行される */
      append $T_1$ to $L$
      pop $T_1$ from $L_1$
      continue
    endif

    /* $T_1$と$T_1$の範囲が重複している */
    /* $T_1$と$T_2$の重複しない最初の部分を求める */
    if( $l_1 < l_2$ ) then          /* 最初の部分は$T_1$*/
      append $\langle l_1 : l_2 : C_1\rangle$ to $L$
    else if( $l_2 < l_1$ ) then     /* 最初の部分は$T_2$*/
      append $\langle l_2 : l_1 : C_2\rangle$ to $L$
    endif

    /* $T_1$ , $T_2$共通に含まれるイタレーションを求める */
    if( $\max(l_1, l_2) < \min(u_1, u_2)$ ) then /*共通部分が存在*/
      append $\langle \max(l_1, l_2) : \min(u_1, u_2) : C_1 \cup C_2\rangle$ to $L$
    endif

    /* 残り部分は$L_1, L_2$の次の要素と比較する
    if( $u_1 < u_2$ )then              /* $T_2$が残り部分になる */
      $T_2 \leftarrow \langle u_1 : u_2 : C_2\rangle$
      pop $T_1$ from $L_1$
    else if( $u_2 < u_1$ )then         /* $T_1$が残り部分になる */
      $T_1 \leftarrow \langle u_2 : u_1 : C_1\rangle$
      pop $T_2$ from $L_2$
    else                         /* 残り部分は存在しない→次へ */
      pop $T_1$ from $L_1$
      pop $T_2$ from $L_2$
    endif
  endwhile
  /* 残り部分 */
  append $T_k$ to $L$ if $T_k \ne \phi\ (k = 0, 1)$
  append $T_k \in L_k$ to $L$ if $L_k \ne \phi\ (k = 0, 1)$
  return $L$
}
```

図 15: 実行イタレーションリストの合成

```
for $T = \langle l : u : C\rangle$ in $ExecList$ do
  switch( $C$ )
   case $\{x_0\}$: /* 項$x_0$のみローカル参照可能 */
    for $i$=$l$, $u$ do
      receive $X_1(a_1 i + b_1), X_2(a_2 i + b_2), ..., X_m(a_m i + b_m)$
      $X_0(a_0 i + b_0) =$  ....
    endfor
   case $\{x_1\}$: /* 項$x_1$のみローカル参照可能 */
    for $i$=$l$, $u$ do
      send $X_1(a_1 i + b_1)$
    endfor
   case $\{x_2\}$:
     :
   case $\{x_1, x_2\}$:
     :
   case $\{x_0, x_1, x_2, ..., x_{m-1}\}$:
     :
   case $\{x_0, x_1, x_2, ..., x_{m-1}, x_m\}$:
            /* 全項にローカル参照可能 */
    for $i$=$l$, $u$ do
      $X_0(a_0 i + b_0) = X_1(a_1 i + b_1) \circ X_m(a_m i + b_m)...$
                        $\circ X_m(a_m i + b_m)$
    endfor
  endswitch
endfor
```

図 16: ループ実行部

  - $x_l \notin C_k$となる $x_l (l = 1, ..., m)$ について，x_lのデータを受信するための文`recv(`$RP\_OWNER(x_l)$，tmp_l`)` を挿入する。
  - x_0に対する代入演算を行う文を挿入する。ただし`recv` 文を挿入した項は受信データtmp_lを使用するよう変更を加える。
- $x_0 \notin C_k$の時:
  - $x_l \in C_k$となる x_lについて，x_0を所有する実プロセッサに x_lを送信するための文 `send(`$RP\_OWNER(x_0)$，x_l`)` を挿入する。
- $C_k = \phi$の時: コード生成を行わない。

また，データがサイクリック分割されている時には実行イタレーションリストをたどるループの外側にはストリップマイニングにより生成された外側ループも挿入される。

4.3 分割データのメモリ配置

各実プロセッサには複数個の抽象プロセッサが任意の形状で分散される。このため，実プロセッサごとに必要な記憶領域が異なり，また各データのローカルなアドレスも簡単に求めることができない。また，動的再分散によって動的に抽象プロセッサの割り付けが変化するため，実行開始時に記憶領域を固定できない。本手法では，配列データを抽象プロセッサ毎に確保し，ポインタ配列により一元的に管理することでデータ領域の非均質性や動的変化に対応する。

図 17 に本手法における 1 次元に分割されたデータのメモリ配置の例について示す。分割された配列データ $X(k)$ の各実プロセッサにおけるインデックスは

- データ $X(k)$ が分散された抽象プロセッサ番号 $RP\_OWNER(X(k))$
- 抽象プロセッサ内でのデータ $X(k)$ のインデックス $LOCAL\_INDEX(k)$

という 2 つのパラメータにより特定され，ポインタ配列 $(*X[N_{AP}])[N/N_{AP}]$ を用いて参照する。すなわち，データ $X(k)$ に対するアクセスは

$$(*X[AP\_OWNER(X(k))])[LOCAL\_INDEX(k)]$$

として行う。

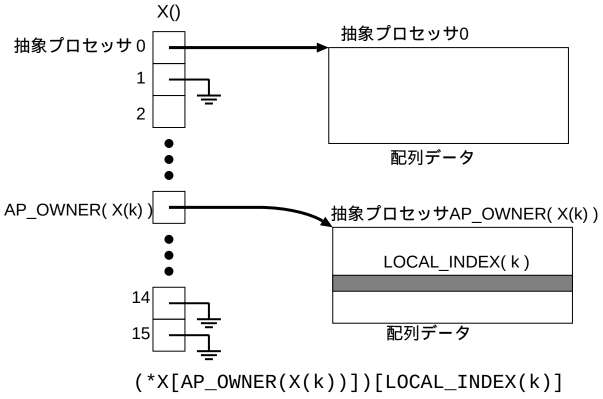


図 17: 非均質環境におけるメモリ配置

各実プロセッサに対して割り付けられた抽象プロセッサのデータ領域は抽象プロセッサ単位で記憶領域に確保されているため，割り付けが決定した時点で必要となる抽象プロセッサのデータ分の記憶領域確保を行えばよい。また，一抽象プロセッサあたりのデータ量は一定であるため，動的再分散の際に出入りのある抽象プロセッサ間で記憶領域の再利用もでき効率的である。

5 動的負荷分散機能

本章では，*hetero-TINPAR* による動的負荷分散機能について述べる。

5.1 概要

第 2.4 節で述べたとおり，非均質環境では，プロセッサに割り付けるタスク量の決定が困難であり，さらにマルチユーザ環境では各プロセッサの利用可能な計算能力が動的に変化するといった問題がある。これらを解決するためには以下の機能が不可欠となる。

自動的なデータ分散比の決定　ユーザが実行開始時にパラメータとしてデータ分散を指定しなくても，実行中にプロセッサの性能を実測し適切なデータ分散比を求め，さらにデータをその分散比で再分散させ実行の最適化を図る。

動的負荷分散　実プロセッサの負荷をモニタリングしながら実プロセッサに対するタスク量を変化させ，負荷バランスを保ったまま実行を行う。

これらの機能はどちらもプロセッサの性能予測およびタスクの移動といった同様の処理によって行われるため，以下ではこれらを総称して動的負荷分散とする。

hetero-TINPAR では，抽象プロセッサの実プロセッサへの割り付けをプログラム実行中に変更することで各実プロセッサに割り付けるタスク量を変化させる。プロセッサ割り付けの変更は，第 4.1, 4.2 節で述べたように，

- インデックステーブル $APtoRPindex()$ の変更
- ループのスケジューリング
- 実プロセッサ間のデータ移動

を実行中に行うことで可能である。

従来からコンパイラレベルの動的負荷分散方式として Chunk スケジューリング [18] や GSS[19, 18] など種々の方式が提案されている。ところがこれらは主に doall 型の並列実行を前提としているため，より一般的な，依存を含むループへの適用が困難である。

そこで *hetero-TINPAR* では，計算に参加している全実プロセッサが，ある定められた時点で協調して負荷分散を行う調歩協調型の負荷分散戦略を採用した。チェックポイントの設定法，負荷情報の種類と収集方法，協調の方法に関して以下のように定める。

チェックポイント　抽象プロセッサに割り当てられたタスクの最外ループの数イタレーションに対応する処理が実プロセッサ上で全て終了した時点とする。すなわち，プログラム中において全実プロセッサが並列処理を終了し，同期を取ることができるような点をチェックポイントとする (図 18)。 具体的には SOR などの反復法における最外ループ (時間軸) におけるイタレーションが相当する。

hetero-TINPAR では実行時オーバヘッドを低減するため，シングルスレッド方式によって複数抽象プロセッサの処理を 1 つのタスクとして処理する。このため，実プロセッサ上では複数の抽象プロセッサの処理が融合された形で行われており，抽象プロセッサの独立性は非常に低いものとなっている。このような状態で特定の抽象プロセッサのみを他のプロセッサに移動させるためには，各イタレーションごとに実行したかどうかのフラグを立てるなどの処理が必要となるため，実行の高速化などの観点から並列実行が行われている部分 (図 18 における S の部分) では，抽象プロセッサの実プロセッサ間の移動による負荷分散は行わない。

これは負荷分散実行上の制限となりうるが，負荷分散が実際に必要となるのは数十分から数時間といった長時間の実行が必要となるアプリケーションと考えられ，このようなプログラムは反復法などのチェックポイントが設けやすい形となっていることが多いため，実際上は大きな問題とはならない。

チェックポイントの指定はループに対するディレクティブの形で与え，特定のイタレーション毎に負荷分散を行うことができる。ディレクティブはループ構文に対し

$$\texttt{for}\ index\ \texttt{=}\ lower,\ upper,\ step\ \underline{\texttt{remap}\ r}\ \texttt{do}$$

のように記述し，この例では $index$ ループにおいて r イタレーション毎に負荷分散を行う。

負荷情報　一つ前のチェックポイントで割り付けられた抽象プロセッサ数とその実行に要した時間とする。

協調の方法　得られた負荷情報をもとに，次のチェックポイントまでの各実プロセッサの処理能力を予測しその能力に比例して抽象プロセッサを割り付けるという方式をとる [20]。

5.2 アルゴリズム

動的負荷分散機能は負荷分散フェーズとデータ再分散フェーズからなり，図 18 のようにチェックポイントで全プロセッサ間で同期をとった後に実行する。図 19 に動的再分散のアルゴリズムを示す。

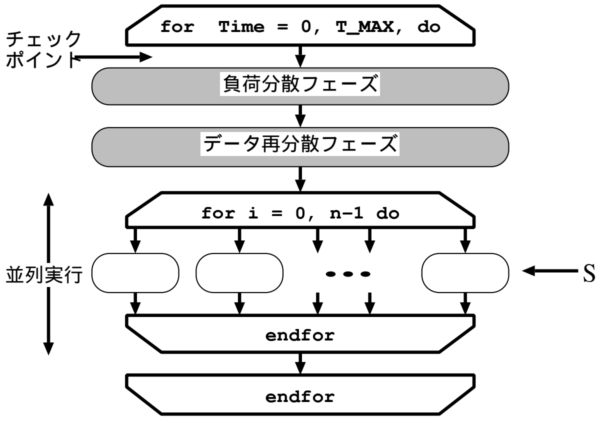


図 18: 負荷分散の流れ

負荷分散フェーズ　負荷分散フェーズでは，各プロセッサから実行時間・通信時間などの情報を得ることで各実プロセッサのおおよその処理能力を判断する。これを用いて次の演算における計算時間を予測し最適な分散比率を決定する。

抽象プロセッサはほぼ同一の計算量を持つと仮定すると，負荷の平均化は全抽象プロセッサが同一実行時間となるような配分で実プロセッサに再分散することによって可能である。時刻 $t-1$ から t までの，各実プロセッサ k における通信を除いた実行時間 $Rcomp_k(t)$ から，1 抽象プロセッサ当りの平均実行時間 $Acomp_k(t)$ を求め，新たな $Restim_k(t) = Acomp_k(t) \times AP_k(t)$ が等しくなるように再分散する。新しく求めた $Restim_k(t)$ は次の $t \sim t+1$ の実行時間予測であり，再分散によってこれを均等化しているといえる。各実プロセッサに対する最適な抽象プロセッサ数の予測 $AP_k(t)$ が決定するとこれをもとに第 5.3 節で述べる実プロセッサに対する割り付け手法に従って最終的な $APtoRPindex()$ を求める。

なお，負荷分散フェーズの計算を実際に行うプロセッサは，全プロセッサの中で最も演算時間の短かったものとし，負荷分散時間の短縮を図っている [21]。

データ再分散フェーズ　データ再分散フェーズでは，負荷分散フェーズによって決定されたデータ分散比率をもとに，実プロセッサ間でデータの移動を行う。本手法では，データの分散の単位として抽象プロセッサを採用しているため，再分散も抽象プロセッサを単位とした移動を行う。データの移動に伴い抽象プロセッサの所属する実プロセッサ番号が変更となるため，インデックス $APtoRPindex()$ を全プロセッサ上で変更する。

また，割り付けが変更されなかったプロセッサは再分散前のループスケジューリングの結果をそのまま利用することができるが，抽象プロセッサの割り付けの変更された実プロセッサは，ループの実行すべきイタレーションが変化するためループの再スケジューリングを行う。

5.3　抽象プロセッサ割り付けの決定手法

負荷分散フェーズにより性能予測に基づいた理想的な抽象プロセッサの実プロセッサへの割り付け数が求まる。本節ではこの理想的な割り付け数にもとづいた割り付け抽象プロセッサ数の最終的な決定や，どの抽象プロセッサをどの実プロセッサに割り付けるかなどの決定手法について述べる。

変数:

- 時刻 t—抽象プロセッサに割り当てられたタスクの最外ループの整数倍にとった，再分散のタイミング
- N_{AP}, N_{RP}—実プロセッサ数・抽象プロセッサ数
- RP_k—実プロセッサ $(k = 1, 2, ..., N_{RP})$
- $AP_k(t)$—時刻 t において実プロセッサ RP_k が所有する抽象プロセッサ数
- $Rcomp_k(t)$—RP_k における時刻 $t-1$ から t までの，同期待ち・通信時間をのぞいた実行時間
- $Acomp_k(t)$—RP_k における時刻 $t-1$ から t までの 1 抽象プロセッサ当りの，通信をのぞいた実行時間
- $Restim_k(t)$—時刻 t から $t+1$ までの，RP_k における通信をのぞいた実行時間の予測

手順:

1. [負荷分散フェーズ] 時刻 t において実プロセッサ $RP_k(k = 1, 2, ...N_{RP})$ 間の同期により実行を中断する。
2. RP_k における計算時間より 1 抽象プロセッサ当りの平均実行時間を求める。
$$Acomp_k(t) = \frac{Rcomp_k(t)}{AP_k(t-1)}$$
3. 次の条件を満たすような $AP_k(t)$ を求める。
$$Restim_1(t) = Restim_2(t) = ... = Restim_{N_{RP}}(t)$$
ただし $Restim_k(t) = Acomp_k(t) \times AP_k(t)$
$$\sum_{k=1,..N_{RP}} AP_k(t) = N_{AP}$$
4. (3) で求めた $AP_k(t)$ 個分の連続した抽象プロセッサを割り付けるような新しいインデックス $NewAPtoRPindex()$ を求める。割り切れなければ近似的な値を使用する。
5. [データ再分散フェーズ]$NewAPtoRPindex()$ に従いデータ再分散を行う。
    - 各実プロセッサ上で，データ分散情報を保持する変数である $NewAPtoRPindex()$ を変更する。
    - 所有者に変更のあった抽象プロセッサ上のデータを移動させる。
    - ループのスケジューリング (イタレーションリストの作成) を行う。
6. 時刻 t 以降の処理を開始する。

図 19: 動的データ再分散の手順

5.3.1　実プロセッサに対する抽象プロセッサの分散比の決定

負荷分散フェーズにより求めた理想的な分散比に従ってデータ移動を行うのみではデータ移動量や分散粒度の点で問題が生ずることがある。*hetero-TINPAR* では，理想的な分散比に従った実際の割り付け個数の決定は以下の点を考慮して行う。

再分散の粒度　ある実プロセッサの負荷が大きく増加した場合にはそのプロセッサから大量のデータを他のプロセッサに移動する必要あるが，大量のデータを一度に移動することは大きなオーバヘッドとなる。また負荷分散フェーズによる最適性能比はあくまで予測であり，大量のデータを移動させることで逆に移動先プロセッサでの次のチェックポイントまでの処理時間が予測以上に長くなることもありうる。そこで，本手法では一度に移動させるデータ量に制限を加え，段階的に最適状態に移行する方針を取る。この方式では収束に多少時間がかかるが安定した負荷分散が可能となる。*hetero-TINPAR* ではデータ移動量を，現在の分散比から予測によって求めた分散比になるまでに必要なデータ移動量の半分に制限している。ただし，プログ

ラム開始後 1 回目の再分散に関しては，均質分散で開始した実行を迅速に非均質分散に移行させることを目的しているため移動抽象プロセッサ数に関して制限は設けない。

移動通信量と負荷の不均衡のトレードオフ　ある 2 つのプロセッサの負荷の差がわずかであった場合や，プロセッサ間の通信レイテンシが大きい場合には，負荷均衡のために再分散を行うよりも，再分散しないで負荷不均衡のまま実行した方が実行時間が短くなることがありうる。このためデータ再分散に必要な時間も考慮した予測が必要となる。

現在のところは，データ再分散のコストを考慮に入れておらず，1 抽象プロセッサの移動しか必要とならない再分散はデータ移動しないでそのまま実行することとしている。

5.3.2　実プロセッサに割り当てる抽象プロセッサの位置

各実プロセッサに対して割り付ける個数が決定したのち，実際の抽象プロセッサをどのような形で実プロセッサに割り付けるかを決定する。

ここで，再分散時にどのように抽象プロセッサを割り当てるかで，再分散の時のコストを最小に抑える手法と連続した抽象プロセッサを割り当てる手法が考えられる。以下に図 20(a) のプログラムを例にしてこれら 2 手法について述べる。図 20 ではデータがブロック分割により抽象プロセッサに対して分散されており，このようなプログラムを実行した時に，各実プロセッサの性能比が負荷変動により 4:4:4:4(状態 1)→5:4:3:4(状態 2)→6:4:3:3(状態 3) と変化するものとする。

手法 1: 再分散のコストを抑える手法　動的再分散の実行により必要となる，実プロセッサ間のデータ移動をできるだけ削減する手法である。データ移動を予測性能比が変化したプロセッサの間でのみ行うことでデータ移動量は負荷による変動分に制限できる。また，分散比が変化しなかったプロセッサはデータ移動およびループスケジューリングが不要となり，負荷分散時間を短縮できる。図 20(b) の例では，状態 1 と状態 2 の間で RP0 と RP2 のみに負荷変動が生じている。データ移動を RP2 と RP0 の間で行うことで，データ移動が必要な抽象プロセッサ数を 1，ループ再スケジューリングを実行するプロセッサが RP0 と RP2 に抑えられている。状態 2 から状態 3 へも同様に RP0 と RP3 でのみデータの移動を行う。

手法 2: 連続した抽象プロセッサを割り当てる手法　動的再分散の前後で実プロセッサどうしの配置を変更しないよう，連続した領域を割り当てる手法である。各実プロセッサが，抽象プロセッサを予測性能比に従った個数ずつ連続して持つことにより，演算中に生ずる隣接した抽象プロセッサ間のデータアクセスを実プロセッサ内の代入にできる。反面，一つのプロセッサの割り当ての変化が他のプロセッサの割り当てに影響するため，データ移動コストは増加する。図 20(b) の例では，状態 1 と状態 2 の間で RP0 と RP2 のみに負荷変動が生じているが，RP0~3 の実プロセッサ間で再分散およびループスケジューリングを行わなければならない。状態 2→状態 3 の時も同様に 2 プロセッサ間の負荷の変動が全プロセッサに対して再分散を引き起こしている。

hetero-TINPAR ではこれらのうち，手法 2 を採用した。これは以下に述べる理由からである。

- 隣接要素のデータ参照が多い場合には，手法 2 を用いることで実プロセッサ内に生ずる抽象プロセッサ間の通信を，第 4.2 節で述べたループスケジューリングによって削除することができる。
- 負荷分散は数回から数十回の反復演算につき 1 回だけ行うことを考えると，データ再分散自体はプログラム全体に対して大きなオーバヘッドとならない。
- 手法 2 の領域の分割アルゴリズムは，全抽象的プロセッサを性能比で分割するだけであるため単純である。

5.3.3　多次元抽象プロセッサの割り付け

負荷分散フェーズにより求めた計算性能比で抽象プロセッサを実プロセッサに割り付ける際，抽象プロセッサ配列が 1 次元であれば抽象プロセッサの計算機の性能に比例して分割することは容易である。

ところが，多次元の抽象プロセッサ配列の予測性能に応じた割り付けを任意の形状において正確に求めることは容易ではない。そこで抽象プロセッサを 1 次元でのみ帯状に割り当てる手法や近似的な矩形領域として求める手法 [22] を用いてデータ割付けを求める。

1 次元での割り付け　図 21(a) のように特定の次元でのみ抽象プロセッサを分割する。この手法では 1 次元の時と同様に性能に比例して区切るだけでよく，容易に割り付けを決定できる。図の例では 64×64 の抽象プロセッサを x 方向に性能比 5:4:3:2:2 で分割させている。この手法では抽象プロセッサが多次元に分散されていることを結果的に無視する形になるため，アプリケーションによっては通信量の増加を招く可能性がある。例えば SOR 法では同じ数の抽象プロセッサを 2 次元で分割する時と比較して 1 次元で分割するほうが通信量が多くなる。

矩形領域による近似的な割り付け　多次元の抽象プロセッサを矩形領域に分割する場合，性能に完全に比例した形の割り付けを再分散決定時に瞬時に行うことは困難であると考えられる。そこで，次のようにして近似的に性能に比例した量の抽象プロセッサを矩形領域に分割する。

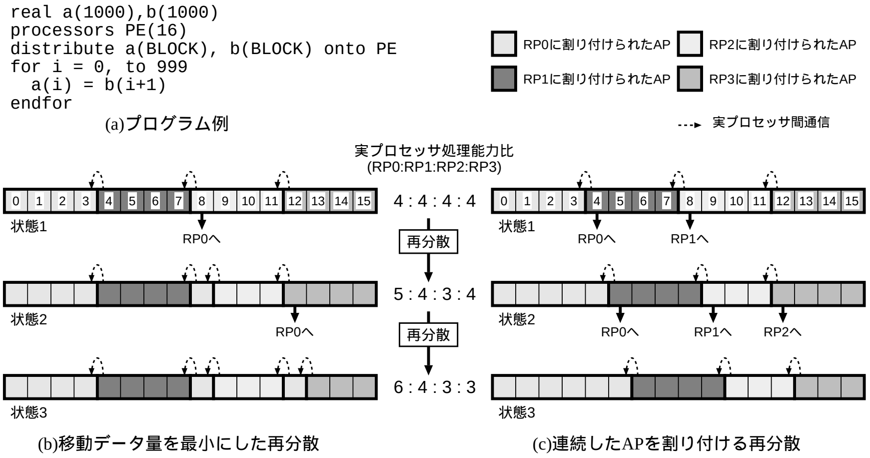
1. 実プロセッサ全体をいくつかのプロセッサグループに分割する
2. プロセッサグループにおける全体性能を求める
3. プロセッサグループの性能比で特定の次元を分割する
4. 分割された抽象プロセッサを，プロセッサグループ内の実プロセッサで別次元方向に分割する

図 21(b) に例を示す。最初に実プロセッサ 0 ∼ 4 をプロセッサグループ 1(RP0, RP1)，プロセッサグループ 2(RP2, RP3, RP4) の 2 つに分け x 方向にプロセッサグループの性能比で分割する。次に，分割された抽象プロセッサをプロセッサグループ内で y 方向に分割することで矩形領域として割り付けを行っている。この方法では，割り付けに必要なコストが 1 次元割付けの分割次元数倍で済むため，比較的容易に使用することができる。

6　性能評価

本章では，本論文に述べた非均質環境向け並列化手法により，実際に非均質環境において実行効率のよいコードを生成できるかどうかの評価を行った。

以下では，本評価に用いたプログラム・計算機環境について述べた後，第 4 章で述べたコード生成手法の評価，および第 5 章で述べた動的負荷分散の評価を行う。



```
real a(1000),b(1000)
processors PE(16)
distribute a(BLOCK), b(BLOCK) onto PE
for i = 0, to 999
  a(i) = b(i+1)
endfor
```

図 20: 再分散による割り付け手法

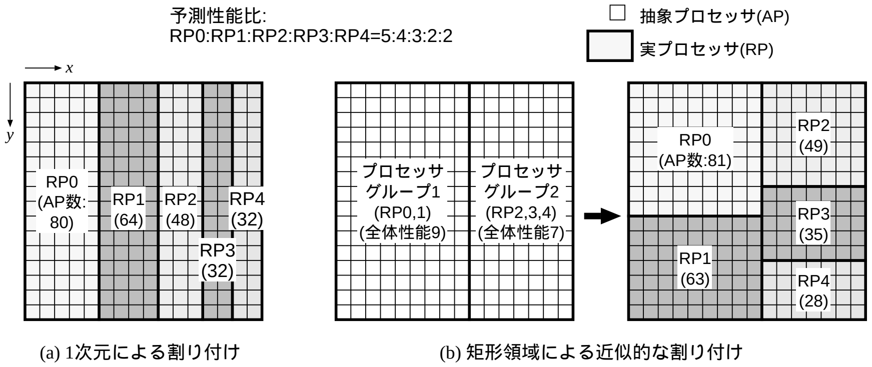


図 21: 多次元抽象プロセッサの割り付け手法

6.1 評価環境

本節では，性能評価に使用したプログラムおよび使用した非均質計算機環境について述べる。

6.1.1 評価プログラム

評価に使用したプログラムは行列積とSORの2種類である。

行列積 2次元行列 a(), b() の行列積 c() を求めるプログラムである。図22にリストを示す。行列サイズは 512×512 であり，a(), b() はともに1次元方向にブロック分割を行う。ただし配列 local_b() はリモートデータ b() の一時的なローカルコピーを格納するための配列であり，全プロセッサがそこコピーを持っている。このローカルコピーの作成に関しては *hetero-TINPAR* の前処理として行われるべきものであるが未実装であるためハンドコンパイルにより生成している。

SOR SOR法による楕円型偏微分方程式の求解プログラムである。ただし，*hetero-TINPAR* の性能・挙動の評価に用いるため収束判定は行わず，反復回数を評価時に指定するものとした。図23にプログラムを示す。使用した行列サイズは 1024×1024 である。outer_j のループは，タイリングの技法を用いるために， j のループをストリップマイニングして得られた外側ループである。なお，図23のプログラムにおいて最外ループ t は500回反復実行されるが，50反復ごとに動的負荷分散を行うようディレクティブが追加されている。

データは行列積・SORともに列方向にブロック分割を行ったものを抽象プロセッサに対して分散させた。各プログラムは抽象プロセッサ数や，抽象プロセッサの実プロセッサに対する割り付け比は評価により変化させた。SORでは反復回数や，反復中の動的再分散のタイミングも評価によさまざまなものを用いた。また，第6.2節の評価に関しては従来のTINPARによって並列化した均質環境向けコードも使用した。

なお，第4.3節で述べた非均質分散のメモリ配置については現在のところ未実装であるため，均質環境・非均質環境ともにメモリのアドレッシングの変更は行わないものを使用した。

6.1.2 評価に使用した計算環境

評価は，10Mbps のイーサネット・スイッチにより接続された SPARC Station 20(以下 SS20)4台と SPARC Ultra-1(以下 Ultra)2台により構成された環境 (図24) において最大4台構成で行い，第6.4.2節の評価以外では，他のユーザプロセスのない無負荷状態で測定を行った。また，通信ライブラリとして

```
const n = 512
processors  AP(NAP)
real a(n,n), b(n,n), c(n,n)
real local_b(n,n)
distribute a(BLOCK,*) onto AP
        /* a(), b(), c() は1次元めでブロック分割 */
distribute b(BLOCK,*) onto AP
distribute c(BLOCK,*) onto AP

/* リモートデータのローカルコピーを作成する */
for i = 0, n - 1 do
  for j = 0, n - 1 do
    local_b(i, j) = b(i, j)
  endfor
endfor
/* ローカルに計算する */
for i = 0, n - 1 do
  for j = 0, n - 1 do
    c(i, j) = 0.0
    for k = 0, n - 1 do
      c(i, j) = c(i, j) + a(i, k) * local_b(k, j)
    endfor
  endfor
endfor
```

図 22: 行列積

MPI[16] の実装処理系である MPICH[23] を使用した。表 1 に評価計算環境の詳細を示す。

表 2 に，SS20 と Ultra について行列積と SOR における逐次実行時間を示す。表より Ultra の方が SS20 の 2.2〜2.3 倍の実行性能があることがわかる。

表 2: 各プロセッサの単体性能

評価 プログラム	逐次実行時間 (s)		性能比
	SS20	Ultra	SS20:Ultra
行列積	126.4	57.0	5:11
SOR	300.1	131.3	10:23

6.2 実行時オーバヘッドの評価

本手法によって生成されたコードと従来の均質環境のためのコードとの実行時間を比較し，本手法による非均質実行のオーバヘッドについて評価した。

評価環境はSS20 を 4 台用いた均質環境とした。評価プログラムは行列積および 500 反復の SOR とし，従来の TINPAR により生成された均質環境向け並列コード，*hetero-TINPAR* により生成された非均質環境向け並列コードを比較した。

非均質環境向けコードのデータは，均質環境向けコードと同じ評価環境とするために均等に分散させ，実行中に負荷変動によるデータ移動が生じないよう動的負荷分散機能を実装しないものを用いた。また，抽象プロセッサ数は評価によって変化させた。

表 3 に均質環境と比較した実行時間，台数効果および実行時間内の処理の内訳を示す。表中の実行時間，演算時間，ループ実行時間，スケジューリング時間はそれぞれ，プログラム実行に要した時間，実行時間から通信時間を除いた時間，ループの初期値・終了値判定などループ実行に関わるオーバヘッド，ループスケジューリング部の実行時間である。なお，演算時間はループ実行時間およびスケジューリング時間も含まれた時間としている。

```
const n = 1024
const n_time = 500
const tile_width = 512
real omega
real a(n,n)
processors AP(NAP)
distribute a(BLOCK,*) onto AP
        /* a() は1次元めでブロック分割 */

for time = 0, n_time - 1 remap 50 do
           /*反復(50反復毎に動的負荷分散)*/
  for outer_i = 0, n - 2, tile_width do
    for i = 1, n - 2 do
      /* 2次元めには幅tile_widthのタイリングを施す */
      for j = MAX(1, outer_i),
              MIN(n - 2, outer_i + tile_width - 1) do
        a(i, j) = (a(i - 1, j) + a(i, j - 1) \
           + a(i, j + 1) + a(i + 1, j)) * (omega / 4) \
           + a(i, j) * (1 - omega)
      endfor
    endfor
  endfor
endfor
```

図 23: SOR

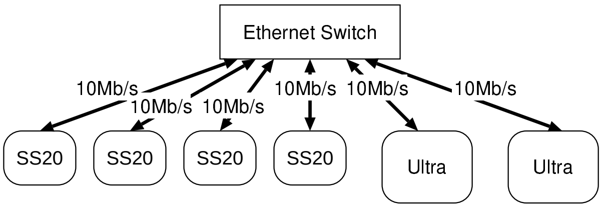


図 24: 評価に用いた計算環境

実行時のオーバヘッド　表で示されるように，行列積，SOR 共に本手法による非均質環境向けコードの実行時間は，従来の均質環境向けコードと比較して実行時間がほとんど低下していない。SOR において均質環境における実行時間が非均質環境での実行時間を上回っているが，これは通信時間が原因であると思われる。通信を除いた演算時間およびループ実行時間にはほとんど違いは見られない。また，ループスケジューリング時間も全体の実行時間や 1 ループあたりの実行時間と比較してほとんど無視できるといえる。特に反復法である SOR では，プログラム実行開始直後にのみスケジューリングを行うため実行時間に対する影響は無視できる。これらから，本手法では非均質なデータ分散に起因した処理のオーバヘッドが均質分散向けコードに生ずるオーバヘッドと同程度に抑えることができており，非常に良好な結果が得られたといえる。

抽象プロセッサ数の実行時間に対する影響　各プログラムに対して抽象プロセッサを変化させた時の実行時間にはほとんど差が見られたなった。特に SOR では，データの 1 要素を 1 抽象プロセッサとした時でもループの実行時間やループスケジューリング時間は全く変化していない。すなわち，通信相手特定のためのテーブル参照などのオーバヘッドはそのサイズに関係なく小さいものであることがわかった。

6.3 非均質環境におけるデータ分散比と実行時間の評価

次に，非均質環境においてさまざまなデータ分散比によりプログラムを実行させ，データ分散比が実行時間にどう影響するか

表 1: 計算環境の詳細

機種名	SPARCStation-20	Ultra-1
CPU	SuperSPARC 60MHz	UltraSPARC 166MHz
主記憶	64MB	128MB
OS	SunOS 5.5	SunOS 5.5
C コンパイラ	SPARCompiler C 3.0.1	
通信ライブラリ	MPICH ver. 1.0.12 (MPI ver. 1.1 準拠)	

表 3: 均質環境/非均質環境向けコードの実行時間

	分散手法	抽象プロセッサ数	実プロセッサ数	実行時間 (s)	演算時間 (s)	ループ実行時間 (s)	スケジュール時間 (ms)	台数効果
行列積	逐次	—	1	126.4	—	—	—	1.00
	均質	—	4	37.01	33.03	1.72	—	3.40
	非均質	4	4	37.14	32.78	1.72	0.2	3.40
	非均質	128	4	37.13	32.78	1.72	0.2	3.40
	非均質	256	4	36.79	32.80	1.70	0.2	3.43
SOR	逐次	—	1	300.11	—	—	—	1.00
	均質	—	4	100.32	78.82	12.3	—	2.99
	非均質	4	4	98.48	78.78	12.3	0.17	3.05
	非均質	128	4	98.42	78.82	12.3	0.15	3.05
	非均質	1024	4	98.31	78.58	12.3	0.15	3.05

を測定した。計算機は SS20 と Ultra を 2 台ずつ接続し，両者に分散させるデータの比率を変化させた。評価プログラムは行列積および 500 反復の SOR を使用し，抽象プロセッサ数は 16,32 の 2 種類とした。データ分散は実行開始時のパラメータとして指定し，実行中の負荷分散は一切行わないコードを利用した。

図 25(a) に行列積の，図 25(b) に SOR の実行時間のグラフを示す。x 軸は Ultra に分散させたデータ量の，全データ量に占める割合である。例えば $x = 0.5$ では Ultra，SS20 の各実プロセッサに対して抽象プロセッサ数 16 の時には 4 抽象プロセッサづつ，抽象プロセッサ数 32 の時には 8 抽象プロセッサづつ割り付けたことを意味する。また，Ultra の処理能力のほうが SS20 の処理能力より高いため，分散させるデータの比率は $0.5 \leq x < 1$ の範囲とした。グラフ中の演算時間とは SS20,Ultra 各計算機における，通信時間を除いた実行時間である。また均質環境との比較のため，SS20 を 4 台用いて同じプログラムを実行した時間と ULTRA4 台による実行時間も記した。

グラフより，SS20 と Ultra に対して同量のデータを配置した時には処理能力の低い SS20 の影響のため，SS20 を 4 台用いた計算環境からあまり性能が向上していないことが分かる。つまり，均質計算環境のようにデータを均等に分散させて実行しても，非均質環境において実行の高速化を図ることができない。しかし Ultra に与えるタスク量を増加させるにつれ，徐々に実行時間が短縮され，両者の演算時間がほぼ等しい $11:5$ の分散比率の時に最短実行時間となった。これは均質分散時の実行時間と比較して行列積，SOR でそれぞれ 33.2%，34.7%の性能向上となっている。すなわち，本手法により非均質環境においてより高速な実行を行うことができたといえる。

また，抽象プロセッサ数 16 の時と 32 の時と比較した場合には，分散比率が等しい位置での実行時間はほぼ同じとなった。しかし，抽象プロセッサ数が多いと分散の粒度を小さくでき理想的な分散比率に近づけることができる。図 25 の例でも，抽象プロセッサ数が 16 の時の最適分散比率 (3:1) における実行時間より抽象プロセッサ数が 32 の時の最適分散比率 (11:5) における実行時間のほうが短くなっている。

6.4 動的負荷分散機能の評価

最後に動的負荷分散機能の評価を行った。動的負荷分散機能には

1. あらかじめプロセッサ能力がわからない場合に最適な分散比を求める
2. 動的な負荷の変動に対応する

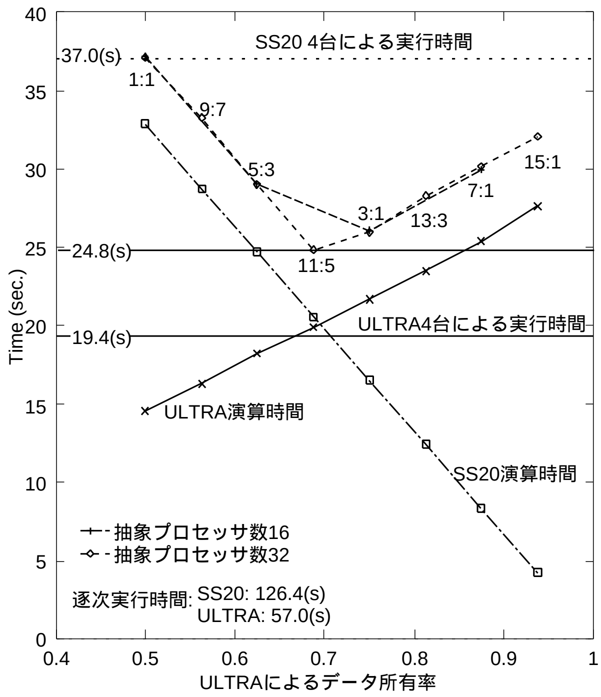
という 2 つの目的があるため，それぞれについて評価を行った。評価には，50 反復ごとに動的再分散を行う SOR を使用した。計算機は Ultra と SS20 を 2 台ずつ用いた。

6.4.1 実行開始時の再分散の評価

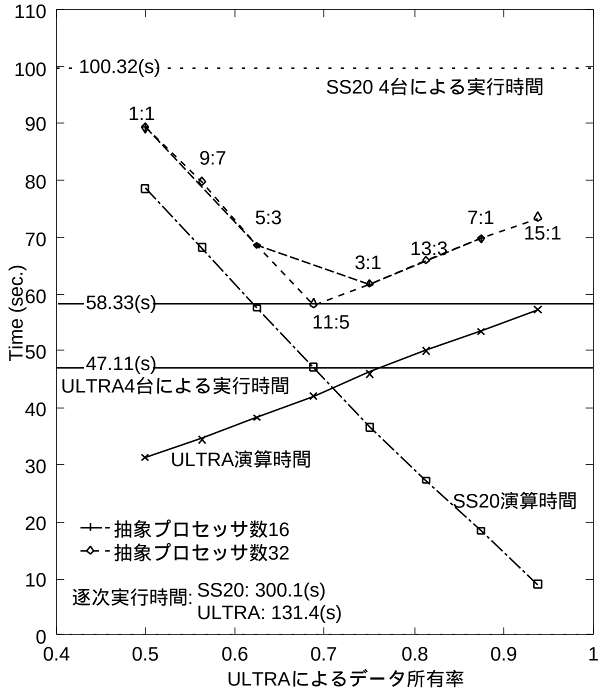
抽象プロセッサを全計算機に均等に分散させた状態で実行を開始し，動的再分散により最適な分散が行われ，実行時間が短縮されるか評価した。表 4 に結果を示す。表中の再分散時間は実行時間の計測や再分散に要した時間，括弧内は再分散回数である。静的分散実行時間とは，動的再分散により最終的に得られた分散比で実行を開始し，動的再分散を行わずに測定した実行時間である。また，図 26 に，全プロセッサに対するデータ分散比の時間的変化を示す。縦軸は，全データ分散比に占める各プロセッサのデータ量の比率であり，均質分散となる実行開始時は各プロセッサが 0.25 を示している。

図より，最初のデータ分散で性能の低い SS20 から高性能の Ultra にデータが移動し，SS20 の負荷の軽減が行われている。抽象プロセッサ数が 16, 32, 64 全ての場合において $1\sim3$ 回目の再分散で理想的な分散比に落ち着いており，以降はデータの再分散がないまま安定した実行ができているといえる。

静的分散と比較して動的負荷分散を行ったプログラムの実行時間が長いのは，実行開始直後の均質分散時における実行時間や動的再分散によるオーバヘッドのためである。



(a) 行列積



(b)SOR

図 25: 分散比率による実行時間の変化

表 4: 動的再分散による最適分散の決定

抽象プロセッサ数	実行時間 (s)	再分散時間 (s)[回数]	開始分散比 Ultra:SS20	最終分散比 Ultra:SS20	静的分散実行時間
16	67.4	2.8[1]	4:4	6:2	61.8
32	65.0	3.0[3]	8:8	11:5	58.3
64	65.6	2.7[1]	16:16	23:9	60.0

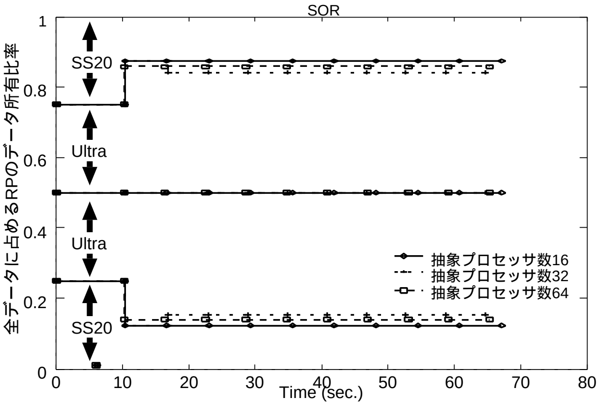


図 26: 動的再分散による分散比の推移

6.4.2 外的要因による再分散の評価

次に，外的要因によるプロセッサ負荷の変化に対応して動的な負荷分散が可能であるか評価した。評価プログラムは 2000 反復の SOR を用い動的負荷分散は 50 反復毎に行う。負荷は実行開始後 100 秒後に 1 台の実プロセッサに一定の負荷をプログラム実行終了まで加え続けた。

均質環境における負荷分散　SS20 を 4 台用いた均質環境において外的要因による負荷変動に対応できているかを評価した。表 5 に本手法により負荷分散分散を行ったものと，行わなかったもの実行時間を抽象プロセッサ数 16,64 について示す。また，図 27 に各実プロセッサにおけるデータ分散比の時間的推移を示す。

図より，特定のプロセッサに負荷を与える前は均質分散により実行が行われている。しかし 1 台に負荷が生ずるとそのプロセッサの処理能力が低下するため，他のプロセッサに対して徐々にデータを移動させ，抽象プロセッサ数 64 の時には最終的に 18:10 というデータ分散比に収束した。また，表から本再分散により抽象プロセッサ数の大小に関わらず実行時間が短縮していることが示される。しかし，抽象プロセッサ数が小さい場合には分散粒度が粗いため，1 抽象プロセッサの移動に必要なデータ量が多い。このため負荷変動時に大量のデータ移動が生し再分散に要する時間が大きくなっている。

しかし抽象プロセッサ数に関わらず，本動的負荷分散を用いることで外的要因に対しても柔軟に対応しており本手法により良好な性能向上が得られたといえる。

非均質環境における負荷分散　SS20 を 2 台，Ultra を 2 台用いた非均質環境における負荷分散の効果を評価した。負荷は 1 台の Ultra に対して加えた。本評価には，負荷分散を行うプログラム，均質分散で開始し負荷分散を行わないプログラム，無

表 5: 動的再分散による負荷分散 (均質環境)

抽象プロセッサ数	負荷分散機能	開始分散比 SS20:SS20:SS20:SS20	最終分散比 S:S(有負荷):S:S	実行時間 (s)	再分散時間 (s)[回数]	速度向上
16	無	4:4:4:4	—	694.3	—	—
16	有	4:4:4:4	5:2:5:4	556.9	40.3[32]	19.8%
64	無	16:16:16:16	—	700.5	—	—
64	有	16:16:16:16	18:10:18:18	527.6	15.6[26]	24.7%

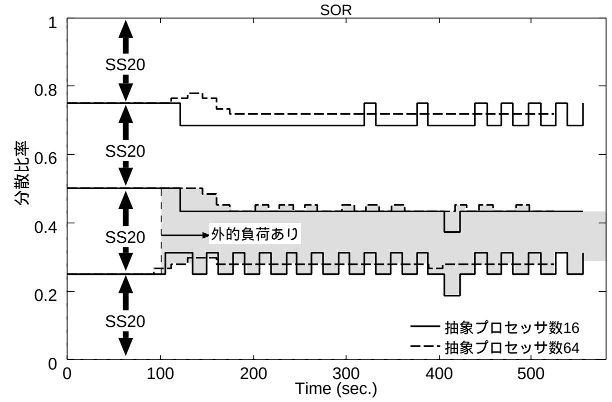


図 27: 動的再分散による分散比の推移 (均質環境)

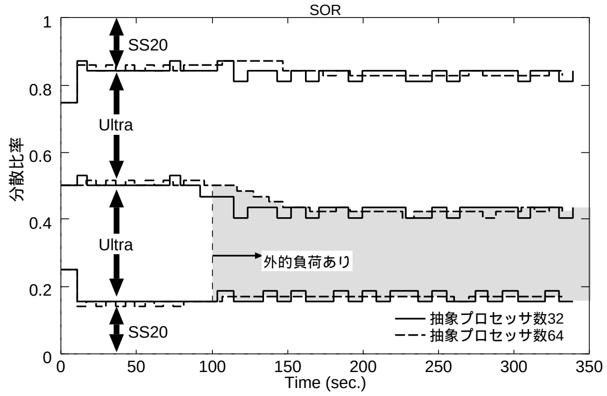


図 28: 動的再分散による分散比の推移 (非均質環境)

負荷状態での最適分散比で開始して負荷分散を行わないプログラムの 3 種類を用いた。表 6 に抽象プロセッサ数 32,64 のプログラム実行時間の表を示す。また，各プロセッサに対するデータ分散比の時間的変化は図 28 に示す。

図 28 より，動的再分散を使用することで負荷の高い Ultra に配分されている抽象プロセッサ数が無負荷の Ultra と比較して減少しており，本負荷分散による効果が示されているといえる。また，表からわかるとおり，非均質環境において外的負荷が生じている場合には無負荷状態で求めた分散比で実行を行うとかえって性能が低下してしまうこともありうる。しかし動的負荷分散の適用により均質分散と比較して実行時間を短縮させることができた。非均質環境においても，均質環境と同様，抽象プロセッサ数が多くなるにつれ理想的な分散比に近い形で負荷分散が行われるため実行性能の向上が見られる。抽象プロセッサ数が多いとデータ再分散単位も小さいため負荷分散時間も小さくなることが示される。

7　おわりに

本稿では，非均質環境を対象とした並列コード生成を行う並列化コンパイラ *hetero-TINPAR* について述べた。*hetero-TINPAR* はプロセッサの能力に合わせてデータ量を非均質に分散させることでタスクを各プロセッサの能力に比例させ非均質実行を行う。本手法ではデータをいったん抽象プロセッサという仮想的なプロセッサに分散させ，これを非均質な実際のプロセッサに割り付けることでテーブル参照や負荷分散を効率的に行うことができた。また，必要なデータ所有条件判定をあらかじめ行っておくことで非均質環境下でもループの実行オーバヘッドを極めて小さなものとすることができた。

性能評価により，本手法により非均質なデータ分散を行うことで，均質なデータ分散による実行と比較して 33 ～ 35%の実行時間の短縮を図ることがわかった。さらに，動的再分散機能により自動的にプロセッサ能力に応じたタスク分散を求めることができ，また計算機の非均質性や他プロセスによる負荷に対して柔軟に対応することで 6 ～ 25%の性能向上が得られた。また，抽象プロセッサ数を変化させることによる実行時オーバヘッドの違いは見られず，抽象プロセッサの粒度は動的負荷分散時のデータ移動時間に最も大きな影響を与えていることがわかった。抽象プロセッサ数が大きい時に良い負荷分散性能が得られているため，ユーザは抽象プロセッサ数を十分大きめに設定することでプログラムを高速に実行可能であるといえる。

今後の課題としては，最適な負荷分散アルゴリズムの改良が挙げられる。現在の負荷分散方式では，わずかな負荷の変動で抽象プロセッサが移動する問題が生じており，負荷に敏感に反応しないようデータ再分散のための閾値を設けるなどの処理や，動的負荷分散のデータ移動コストも考えたスケジューリング手法が必要となろう。

謝辞

本研究の機会を与えて下さった富田 眞治教授，適切な御指導を賜わった中島 浩助教授，森 眞一郎助教授に深甚な謝意を表します。また，貴重なご助言を頂いた五島 正裕助手に深く感謝致します。

さらに，共同研究者である窪田 昌史氏，辰巳 尚吾氏，田中 利彦氏をはじめとして，日頃ご討論頂いた京都大学工学部情報工学教室富田研究室の諸兄に感謝致します。

表 6: 動的再分散による負荷分散 (非均質環境)

抽象プロセッサ数	負荷分散機能	開始分散比 SS20:Ultra:Ultra:SS20	最終分散比 S:U(有負荷):U:S	実行時間 (s)	再分散時間 (s)[回数]	速度向上
32	無	8:8:8:8	—	357.4	—	—
32	無	5:11:11:5	—	364.0	—	−1.9%
32	有	8:8:8:8	5:9:13:5	342.1	25.2[31]	4.2%
64	無	16:16:16:16	—	357.6	—	—
64	無	9:23:23:9	—	386.3	—	−8.1%
64	有	16:16:16:16	11:16:26:11	335.9	12.9[32]	6.0%

参考文献

[1] 後藤慎也, 窪田昌史, 田中利彦, 五島正裕, 森眞一郎, 中島浩, 富田眞治: 並列化コンパイラ TINPAR による非均質計算機環境向けコード生成手法, 情処研報, Vol. 96, No. 107, pp. 7–12 (1996).

[2] T. E. Anderson, D. E. Culler, D. A. P. et al.: A Case for NOW(Networks of Workstations), *IEEE Micro*, Vol. 15, No. 1, pp. 54–64 (1995).

[3] 堀敦史, 手塚宏史, 石川裕, 曽田哲之, 小中裕喜, 前田宗則: 並列プログラム実行環境のワークステーションクラスタ上 での実装, 並列処理シンポジウム JSPP'96 論文集, pp. 49–56 (1996).

[4] 友清孝志, 小中裕喜, 石川裕, 前田宗則, 堀敦史: 超並列オブジェクトベース言語 OCore におけるメタレベル プログラミング, *Proc. JSSST 11th Annual Conf.*, pp. 117–120 (1994).

[5] Beguelin, A., Dongarra, J. et al.: HeNCE: Graphical Development Tools for Network-Based Concurrent Computing, *Proceedings, Scalable Hight Performance Computing*, pp. 129–136 (1992).

[6] Hiranandani, S., Kennedy, K. and Tseng, C.-W.: Evaluating Compiler Optimizations for Fortran D, *Journal of Parallel and Distributed Computing*, Vol. 21, No. 1, pp. 27–45 (1994).

[7] High Performance Fortran Forum: *High Performance Fortran Language Specification (Version 1.1 DRAFT)*, Rice University (1994).

[8] 三吉郁夫, 前山浩二, 後藤慎也, 森眞一郎, 中島浩, 富田眞治: メッセージ交換型並列計算機のための並列化コンパイラ TINPAR, 情報処理学会論文誌, Vol. 37, No. 7, pp. 1265–1275 (1996).

[9] 三吉郁夫, 前山浩二, 後藤慎也, 森眞一郎, 中島浩, 富田眞治: メッセージ交換型並列計算機のための並列化コンパイラ TINPAR –最適化手法と性能評価–, 情処研報 94-HPC-54, Vol. 94, pp. 45–52 (1994).

[10] Grimshaw, A. S., Weissman, J. B., West, E. A. and Ed C. Loyot, J.: Metasystems: An Approach Combining Parallel Processiong and Heterogeneous Distributed Computing Systems, *Journal of Parallel and Distributed Computing*, Vol. 21, No. 3, pp. 257–270 (1994).

[11] Zaki, M. J., Li, W. and Cierniak, M.: Performance Impact of Processor and Memory Heterogenetry in a Network of Machines, *Heterogeneous Computing Workshop*, IEEE Computer Society Press, pp. 101–108 (1995).

[12] Wolfe, M.: The Tiny Loop Restructuring Research Tool, *Proc. of 1994 Int'l Conf. on Supercomputing*, pp. 46–53 (1994).

[13] 辰巳尚吾, 窪田昌史, 五島正裕, 森眞一郎, 中島浩, 富田眞治: 並列化コンパイラ TINPAR における自動データ分割部の実現, 情処研報, Vol. 96, No. 82, pp. 25–30 (1996).

[14] 後藤慎也, 前山浩二, 三吉郁夫, 森眞一郎, 中島浩, 富田眞治: 並列化コンパイラ TINPAR におけるスカラ変数処理, 情処研報, Vol. 95, No. 82, pp. 41–48 (1995).

[15] 前山浩二, 後藤慎也, 三吉郁夫, 森眞一郎, 中島浩, 富田眞治: 並列化コンパイラ TINPAR における専用通信ライブラリの実装と性能評価, 情処学会第 50 回全国大会講演論文集 (1995).

[16] Message Passing Interface Forum: *MPI: Message-Passing Interface Standard* (1995).

[17] 窪田昌史, 三吉郁夫, 大野和彦, 森眞一郎, 中島浩, 富田眞治: 不規則アクセスを伴うループの並列化コンパイル技法–inspector/executor アルゴリズムの高速化–, 情報処理学会論文誌, Vol. 35, No. 8, pp. 532–541 (1994).

[18] 笠原博徳: 並列処理技術, コロナ社 (1991).

[19] Polychronopoulos, C. D. and Kuck, D. J.: Guided self-scheduling: a practical scheduling scheme for parallel supercomputers, *IEEE Transactions on Computers*, pp. 1425–1439 (1987).

[20] Evans, D. J. and Butt, W. U. N.: Dynamic Load Balancing Using Task-Transfer Probabilities, *Parallel Computing*, Vol. 19, No. 8, pp. 897–916 (1993).

[21] Leangsuksun, C., Potter, J. and Scott, S.: Dynamic Task Mapping Algorithms for A Distributed Heterogeneous Computing Environment, *Heterogeneous Computing Workshop*, IEEE Computer Society Press, pp. 30–34 (1995).

[22] Prouty, R.: Adaptive Execution of Data Parallel Computations on Networks of Heterogeneous Workstations, Technical Report CS-94-012, Oregon Graduate Institute CS (1994).

[23] Bridges, P., Doss, N., Gropp, W., Karrels, E., Lusk, E. and Skjellum, A.: *Users' Guide to MPICH, a Portable Implementation of MPI*, Argonne National Laboratory (1995).