並列化コンパイラ TINPAR による非均質計算環境向けコード生成手法

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[†]

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。本稿では，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う，非均質計算機環境向けのコード生成手法について述べる。データは最初に均質な抽象プロセッサに対し分散させ，処理能力に応じて実プロセッサに割り当てる。非均質分散の決定直後に，一度各実プロセッサにおいてループ中で実行すべきイタレーションを求め，リストとして保持しておく。このリストを参照しながらループを実行することで非均質実行によるオーバヘッドを削減する。さらに計算機の負荷を観測しながら，データの実プロセッサへの分散比率を動的に変更することで動的負荷分散機能も実現する。性能評価の結果，非均質計算機環境において低オーバヘッドな実行ができ，さらに動的負荷分散により約 6%の性能向上が示された。

Generating Optimized Code for Heterogeneous Computing Environment using Parallelizing Compiler TINPAR

Shin-ya GOTO,[†,☆] Atsushi KUBOTA,[†] Toshihiko TANAKA,[††]
Masahiro GOSHIMA,[†] Shin-ichiro MORI,[†]
Hiroshi NAKASHIMA[†,☆☆] and Shinji TOMITA[†]

In heterogeneous computing environment which consists of the processors with different computing power, computing tasks should be distributed in proportion to the computing power of processors. In this paper, we present a compiling technique to generate optimized codes for heterogeneous computing environment. First of all, data elements of arrays are 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.

Once the processor mapping is decided, each real processor makes lists of iterations to be executed on the processor for each loop. By reffering the iteration lists, loops are executed with low overheads in heterogeneous environment.

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.

As a result of the performance evaluation, we could confirm that the generated codes are effectively executed in the heterogeneous environment, and that the 6% speed up of execution time as a effect of the dynamic load balancing.

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。

しかしこのような環境であっても，多くのユーザは各計算機上で単独にジョブを実行させる程度であり，ネットワーク接続されたという利点を生かしきれてい

† 京都大学大学院工学研究科/Graduate School of Engineering, Kyoto University
†† 京都大学工学部/Faculty of Engineering, Kyoto University
☆ 現在，住友電気工業(株)/Sumitomo Electric Industries,Ltd
☆☆ 現在，豊橋技術科学大学/Toyohashi University of Technology

ない。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機環境(以下非均質環境と略す)であり，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生ずる。さらにマルチユーザ環境下では他ユーザプロセスの影響で計算機負荷が動的に変化する。

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の生成により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与える必要があり，今までのような一定量の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。

本稿では，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ンパイル手法を提案し，現在われわれが開発中の並列化コンパイラTINPAR[1),2)]に対する実装について述べる。

以下第2章では，非均質環境のための本並列化手法の概要について述べる。第3章でその処理内容について述べ，第4章で性能評価を行い第5章でまとめる。

2. 非均質計算機環境のための並列化手法

2.1 設計目標

各計算機の性能が異なり，しかも負荷が動的に変化するような非均質環境においてプログラムの実行を高速化できるよう，本非均質環境向きコンパイル手法の設計に関し次のような基本方針をとった。

(1) プロセッサ性能の非均質性/動的負荷変動に柔軟に対応する
異なる演算性能を持つプロセッサに対して，処理能力に比例したタスク量を分散させる。また，マルチユーザ下での処理能力の動的変動に対応できるよう負荷分散機能を実現する。

(2) 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
プロセッサ間で不規則な量のタスクが分散されるため，規則的に分散されたタスクを扱う均質環境と比較して実行時に余分なオーバヘッドが生ずる。このオーバヘッドを低く抑えるよう実装する。

(3) 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提案されてきた多くの最適化手法を有効に利用する。このため，均質環境向けコンパイラが多く採用してきた以下の前提のもとで並列化を行う。

[前提1] Owner Computes Ruleを用いてデータ分散とタスクの分散を一元化する

[前提2] 並列化コンパイラはSPMD型のコード生成を行う

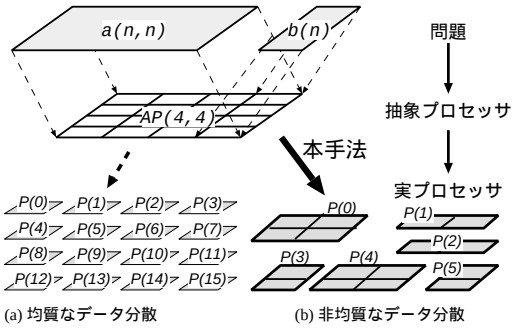

図1 抽象プロセッサによる非均質分散

以下に，われわれがこれらの目標を達成するために採用した設計戦略について述べる。最初に本手法におけるタスク分配の基本概念である抽象プロセッサについて述べ，次に実プロセッサへのタスク割り付け戦略，動的負荷分散戦略について記す。

2.2 抽象プロセッサ

本手法ではデータを実プロセッサに非均質に分散させるために抽象プロセッサという概念を用いる(図1)。

抽象プロセッサは，処理能力が全て等しい仮想的なプロセッサとし，コンパイラはデータ空間をいったん抽象プロセッサに対して分散させる。各抽象プロセッサに分散されたデータに対する演算処理(タスク)はこの時点では均質である。この抽象プロセッサを実プロセッサに対して処理能力に比例した個数だけ割り付け，非均質環境に対応させる(図1(b))。現在のところ，抽象プロセッサはHPF[3)]などにおける抽象プロセッサ(Abstract Processors)と同一のものとしている。

抽象プロセッサに対するデータ分散は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ため，従来のコンパイラからの拡張性を持たせることができる。一方，抽象プロセッサから実プロセッサへの割り付けに関しても，均質環境は非均質環境の特殊ケースとみなして両者を一元的に取り扱うことができる。

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当て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。また，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並列処理粒度をコントロール可能である。

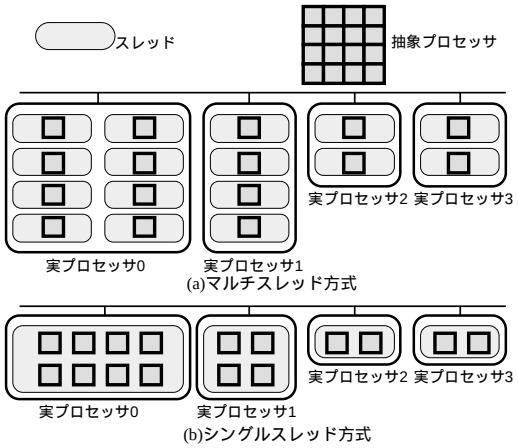

図 2　実プロセッサへの分散方式

2.3　抽象プロセッサの実プロセッサに対する割り付け戦略

複数個の抽象プロセッサを一つの実プロセッサに対して割り付けた場合の，実プロセッサのタスク処理方針としてマルチスレッド方式とシングルスレッド方式が考えられる。

マルチスレッド方式　図 2(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。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[4]。

シングルスレッド方式　図 2(b) のように，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。この方式では，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代入文にでき，またスレッド切り替えも生じない。しかし，スレッド内において複数の抽象プロセッサの処理に対する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に対する処理を矛盾なく扱うようなスレッド生成が必要である。

本手法ではシングルスレッド方式を採用した。これはマルチスレッド方式では実行時にスレッド切り替えのオーバヘッドが大きくなるためである。既存の OS をそのまま利用する場合，1 つのスレッド内で非均質なデータ量を処理するオーバヘッドは，OS の介在するマルチスレッド方式のスケジューリングオーバヘッドと比較して小さい。従ってマルチスレッド方式よりも実行時間を短縮できる。

しかし，シングルスレッド方式では不規則にデータが分散された配列に対して Owner Computes Rule を適用して，実行すべきイタレーションを決定するというスケジューリングが必要である。本手法ではこのスケジューリングを，抽象プロセッサの実プロセッサに対する割り当てが決定した時点で一度だけ行う。この時点では各実プロセッサが実行すべきイタレーションの集合を求め，それをリストとして保持する。実行時にはこのリストを参照しながらループを実行することで，非均質実行の実行時オーバヘッドを抑えることができる。本手法は，データのアクセス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[5]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。詳細は第 3.2節で述べる。

2.4　動的負荷分散

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[6] や GSS[6]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。

そこでわれわれは，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は現在は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実プロセッサでの処理が終わった時点とする。具体的には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。

本手法では複数の抽象プロセッサの処理が 1 スレッドにより行われており各抽象プロセッサの独立性は非常に低い。このため任意の時点で特定の抽象プロセッサのみをスレッドから切り出して他の実プロセッサに移動することは処理の上で大きなオーバヘッドが伴う。そこで全実プロセッサが同期可能な点をチェックポイントとしている。

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

協調の方法　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[7]。

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 3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。以下にそれぞれの概要について述べる。

2.4.1　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

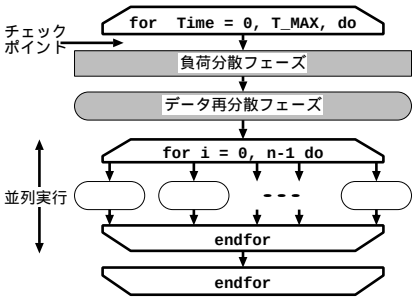

図 3　負荷分散の流れ

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。負荷分散フェーズでは次の点を考慮しながら最適なデータ量を求める必要がある。

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大量のデータを他のプロセッサに移動する必要があるが，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で求める最適分散は予測値であり，データ移動により移動先プロセッサの処理能力が予想以上に低下することもありうる。このため，一度に移動させるデータ量を制限し，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時間がかかるが安定した負荷分散が可能となる。

移動通信量と負荷の不均衡のトレードオフ　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いことがありうる。このためデータ再分散に必要な通信時間も考慮した予測が必要となる。

2.4.2　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する。データ移動に伴い抽象プロセッサの所属する実プロセッサが変更となるため，通信相手プロセッサの判定のためのテーブルなど，非均質分散に必要な情報も全実プロセッサ上で変更する。

3.　非均質計算機環境向けコード生成の手順

本章では，非均質環境のためのコード生成の手順について述べる。

本手法によって生成されるコードは，各実プロセッサが複数の抽象プロセッサのデータの計算を行うような SPMD 形式のコードである。これは抽象プロセッサに対してデータ分散が行われた後のコードを拡張する形で実装される。そのため，抽象プロセッサのための中間コードの生成までは従来の TINPAR の並列化・最適化手法を用いて行われる。抽象プロセッサのためのコードは均質環境向けに並列化されているため，本手法ではこの中間コードに対して

(1)　データを所有するプロセッサ判定および通信文の，非均質分散への対応づけ
(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)　動的データ再分散機能の挿入

といった処理を加えることで非均質環境向けのコードを生成する。以下，それぞれについて記す。

3.1　データを所有する実プロセッサ判定および通信相手プロセッサ判定

均質なデータ分散のもとでは，特定のデータを所有するプロセッサの判定 (この判定を行う関数を本稿では OWNER($data$) と表す) は 1 回の整数除算または剰余と整数加減算により行える。しかし，非均質なデータ分散時にはこのような式では表わせないため，抽象プロセッサとそれを所有する実プロセッサとの対応関係を，インデックステーブル AP_RP_Map[] として全プロセッサに持たせ参照することで行う。例えば

```
AP_RP_Map[8]={0,2,1,2,0,3,3,3}
```

とした時には，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 1 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。このテーブルを用いることで，抽象プロセッサ番号 AP_ID を用いた，データ a(i) を所有する抽象プロセッサの判定

```
if( OWNER(a(i)) == AP_ID ) then  ...
```

は，実プロセッサ番号 RP_ID を用いて

```
if( AP_RP_Map[OWNER(a(i))] == RP_ID ) then ...
```

とできる。また，均質環境では通信相手プロセッサの特定に関しても OWNER($data$) を使用して

```
Send( OWNER(a(i)), a(i+1) )
```

と表現しているため，同様に

```
Send( AP_RP_Map[OWNER(a(i))], a(i+1) )
```

とすることで通信相手の実プロセッサを求め，適切に通信を行う。テーブル参照は判定・通信のたびに行われるが，テーブルサイズは抽象プロセッサ数分であり，間接参照によるオーバヘッドが大きくはならない。

3.2　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

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。

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。例えば，図 4(a) のループを均質環境向けに並列化すると図 4(b) のようになり，実行すべきイタレーショ

```
real a(N),b(N)
processors AP(NAP)
distribute a(BLOCK) onto AP
distribute b(BLOCK) onto AP
for i = 0, N - 2 do
  a(i) = b(i+1)
endfor
```

(a) 逐次コード

```
real a(N),b(N)
if( PID != 0 )then
  i = (N/NAP)×PID-1
  Send( OWNER(a(i)), b(i+1) )
endif
for i = (N/NAP)×PID, (N/NAP)×(PID+1)-2 do
  a(i) = b(i+1)
endfor
if( PID != NAP-1 )then
  i = (N/NAP)×(PID+1)-1
  Recv( OWNER(b(i+1)), tmp )
  a(i) = tmp
endif
```

(b) 均質環境向けコード

図 4　ループの並列化

ンは静的に求まる。

しかし非均質環境では実プロセッサに複数の抽象プロセッサが割り付けられており，各実プロセッサはそれぞれ異なるサイズ 範囲のデータを持ち規則的なデータ配置とならない。また，動的負荷分散により抽象プロセッサの割り付けも変化する。このためループ中で実行すべきイタレーションは容易に求まらない。

本手法では 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ループの実行を高速に行う。さらに本手法では，同一実プロセッサ上に存在する抽象プロセッサ間のデータ参照を単なる代入文とし，抽象プロセッサ間に生ずるデータ参照のオーバヘッドをマルチスレッド方式と比較して大幅に削減できる。

以下に，本ループスケジューリング手法の基本方針および高速化手法について述べる。

3.2.1　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
(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 5 に本手法により図 4(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。

```
makeExecList(){
  ExecList = φ;
  for i=0, n-2 do
    ExecElem.condition = φ
    if( AP_RP_Map[ OWNER(a(i))]= RP_ID ) then
                        /*a(i) を所有*/
      ExecElem.condition に I_a を追加
    endif
    if( AP_RP_Map[ 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 ≠ φ ) then
          /*a(i) または b(i+1) のいずれか所有→実行*/
      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

```
/* 実行開始 */
ExecList = makeExecList()
/*ループ実行部*/
for ExecElem in ExecList do
  i = ExecElem.iteration
  switch ExecElem.conditon
    case I_a ∩ I_b:          /*a(i) も b(i+1) も所有*/
      a(i) = b(i+1)
    case I_a:                /*a(i) のみ所有*/
      Recv( AP_RP_Map[OWNER(b(i+1))], tmp)
      a(i) = tmp
    case I_b:                /*b(i+1) のみ所有*/
      Send( AP_RP_Map[OWNER( a(i))], b(i+1) )
  endswitch
endfor
```

(b) ループ実行部

図 5　非均質分散に対応したループ

図 5 に本手法により図 4(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，ループ実行に先立ち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自実プロセッサ上でローカル参照可能かを求める。いずれかがローカルに参照できるイタレーションの情報は，どのデータがローカル参照可能かの情報 I_a, I_b とともに $ExecList$ に追加する。リスト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 AP_RP_Map[] 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のみを実行する。このため，実プロセッサが所有するデータを参照しないイタレーションの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，ループ実行部では switch 構文により必要となるデータ参照パターンを選択する。

本手法では，実プロセッサが所有している複数の抽象プロセッサ間にまたがる代入は，ループスケジューリング部によってローカルな代入可能 (図の例では $I_a \cap I_b$) と判断されるため，抽象プロセッサ間通信は単なる代入文とできる。

また，リスト作成は実プロセッサに対する抽象プロセッサの分散が決定した時点で 1 回行うのみでよいため，同一のループを繰り返し実行する場合にはリスト生成時間が無視できる。なお，多次元分割された抽象プロセッサ上のデータをアクセスする多重ループにおいても，同様な手順により実行可能である。

3.2.2　スケジューリングおよび実行コストの削減

図 5に示した方針のままでもある程度高速実行可能である。しかし図 5ではリスト作成時に全イタレーションが実行され効率が悪い。ループ実行部も各イタレーションで switch 構文による判定があ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りスケジューリングおよび実行コストを削減し高速化を図る。

3.2.2.1　スケジューリングコストの削減

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 ·均質性を利用することで可能である。

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データを所有しているため，実プロセッサの所有データ領域や実行可能なイタレーションは割り付けられた抽象プロセッサ番号から簡単な式により求まる。例えば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 の範囲は

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

と求められる[1)]。実プロセッサは，自分に割り付けられた各抽象プロセッサについてこのようにして求めたイタレーションを合成し最終的なイタレーションリストを生成する。この方式では 5(a) のように全イタレーションを実行することなく高速にリスト作成可能である。

データがサイクリック分割により抽象プロセッサに割り付けられている場合には，データの分割次元は周期 NAP で同じ抽象プロセッサに所有されているため，周期 NAP で同じ実プロセッサ上に存在している。この周期性を利用することで，ループスケジューリング部では最初の 1 周期分 (長さ NAP) のイタレーションリストを作成するだけでループ全体がスケジューリング可能である。

3.2.2.2　ループ実行コストの削減

図 5(b) ではイタレーション毎にどのようなデータ参照があるかを switch 構文により判定している。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ような場合は，実プロセッサ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 つのリストの要素として扱うことで，複数のイタレーションを条件判定なく連続実行する (図 6)。図の例では，イタレーションリストに参照パターン $condition$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までを連続実行することで高速化を図っている。

```
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 ∩ 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) = b(i+1)
      endfor
    case I_a:                   /*a(i) のみ所有*/
      for i = l, u
        Recv( AP_RP_Map[OWNER(b(i+1))], tmp)
        a(i) = tmp
      end
    case I_b:                   /*b(i+1) のみ所有*/
      for i = l, u
        Send( AP_RP_Map[OWNER(a(i))], b(i+1) )
      endfor
  endswitch
endfor
```

図 6　ループ実行コストの削減 (ループ実行部)

3.3　動的データ再分散

本手法では，実行中に実プロセッサの処理性能を測定し，これに応じてデータの再分散を行うことで動的な負荷分散を行う。

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る。現在のところ SOR などの反復法における最外ループ (時間軸) において特定のイタレーション毎に再分散を行うことができる。

図 7に本動的再分散の手順を示す。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。$AP_k(t)$ が決定すると実プロセッサに対する抽象プロセッサの割り付けを行う。このとき，対象となる抽象プロセッサの持つデータの移動とともに，全実プロセッサ上のインデックス AP_RP_Map[] の変更およびイタレーションリストの作成が必要となる。

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new_AP_RP_Map[] 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]new_AP_RP_Map[] 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AP_RP_Map[] 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 7　動的データ再分散の手順

4. 性能評価

本稿で述べた並列化手法により，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。

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) と SPARC Ultra-1(以下 Ultra) を最大 4 台構成で用い，原則として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 MPI[8]の実装処理系である MPICH[9]を使用した。評価プログラムは 512×512 の行列積及び 1024×1024，500 反復の SOR を用いた。ともに行列データは抽象プロセッサに対し列方向にブロック分割したものを，1 次元の実プロセッサに対して分散させた。また，実プロセッサ間の分散比および抽象プロセッサ数 (AP 数) は評価により変化させた。評価は，1) 非均質環境向けコードと均質環境向けコードの実行時間の計測，2) 異なる性能の計算機に対するデータ分散比を変化させた場合，3) 再分散を行った場合 について行った。

表 1　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	実行時間 (s)	リスト作成時間 (ms)	台数効果
行列積		1 (逐次実行)	126.4	—	—
	均質	4 (1×4 台)	37.01	—	3.40
	非均質	4 (1×4 台)	37.14	0.2	3.40
	非均質	128 (32×4 台)	37.13	0.2	3.40
	非均質	256 (64×4 台)	36.79	0.2	3.43
SOR		1 (逐次実行)	300.11	—	—
	均質	4 (1×4 台)	100.32	—	2.99
	非均質	4 (1×4 台)	98.48	0.17	3.05
	非均質	128 (32×4 台)	98.42	0.15	3.05
	非均質	1024 (256×4 台)	98.31	0.15	3.05

4.1 均質環境向けコードと非均質環境向けコードの実行時間の比較

最初に，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た。非均質環境向けコードのデータ分散比は均等にし，ループのイタレーションリスト作成に要する時間についても求めた。表 1に SS20 4 台からなる均質環境における実行時間を示す。

表で示されるように，行列積 ･SOR 共に非均質処理によるオーバヘッドによる速度低下はほとんど生じていないことが分かる☆。また，イタレーションリストの作成はプログラム実行開始直後に一度しか行わないため実行時間に対して影響していない。

4.2 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，さまざまな分散比のもとでの実行時間を測定した。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評価した。図 8に SOR における実行時間を示す。グラフの 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。またグラフ中の演算時間とは SS20,Ultra 各計算機における，通信時間を除く実行時間である。行列積，SOR ともに $x = 0.5$(SS20 と Ultra が同量の抽象プロセッサを所持) では処理能力の低い SS20 の影響のため SS20×4 台の環境とほぼ同じ実行性能しか得られていない。しかし Ultra の計算量を増加させるにつれ実行時間が短縮され，両者の演算時間がほぼ等しい 11 : 5 の分散比率の時が最短実行時間となった。なお，行列積に関しても同様の結果が得られた。

4.3 動的再分散の評価

最後に動的再分散手法の評価を行った。評価は，1) あらかじめ能力がわからない場合に最適値に収束するかどうか 2) 動的な負荷変動への対応 について行った。

評価には，50 反復ごとに実行時間の計測 ･新しい分散の計算および再分散を行うようディレクティブに

☆ 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はほとんど時間差は生じていない。

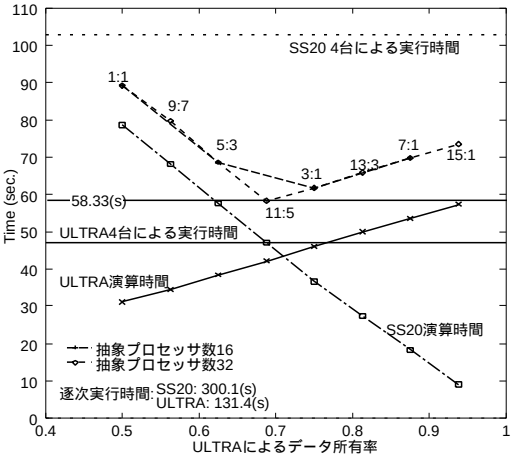

図 8　分散比率による実行時間の変化 (SOR)

表 2　動的再分散による最適分散の決定

AP 数	実行時間	再分散時間	開始分散比	最終分散比	静的分散
	(s)	(s)[回数]	Ultra:SS20	Ultra:SS20	実行時間
16	67.44	2.76[1]	4:4	6:2	68.58
32	64.99	3.03[3]	8:8	11:5	58.33
64	65.57	2.65[1]	16:16	23:9	59.97

よって指定した SOR を使用した。計算機は Ultra と SS20 を 2 台ずつ用いた。

4.3.1　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 2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に初期化，動的再分散を行わずに測定した実行時間である。抽象プロセッサ数が 16, 32, 64 全ての場合において 1〜3 回目の再分散で最終的な分散比に落ち着いた。静的分散と比較して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。

4.3.2　外的要因による再分散の評価

次に外的負荷による動的負荷分散の評価を行った。評価は 2000 反復の SOR を用い，抽象プロセッサ数 32 とした。計算機は Ultra 2 台と SS20 2 台であるが，このうち Ultra 1 台には実行開始 100 秒後にユーザプロセスを起動し負荷をかけ続けた。評価は，①均等な分散で開始し負荷分散を行わないもの②無負荷状態で最良の性能を示した分散比 9:23 により開始し負荷分散を行わないもの③均等な分散で開始し負荷分散を行うもの を用いた。これを表 3に示す。計算機に負荷が生じた場合，動的負荷分散により約 6%実行時間が短縮した。表では負荷の高い 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，負荷分散による効果が示されたといえる。

表 3　動的再分散による負荷分散

プログラム	実行時間	再分散時間	最終分散比
	(s)	(s)[回数]	Ultra:Ultra(有負荷):SS20:SS20
①	357.6	—	16:16:16:16
②	386.3	—	23:23:9:9
③	335.9	12.9[32]	26:16:11:11

5. おわりに

本稿では，非均質環境においてプロセッサの能力に合わせたデータ量を分散させることで，実行を高速化するための並列化コンパイル手法について述べた。ループ実行時に必要なデータ所有条件判定をあらかじめ行うことで非均質環境下で低オーバヘッドに実行可能であることを示した。また，動的再分散機能によりプロセッサ能力に応じた分散比による実行ができた。

今後の課題としては，最適な負荷分散アルゴリズムの研究がある。現在の方式では，微妙な負荷変動で抽象プロセッサが移動する問題が生じているため，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処理や，負荷分散のデータ移動コストも考慮したスケジューリング手法を研究する予定である。

謝辞　日頃より御討論いただく京都大学工学部情報工学教室富田研究室の諸氏に感謝致します。

参考文献

1) 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, July 1996.
2) 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 PRO-10-2, pp.7-12, October 1996.
3) High Performance Fortran Forum: "High Performance Fortran Language Specification (Version 1.1 DRAFT)," Rice University, 1994.
4) Grimshaw, A. S., Weissman, J. B. et al.: "Metasystems: An Approach Combining Parallel Processing and Heterogeneous Distributed Computing Systems," Journal of Parallel and Distributed Computing,21(3).pp.257-270, 1994.
5) 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, April 1994.
6) 笠原 博徳: 並列処理技術, コロナ社, 1991
7) Evans, D. J., Butt, W.U.N.: "Dynamic Load Balancing Using Task-Transfer Probabilities," Parallel Computing, Vol. 19, No. 8, pp.897-916, 1993.
8) "MPI: Message-Passing Interface Standard," Message Passing Interface Forum, 1995
9) Patric Bridges et al.: "Users' Guide to MPICH, a Portable Implementation of MPI," Argonne National Laboratory, 1995