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，これを処理能力に応じて実プロセッサに割り当てる。連続した抽象プロセッサを割り当てることによりオーバヘッドの少ないループ実行ができる。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高速な実行ができ，さらに動的負荷分散により13%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 this paper, we present a compiling technique to generete optimized codes for heterogeneous computing environment. Here we assume the compiler follows the data owner compute rule, thus the amount of data assigned to a processor directory reflects the amount of computing task of it.

In our compiler, data objects are first distributed onto a homogeneous abstract processors, and these abstract processors are mapped onto real processors so that the amount of abstract processors assigned to a real processor is in proportion to the available computing power of the real processor. During the computation, available computing power of each proecessor is evaluated occasionally and remapping of abstract processors is performed to achieve optimal load distribution at that time. These mechanisms for dynamic load balance are also integrated into the genarated code.

As a result of the performance evaluation, we could confirm that the generated code effectively runs in the heterogenious environment, and that the 13 % speed up of execution time as a effect of the dynamic load balancing.

1. はじめに

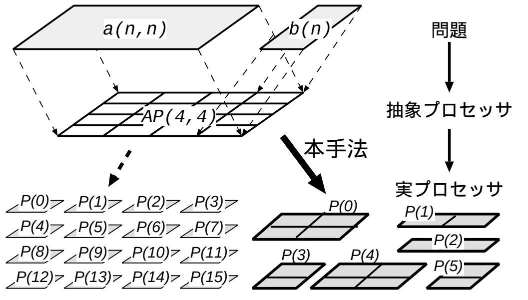
近年，ワークステーションやパーソナルコンピュータなどの安価で高性能な計算機が普及してきており，またこれらの計算機どうしはEthernetやさらに高速なLANによって接続されている。こういった計算機群の全体または一部を用いて仮想的な並列計算機を構築する試みがなされている。このようなシステムはスーパコンピュータと比較しても比較的安価に高性能の環境を構成できるといった利点がある。

しかしこのような並列計算環境であっても，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などの影響で各計算機の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，

---

† 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



(a) 均質なデータ分散　(b) 非均質なデータ分散

図 1　抽象プロセッサによる非均質分散

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 TINPAR1) 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) 均質環境向き並列化コンパイラからの拡張性を持たせる。
(2) データ分散の最適化により，非均質性に起因するオーバヘッドを最小とする。
(3) プロセッサの動的負荷変動/非均質性に柔軟に対応する。

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。なお，本稿では，以下の前提のもとで議論を行う。

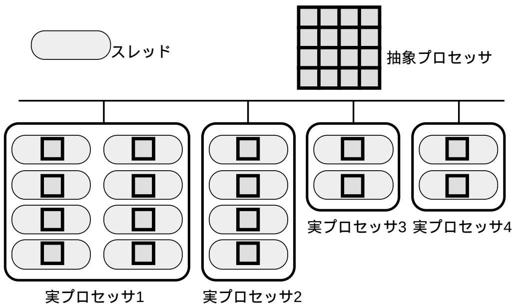
[前提 1] Owner Computes Rule を用いることでデータ分散と処理の分散を一元化する

[前提 2] 並列化コンパイラは SPMD 型のコード生成を行う。

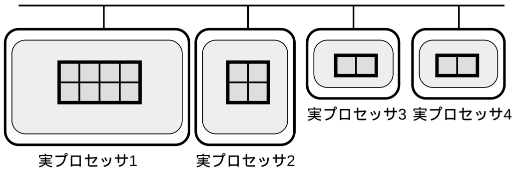
2.2 抽象プロセッサ

本手法では与えられたデータを各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 参照)。分散メモリ計算機用のデータ並列記述言語である HPF2) では，与えられたデータを抽象プロセッサという仮想的なプロセッサに割り当てる。実プロセッサにはこの抽象プロセッサを割り付ける形で分散される。本手法では，各実プロセッサに対してこの抽象プロセッサを処理能力に比例した個数だけ割り付けることで非均質環境に対応させる (図 1(b))。

抽象プロセッサに対するデータの分散に関しては従来のコンパイル手法をそのまま適用できるため，均質環境向きの並列化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。



(a)マルチスレッド方式



(b)単一スレッド方式

図 2　実プロセッサへの分散方式

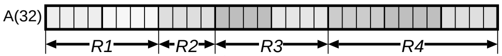
2.3 抽象プロセッサの実プロセッサに対する割り付け戦略

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単一スレッド方式が考えられる。

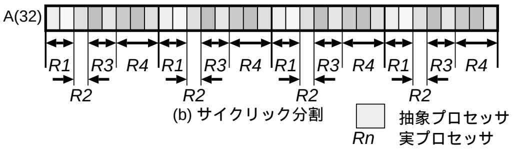
マルチスレッド方式　図 2(a) のように，複数の抽象プロセッサをそのままの形で実プロセッサ上に分散させる。各実プロセッサはそれぞれの抽象プロセッサを独立して扱い，複数スレッド (軽量プロセス) または実行処理系によるスケジューリングによって抽象プロセッサのタスクを実行する。同じ実プロセッサ内の複数の抽象プロセッサどうしの通信は，スレッド間の通信として実現する。この方式では各抽象プロセッサの独立性が高く，各スレッドは均質な抽象プロセッサの処理を行うことができるため，コンパイラは均質な処理を行うようなスレッドを生成すればよい3)。

単一スレッド方式　図 2(b) のように，複数の抽象プロセッサを一つにまとめて実プロセッサに割り付け，実プロセッサはこの抽象プロセッサのかたまりを一つのスレッドとして処理する。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。この方式ではスレッドが単一であるため実プロセッサは実行時にスレッドの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である。しかし，各スレッド毎に異なるサイズのデータ領域を持つことになり，ループ中の実行すべきイタレーションの範囲などがスレッドによって異なってくる。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。

本手法では単一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ことを考えると，処理粒度の非均質性に伴う実行時のオーバ



(a) ブロック分割



(b) サイクリック分割

図 **3**　実プロセッサに対する抽象プロセッサの割り付け
[抽象プロセッサ数 8, 実プロセッサ数 4(性能比 2:1:2:3)]

ヘッドは，スレッド切り替えやスケジューリングによるオーバヘッドと比較して一般に小さい。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。

さらに本手法では，連続した抽象プロセッサを実プロセッサに対して割り付けることで，複数の抽象プロセッサの実行を連続して行う (図 3 参照)

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4) や GSS4)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共有メモリを持つシステムでの doall 型の並列実行を前提としているため，我々が前提としている分散型計算機環境への対応が困難である。

これは，依存を解決するための「待ち」や「待ちからの復帰」処理の問題や，現在実行されていないあるいは待ち状態にあるタスクが，将来「どの実プロセッサ」で「いつ」実行されるかが実行時にならないと不明であるという問題，さらには Owner Computes Rule との親和性の悪さの問題等があるからである。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 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。

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　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。

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 4 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を行う。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の予測を立て，最適な分散比率を決定する。この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。

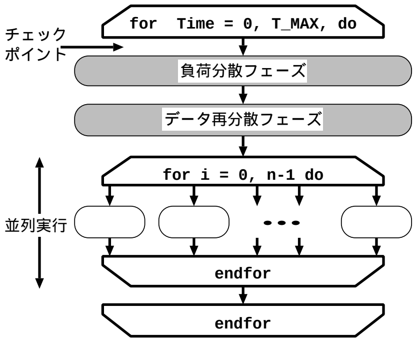


図 **4**　負荷分散の流れ

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が出てくる。しかし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分散はあくまで予測であるため，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に制限を加え最適状態に順次移行していく方針を取る。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。

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，負荷を均衡化するために再分散を行って処理を続けた時の実行時間よりも，再分散しないで負荷不均衡のまま実行した時間の方が短くなることもありうる。このためデータ再分散に必要な通信時間も考慮した性能予測が必要となる。

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した移動を行う。この際，移動の際には連続した抽象プロセッサが実プロセッサに対して割り付けられるような移動を行わなければならない。また，移動を行うことにより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，同時に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ためのループ・通信文

☆ 各実プロセッサは 1 スレッドで複数の抽象プロセッサの処理を実行している。そのため実プロセッサ内部ではある時間において唯一の抽象プロセッサの処理を行っている。そこで，一部の抽象プロセッサのみの処理を終えた所で同期を取り再分散を行うと，抽象プロセッサによっては未実行の部分があるままで他実プロセッサに移動することになるため，その後の実行が困難となる。

```
real a(0:n-1:BLOCK)
processors N_AP
/* blockSize = n/N_AP */

for i = 0, n - 1 do
      a(i) = a(i) + 5
endfor
```

⇓

```
for i=blockSize× MY_START_AP,
     blockSize×(MY_START_AP+MY_OWN_AP)-1 do
      a(i) = a(i) + 5
endfor
```

非均質分散に対応したループ

図 5　ループの実行範囲の変更

の対応

(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

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 (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のもとで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[] として各プロセッサに持たせ参照することで行う。図 3 の例ではいずれも

```
AP_RP_Map[8]={1,1,2,3,3,4,4,4};
```

となる。これを用いて，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判定のための条件式

```
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。このテーブル参照は判定のたびに行うが，テーブルサイズはたかだか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　非均質分散への対応

非均質分散によって，各実プロセッサはそれぞれ異なったサイズのデータを持つことになるため実行文にたいして非均質データを扱うための変更を加える。具体的には

(1)　非均質データをアクセスするループの初期値・終了値の変更

(2)　通信相手プロセッサの非均質分散への対応

を行う。以下これらについて述べる。

3.2.1　非均質データをアクセスするループの初期値・終了値の変更

本手法では連続した抽象プロセッサを実プロセッサに割り付けることでループ内の複数の抽象プロセッサのイタレーションを連続実行し，実行時のオーバヘッドの低減を行う。例えばブロックサイズ $bsize$ のブロック分割において連続する n 個の抽象プロセッサを所有することにより，実プロセッサはあたかもブロックサイズ $n \times bsize$ のデータを所有しているかのように扱うことができる。これによりループの初期値と終了値は，各実プロセッサの所有する先頭の抽象プロセッサ番号MY_START_AP，および所有抽象プロセッサ数MY_OWN_AP という 2 変数から求められる (図 5)。本手法により多数の抽象プロセッサのループを連続化することでループの初期値・終了値の判定を 1 回で済ますことができ，実行時のオーバヘッドが抑えられる。この処理は任意幅のブロックサイクリック分割に対して適用できる。

```
real a(0:n-1:BLOCK)
processors N_AP

for i = 0, n - 2 do
    a(i) = a(i+1)
endfor
```

(a) 通信を伴うループ

```
i = blockSize× MY_START_AP-1
Send( AP_RP_Map[OWNER(a(i))], a(i+1) )
                 /*a(i+1) を a(i) の所有者に送信*/
for i= blockSize×MY_START_AP,
   blockSize×(MY_START_AP+MY_OWN_AP)-2 do
   a(i) = a(i+1)  /* ローカルな計算 */
endfor
i = blockSize×(MY_START_AP+MY_OWN_AP)-1
Recv( AP_RP_Map[ OWNER(a(i+1))], a(i) )
                 /*a(i+1) を受信し a(i) に代入*/
```

(b) 実プロセッサにおける通信処理

図 6　実プロセッサ間通信の作成

3.2.2　プロセッサ間通信の非均質分散への対応

抽象プロセッサに対し分散されたコードには抽象プロセッサ間の通信が含まれているため，これを実プロセッサ間通信に変更する。抽象プロセッサのコードには OWNER($data$) を用いた抽象プロセッサ番号が通信相手として指定されている。そのためインデックスAP_RP_Map[] を用いて抽象プロセッサ番号から通信相手の実プロセッサ番号を求める。

図 6 に例を示す。(a) の逐次コードを並列化した場合，非均質環境に対応させるため，送受信文 (Send, Recv) は AP_RP_Map[] を用いて実プロセッサ間通信に置換を行う。また，3.2.1 で述べた複数の抽象プロセッサのループを連続して実行することにより，同じ実プロセッサ上にある抽象プロセッサ間の通信が規則的であれば消去できることがある。このような通信を消去すると最終的に (b) のようなコードが生成される。

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。

負荷分散を行うための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) に対し，特定のイタレーション毎に行うことができる。また現段階では実プロセッサが 1 次元の場合の動的再分散が実装済みである。

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が，この時連続した抽象プロセッサを割り付ける必要がある。再分散時には，対象となる抽象プロセッサの持つデータを移動させるとともに，インデックスAP_RP_Map[] およびイタレーションの実行判定に用いる変数MY_START_AP，MY_OWN_AP も変更する。

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。

$$Restim_1(t) = Restim_2(t) = ... = Restim_{N_{RP}}(t)$$

$$\text{ただし } Restim_k(t) = Acomp_k(t) \times AP_k(t)$$

$$\sum_{k=1,..N_{RP}} AP_k(t) = N_{AP}$$

(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以下の変数を変更する。
    - インデックス`AP_RP_Map[]`
    - 所有する先頭の抽象プロセッサ番号`MY_START_AP`
    - 所有する抽象プロセッサ数`MY_OWN_AP`
  - 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。

(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，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[5]の実装処理系である MPICH[6]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

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 共に本非均質処理のテーブル参照によるオーバヘッドがほとんど存在しないことが分かる。抽象プロセッサ数を増加させることによりわずかながら実行時間が多くなっているが 1%未満であり有意な差であるとは考えられない。

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	行列積		SOR	
		実行時間 (s)	台数効果	実行時間	台数効果
	1 (逐次実行)	122.71	—	296.96	—
均質	4 (1×4 台)	32.99	3.72	103.13	2.88
本手法	4 (1×4 台)	33.01	3.72	103.42	2.87
本手法	128 (32×4 台)	33.01	3.72	103.54	2.87
手法	256 (64×4 台)	33.04	3.71	103.78	2.86

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，通信時間を除いた実行時間である。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，だんだん実行時間が短縮され，両者の演算時間はほぼ等しくなる 11 : 5 の分散比率の時が最短実行時間となった。なお，行列積のプログラムに対しても同じ評価を行ったが同様の傾向が見られた。

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でも最適値に収束するかどうか 2) 動的な負荷の変動時の対応 について行った。

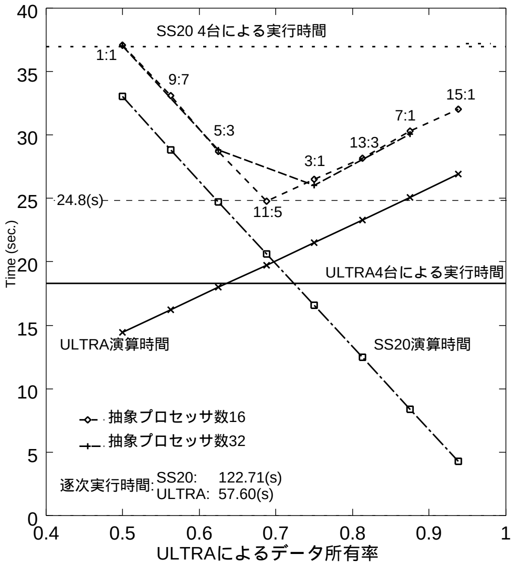
評価には，50 反復ごとに実行時間の計測 ·新しい分散の計算および再分散を行うようディレクティブによって指定した SOR を使用した。計算機は Ultra と SS20 を 2 台ずつ用いた。

4.3.1 実行開始時の再分散の評価

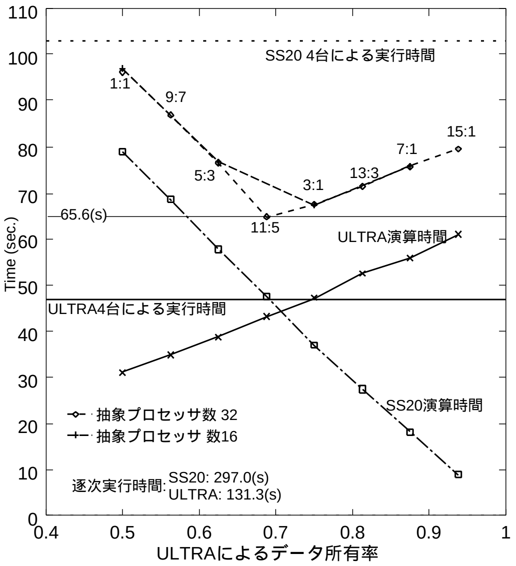
抽象プロセッサを全計算機に均等に分散させた状態で実行を開始し，動的再分散により最適な分散が行われ，実行時間が短縮されるか評価した。表 2 に結果を示す。表中の再分散時間は実行時間の計測や再分散に要した時間，括弧内は再分散回数である。静的分散実行時間とは，動的再分散により最終的に得られた分散比に初期化，動的再分散を行わずに測定した実行時間である。抽象プロセッサ数が 16, 32, 64 全ての場合において 1 回目の再分散で最終的な分散比に落ち着いたが，抽象プロセッサ数 64 では分散の粒度が小さいために以降も微妙な負荷の変動により再分散をくり返した。静的分散と比較して実行時間が長いのは，実行開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。特に抽象プロセッサ数 64 では収束に時間がかかったためその差は大きくなっている。

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な負荷分散の評価を行った。評価プログラムは 2000 反復の SOR を用い，抽象プロセッサ数は 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 台にはユーザプロセスを起動し負荷をかけ続けた。評価は，①無負荷状態で最も良い性能を示した比率



(a) 行列積



(b)SOR

図 8　分散比率による実行時間の変化

5:11 を用いて動的再分散を行わなかったもの ②比率 5:11 で開始し動的再分散を行ったもの③均等な分散で開始し動的再分散を行ったもの について行った。これを表 3 に示す。計算機に負荷が生じた場合，本動的負荷分散を使用することにより 13%実行時間が短縮していることがわかる。また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。しかし，負荷分散全体 (40 回) のうち過半数においてデータ再分散を行っており，わずかな負荷の変動で抽象プロセッサが移動するという問題が生じている。

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 (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	73.73	2.95(1)	4:4	6:2	67.14
32	69.92	2.72(1)	8:8	11:5	65.60
64	72.48	3.45(5)	16:16	23:9	64.80

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 (回数)	最終分散比 Ultra:Ultra(有負荷):SS20:SS20
①	468.5	—	11:11:5:5
②	406.7	23.8(25)	13:8:6:5
③	416.3	24.6(25)	13:8:6:5

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，実行時間の高速化を行うための並列化コンパイル手法について述べた。連続した領域の抽象プロセッサを実プロセッサに分散させることにより，高速なループの実行やデータの所有者判定ができることを示した。また，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分散を得ることができた。

今後の課題としては，多次元構成の実プロセッサに対して抽象プロセッサを矩形領域として割り当てるための手法の検討や，最適な負荷分散アルゴリズムの研究などがある。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という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) High Performance Fortran Forum: “High Performance Fortran Language Specification (Version 1.1 DRAFT),” Rice University, 1994.
3) Grimshaw, A. S., Weissman, J. B. et al.: “Metasystems: An Approach Combining Parallel Processiong and Heterogeneous Distributed Computing Systems,” Journal of Parallel and Distributed Computing,21(3).pp.257-270, 1994.
4) 笠原 博徳: 並列処理技術, コロナ社, 1991
5) “MPI: Message-Passing Interface Standard,” Message Passing Interface Forum, 1995
6) Patric Bridges, Nathan Doss, William Gropp, Edward Karrels, Ewing Lusk , Anthony Skjellum: “Users’ Guide to MPICH, a Portable Implementation of MPI,” Argonne National Laboratory, 1995