データ並列言語を対象とした動的負荷分散機構の実現と評価

荒木拓也[†,☆] 村井均[†,☆☆]
蒲池恒彦[†,☆] 妹尾義樹[†,☆]

近年，複雑化した数値シミュレーションを高速に実行するシステムとして，異なった種類の並列計算機を結合した異機種並列分散システムが注目されている．しかし，このようなシステムでは各計算機の速度が異なるため，適切な負荷分散を行わなければ高性能は実現できない．また，複数のユーザで共用される計算機で構成されたクラスタにおいても，ほかのジョブの影響でプロセッサごとの速度が変わる可能性があるため，負荷分散が重要となる．しかし，従来は分散メモリシステムにおいて数値計算プログラムの動的負荷分散を行うには，MPI ライブラリを用いた複雑なプログラミングを行う必要があった．この問題を解決するため，我々は，HPF（High Performance Fortran）を拡張し，容易に動的な負荷分散を行うことができるシステムを実現した．本システムでは，繰り返し実行される計算の各実行時間を計測することで，システムの速度を推定する．推定した速度に応じて自動的にプロセッサに割り当てるデータ量を変更することで，負荷分散を実現した．本研究ではこのシステムを実装し，SX-4，Cenju-4 から構成される異機種システム，PC クラスタ，Cenju-4 から構成される異機種システム，および負荷をかけた PC クラスタ上で評価を行った．評価により，本システムによって高い速度向上が得られることを確認した．

Implementation and Evaluation of Dynamic Load Balancing Mechanism for a Data Parallel Language

Takuya Araki,[†,☆] Hitoshi Murai,[†,☆☆] Tsunehiko Kamachi[†,☆] and Yoshiki Seo[†,☆]

In recent years, heterogeneous parallel and distributed systems which are composed of various kinds of computers are considered promising for high performance execution of large, complex numerical simulation. However, proper load balancing is critical in order to extract high performance from such systems, because performance of each processor is different. In addition, cluster systems which are shared by various users also need load balancing, because processor performance may be affected by other jobs. However, dynamic load balancing of numerical programs was difficult on distributed memory systems because it required complex programming with an MPI library. To solve this problem, we implemented a system by extending HPF (High Performance Fortran) with which dynamic load balancing can be easily realized. This system estimates performance of each processor by measuring execution time. Using estimated performance, amount of data assigned to each processor is automatically changed in order to balance the loads. We implemented this system, and evaluated it on a heterogeneous system which is composed of an SX-4 and a Cenju-4, a heterogeneous system which is composed of a PC cluster and a Cenju-4, and a PC cluster which is shared with another job. The evaluation results show that this system provides quite high performance improvement.

1. はじめに

近年，複雑化した数値シミュレーションを高速に実行するシステムとして，異なった種類の並列計算機を結合した異機種並列分散システムが注目されている．しかし，このようなシステムでは各計算機の速度が異なるため，適切な負荷分散を行わなければ高性能は実現できない．また，複数のユーザで共用される計算機でクラスタを構成した場合にも，ほかのジョブの影響でプロセッサごとの性能が変わる可能性があるため，負荷分散が重要となる．

従来，分散メモリシステムで数値計算プログラムの動的負荷分散を行うためには，通常 MPI ライブラリ

† 新情報処理開発機構並列分散システム NEC 研究室
Parallel and Distributed Systems NEC Laboratory, RWCP
☆ 現在，NEC インターネットシステム研究所
Presently with NEC Internet Systems Research Laboratories
☆☆ 現在，地球シミュレータセンター
Presently with Earth Simulator Center

を用い，アプリケーションレベルで負荷分散を記述する必要があった．しかし，このような記述は困難であり，プログラマの負担となっていた．

この問題を解決するため，我々はデータ並列言語である HPF（High Performance Fortran）を拡張し，動的に負荷分散を行うシステムを実現した．HPF を用いることで，プログラミングの容易性を保ちながら，動的負荷分散を行うことができる．

本システムでは，ユーザは GEN_BLOCK 分散を用いて配列を分散する☆．そして，この分散幅をシステムが自動的に変更することで，動的負荷分散を実現する．これにより，HPF の仕様を自然に拡張した形で動的負荷分散を実現できた．

システムは，分散幅を決定するために，各 PE の性能を推定する．これは，繰り返し実行される計算において，その実行時間をそれぞれのプロセッサで計測することで行う．ただし，計算性能を推定する必要があるため，システムが通信時間を別に計測し，実行時間から取り除く．

ここで，実行時間を計測する範囲および負荷分散のタイミングは指示文によりユーザが指定することとした．これにより，柔軟かつ容易な指定が可能になった．

以上に述べたシステムを実装し，SX-4，Cenju-4 から構成される異機種システム，PC クラスタ，Cenju-4 から構成される異機種システム，および負荷をかけた PC クラスタ上で評価を行った．評価により，本システムによって高い速度向上が得られることを確認した．

本論文の構成は以下のとおりである．まず 2 章で本研究の負荷分散手法の概要について述べる．次に 3 章で本研究で構築した動的負荷分散システムについて説明する．次に 4 章で評価について述べた後，5 章で関連研究との比較を行い，最後に 6 章でまとめと今後の課題について述べる．

2. 負荷分散手法の概要

本負荷分散手法では，ユーザは GEN_BLOCK 分散を用いて配列を分散し，システムがその分散幅を自動的に変更することで，負荷分散を実現する．

本章では，まず GEN_BLOCK 分散について説明したあと，システムの動作を説明する．

GEN_BLOCK 分散とは，BLOCK 分散を一般化し，任意の幅での BLOCK 分散を実現したものである．たとえば，

☆ GEN_BLOCK 分散は HPF2.0[4] の公認拡張で規定されている．

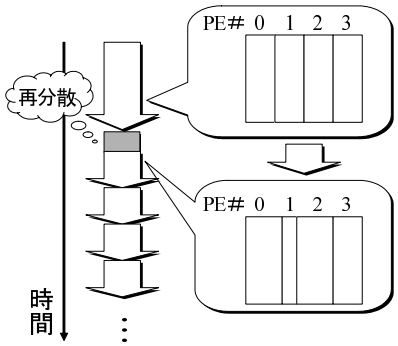

図 1 GEN_BLOCK 分散の再分散による負荷分散
Fig. 1 Load balancing by redistribution of GEN_BLOCK distribution.

```
      real a(300)
      integer map(4)
      data map/100,100,50,50/
!hpf$ distribute a(gen_block(map))
```

のように記述する．gen_block(map) と記述することで，マッピング配列 map 内の値を分散幅として分散することを指示している．この場合，大きさ 300 の配列を，4 台のプロセッサに 100，100，50，50 の幅で分散することを指定している．

HPF では “Owner Computes Rule” を用いて計算を割り当てるため，配列を保持するプロセッサが計算を行う．したがって，このように配列を分散することで，PE#0，PE#1 では 100 要素分の計算を，PE#2，PE#3 では 50 要素分の計算を行うことになる．このように，GEN_BLOCK 分散を用いることで，手動で負荷分散を記述することができる．

しかし，プログラマが手動で GEN_BLOCK 分散の分散幅を適切に決定するのは難しい．計算機の速度比は，プログラムやプログラムの中のループごとに異なる可能性があるためである．また，実行時にプロセッサごとの速度比が変化する場合にも対応できない．

そこで，本論文で示すシステムでは，ユーザが GEN_BLOCK 分散した配列の分散幅を，実行時に測定した各プロセッサの計算速度を元に増減する．すなわち，高速なプロセッサには多く，低速なプロセッサには少なく配列を割り当てるよう，システムが配列の再分散を行うことで，動的負荷分散を実現する．

この様子を**図 1** に示す．この図は，時間発展ループにおいて，同じ計算が繰り返し実行されている様子を表す（矢印 1 つが 1 回分の実行を表す）．この図の場合，最初同じ幅で分散されていた 2 次元配列が，PE#1 が低速であるため，2 回目の繰返しの前に再分散されている．この再分散により，その後の 1 回の繰返しの実

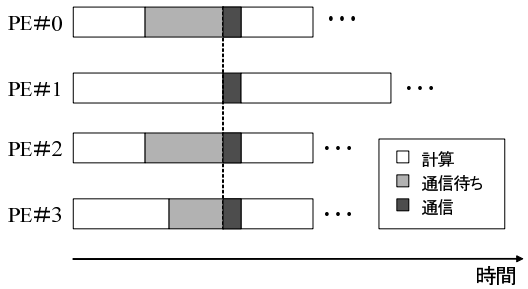

図 2 通信時間と計算時間
Fig. 2 Communication time and calculation time.

行時間が短くなっている．

システムは，分散幅を決定するために，各 PE の性能を推定する．このため，まずユーザが指示文で指定した範囲の実行時間を計測し，計測した実行時間から通信時間を取り除く．ここで，通信時間には，同期のための待ち時間も含まれる．

この様子を**図 2** に示す．白色の部分が計算にかかっている時間，薄い灰色の部分が通信のための待ち時間，濃い灰色の部分が通信時間を示す．この図の場合，最も低速な PE#1 が通信ルーチンにたどり着くまで実際の通信は行われず，ほかの PE は点線の部分まで待たされている．したがって，実行時間から，待ち時間および通信時間の両方を取り除くことで，実際の計算にかかった時間が取り出せる．ただし，待ち時間と通信時間はともに通信ルーチンの中で発生するため，これらを取り除くためには単純に通信ルーチンにかかった時間を取り除けばよい．

システムは各プロセッサに割り当てた分散幅を実行時間で割ることで 1 要素あたりの計算速度を推定する（プログラムは分散幅と計算量が比例するものと仮定する）．この計算速度に比例配分する形で新たな分散幅を計算し，再分散を行う．

ここで，再分散後の全体の計算時間を推定し，速度向上率が一定以上でないと再分散は行わないものとする．これは再分散のための通信コストの方が速度向上分よりも大きい可能性があるためである．

以下の章より，本システムの具体的な実装について詳細に説明する．

3. 動的負荷分散システム

本システムでは，動的負荷分散の指定には，指示文を用いる．本章では，まず指示文の仕様を述べた後，システムの実装について詳細に述べる．

3.1 指 示 文

指示文は，

- ユーザが制御できる余地を残し，柔軟性を高める，
- 処理系に対する負担をできるだけ避ける，

という方針で設計した．この方針に従って，特殊な分散形式を導入するのではなく，GEN_BLOCK 分散をそのまま利用することとした．また，実行時間の測定と再分散のタイミングは，指示文で明示的に指定することとした．

指示文の例を以下に示す．

```
      real a(1000)
      integer map(10)
      data map/10*100/
!hpf$ distribute a(gen_block(map))
      do iter=1,100
!dlb$ calc_load(1) begin
        do i = 1, 1000
          a(i) = i
        enddo
!dlb$ end calc_load(1)
!dlb$ balance_load(1,a,map)
      enddo
      end
```

`!dlb$`で始まる行が，本実装で導入した指示文を表す．`!dlb$ calc_load(1) begin` から`!dlb$ end calc_load(1)` までの間で実行時間を計測する．通信にかかった時間は，別に計測しておき，その分を差し引くことで，計算だけにかかった時間を測定する．

ここで，指示文中の引数にある数字（`1`）は，計時，再分散のときに利用される ID である．測定した実行時間は，引数に与えられた ID で区別される．同じ ID で上記指示文が複数回呼ばれた場合は，それらの実行時間を加算する．

次に`!dlb$ balance_load(1,a,map)` で実際に負荷分散を行う．この場合，ID が “1” の実行時間，および割り当てられていた負荷量を元に，各プロセッサの実行速度を計算する．これを元に，負荷分散後の分散幅を計算する．新しい分散幅と元の分散幅からどれだけの速度向上が見込まれるかを計算し，十分な速度向上率が得られる場合に指定された配列（この場合は `a`）を新しい分散に再分散することで，負荷分散を実行する．

また，配列の分散を変更するほか，マッピング配列（この例の場合は `map`）も変更する．これは，プログラムから変更された分散幅の値にアクセスできるようにするためである．

ここで行われる負荷分散は，処理系の内部では GEN_BLOCK から GEN_BLOCK への再分散とし

て扱われる．

上記の例では，1 つの配列だけを用いていたが，複数の配列を用いる場合は，GEN_BLOCK 分散した負荷分散対象配列に対し，整列するように宣言する．たとえば，

```
!hpf$ distribute a(gen_block(map))
!hpf$ align b with a
      ...
      do i = 1, 100
         a(i) = b(i)
      enddo
      ...
```

などのようにする．これにより，負荷分散時に a の再分散が行われた際，整列された配列 b も同時に再分散されることになる．

計時と再分散を分け，ID で区別するようにしたのは，次のような理由からである．たとえば，

```
      do 時間発展ループ
        do
           ... a(i) ...   ! loop1
        enddo
        do
           ... b(i) ...   ! loop2
        enddo
        do
           ... a(i) ...   ! loop3
        enddo
      enddo
```

というループがある場合を考える．ここで，loop2 だけ除き，loop1 と loop3 の実行時間に基づいて配列 a の負荷分散を行いたい場合を考える．この場合，loop1 と loop3 の実行時間を足したうえで，その実行時間を元に再分散を行う必要がある．そのため，計時の終了と再分散を別の指示文に分け，ID で区別することにした．こうすることで，loop1 と loop3 をそれぞれ同じ ID を持つ指示文で囲むことにより，loop1 と loop3 の実行時間を足した値を計測することができる．

また，これらの指示文は，すべて実行文扱いとする．これは，たとえば 1000 回繰り返し実行される時間発展ループに対し，100 回に 1 回測定および再分散を行いたいという要求があるためである．実行文扱いのため，ユーザは IF 文を使い，自由に実行時間の測定および再分散のタイミングを指定できる．

3.2 システムの構成

システムの構成を**図 3** に示す．

本実装では指示文の解釈をプリプロセッサ（図の指示文解釈部）で行うようにした．

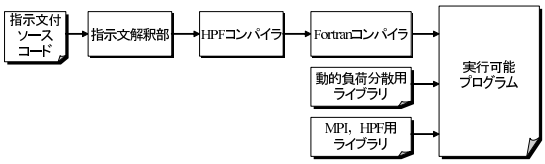

図 3 システム構成
Fig. 3 System organization.

HPF コンパイラは Fortran ソースコードへのトランスレータとして実現されている．この HPF コンパイラは我々が RWC プロジェクトで開発中のものであり，GEN_BLOCK 分散を解釈できるように拡張されている．本実装では，通信時間の計測ができるよう，HPF コンパイラが出力する Fortran ソースコードに通信時間計時用ルーチンを付加できるように改造した．

Fortran ソースコードはコンパイル後，動的負荷分散を行うためのライブラリ，および HPF，MPI 用ライブラリとリンクされ，実行可能ファイルが生成される．

以下の節から，指示文解釈部と HPF コンパイラ，および動的負荷分散用ライブラリの実装について述べる．

3.3 指示文解釈部

前節までで説明した指示文は，コンパイラのプリプロセッサである指示文解釈部によって解釈される．解釈しなければならない指示文は

- `!dlb$ calc_load(ID) begin`
- `!dlb$ end calc_load(ID)`
- `!dlb$ balance_load(ID,ARRAY,MAPARRAY)`

の 3 種類である．これらの指示文を指示文解釈部が見つけると，それぞれをライブラリルーチンへの呼び出しに変換する．この変換は単純に，指示文ごとに 1 対 1 に行う．すなわち，`!dlb$ calc_load(ID) begin` は`call calc_load(ID)` に，`!dlb$ end calc_load(ID)` は`call end_calc_load(ID)` のように変換する．HPF コンパイラによって Fortran ソースコードにコンパイルされる際，システムでは通常のサブルーチンコールと同様に扱われる．

3.4 HPF コンパイラ

HPF コンパイラは GEN_BLOCK 分散を解釈できるよう，拡張したものを利用した．その実装の詳細および評価は別稿に譲り，ここでは概略だけを説明する．

まず，HPF のランタイムシステムでは実行時に配列の分散情報を管理している．これは，分散の種類やパラメータなどをテーブルに記録しておくもので，通信を行う際などに参照される．GEN_BLOCK 分散である場合は，分散情報として，分散幅の情報も記録される．

HPF コンパイラでは “Owner Computes Rule” に従い，配列を保持するプロセッサが計算を行うよう，並列化を行う（異なる分散の配列が同一ループ中で用いられている場合は，ループに入る前に同一の分散になるよう通信が行われる）．配列が GEN_BLOCK 分散されている場合は，実行時に前述の分散幅情報をテーブルから取得し，プロセッサごとの実行範囲を計算する．各プロセッサはこの実行範囲を計算することになる．

通常の BLOCK 分散と比較すると，実行範囲の計算が異なる以外はほぼ同様の Fortran ソースが出力される．したがって，GEN_BLOCK 分散と BLOCK 分散との実行速度の差はほとんどない．

また，動的負荷分散を用いるモードで HPF コンパイラが起動されると，通信用に挿入されるライブラリルーチンの前後に，通信時間を計時するルーチンを挿入するよう拡張を行った．たとえば，

```
call calc_comm
call （通信ルーチン）
call end_calc_comm
```

のようになる．この`calc_comm`および`end_calc_comm`で通信時間の計時を行う．この呼び出しは，コンパイラの通信生成部において，すべての通信用ランタイムルーチンの前後に挿入される．

3.5 ライブラリルーチンの実装

3.5.1 `calc_comm`，`end_calc_comm`

`calc_comm`，`end_calc_comm`では，通信用ライブラリルーチン内での経過時間を計測する．ルーチン内では，経過時間を`mpi_wtime`を用いて測定する．

計測した通信時間は単一のグローバル変数に足し込んでいく．すなわち，通信が発生するたびに，本変数値は増加し，累積通信時間を表すことになる．この値を次の項で述べる`calc_load`，`end_calc_load`で利用する．

3.5.2 `calc_load`，`end_calc_load`

`calc_load`，`end_calc_load`は実行時間計時のための指示文が変換されたライブラリルーチンである．それぞれ引数には ID が与えられる．

ルーチン内では，指示文で囲まれた区間の時間を，`mpi_wtime`を用いて測定する．

ここで，前項で述べた累積通信時間を，`calc_load`の実行時に記録しておく．`end_calc_load`の実行時に，そのときの累積通信時間から`calc_load`実行時の累積通信時間を引くことで，`calc_load`，`end_calc_load`間で発生した通信時間が計算できる．これを測定した時間から減算し，実行時間を計算する．

計算した実行時間は，引数に与えられた ID をキーとして区別し，グローバル変数で実現されたテーブルに保存する．同じ ID に対してこれらのルーチンが複数回呼ばれた場合は，それらを加算する．

このようにして保存された実行時間は，次の項で述べる`balance_load`の実行時にゼロに戻す．

3.5.3 `balance_load`

`balance_load`の引数には，ID，配列，マッピング配列が与えられる．`balance_load`では，与えられた引数および，`calc_load`などで計時した値を用い，実際の再分散処理を行う．

再分散処理では，特定の PE（たとえば PE#0）がマスタになって分散幅などを決定する．まず，PE#0 が各プロセッサの実行時間を収集する．割当て要素数を実行時間で割ったものを速度として扱い，分散対象配列の要素数を速度で比例配分する．

たとえば，PE#0，#1，#2，#3 の（通信時間を除いた）実行時間が 10，40，40，40（秒）で，それぞれ 250，250，250，250 個の要素が割り当てられているとする．ここで，実行時間は`calc_load`で作成したテーブルから該当 ID のものを取得する．また，割り当てられている要素数は，システムが提供する分散情報から取得する．この例の場合，速度は，割り当てられた要素数を実行時間で割ることで得られ，25，6.25，6.25，6.25 になる．

全要素数は 250 + 250 + 250 + 250 = 1000 個であり，これを速度比で比例配分する．PE#0 は 1000 * (25 / (25 + 6.25 + 6.25 + 6.25)) = 571.4 で，PE#0 には 572 個の要素が割り当てられる（block 分散の場合と同様，切り上げ除算を行うことにする）．

ほかのプロセッサも順に同様に要素数を決定する．ただし，割当て要素数が 0 にならないよう，最低 1 要素は割り当てるようにした．これは，以下の理由からである．割当て要素数を実行時間で割ったものを速度として扱うため，割当て要素数が 0 の場合，速度は 0 となる．再分散では要素は速度比で比例配分されるため，速度が 0 のプロセッサには要素は割り当てられない．したがって，1 度割当て要素数が 0 になったプロセッサには，たとえ性能が向上しても，2 度と要素が割り当てられないことになる．これを避けるため，最低 1 要素は割り当てるようにした☆．

また，割当ては PE#0 から行い，割り当てる要素

☆ 再分散が 1 度だけならば，上記の問題は考慮する必要はない．したがって，再分散が 1 度だけの場合を区別して，0 個の割当て要素数を許すという実装も考えられるが，本実装では実装を単純にするため，その区別は行わなかった．

がなくなったところで終了する．

これにより，最終的には PE#0，#1，#2，#3 の割当て要素数は，572，143，143，142 となる．

次に再分散処理を行う．ここで，実際に再分散を行っても通信によるオーバヘッドが負荷分散による速度向上を上回ってしまう可能性がある．そこで，現在の実装では，以下のようなアルゴリズムで負荷分散を実際に行うかどうかを判断している．

まず，負荷分散による速度向上率を見積もる．ここで，それぞれのプロセッサに対して，どれだけ実行時間がかかるかを計算する．上で計算した速度は 1 秒あたりに処理した要素数に相当するので，割り当てられた要素数を速度で割ればよい．この場合，それぞれ $572/25 = 22.88$，$143/6.25 = 22.88$，$143/6.25 = 22.88$，$142/6.25 = 22.72$ となる．これらのうちの最大値が予測実行時間となる．この場合 22.88 秒かかることになる．

したがって，予測される実行時間の向上率は，測定した実行時間の最大値（40 秒）を予測実行時間（22.88 秒）で割ったものであり，この場合は $40/22.88 = 1.75$ 倍の速度向上が見込める．

現在は，このように速度向上率を計算し，その速度向上率が一定以上（10%）の場合だけ，実際に負荷分散を行う実装になっている．

本手法は通常の運用ではうまく動作するが，本来は，通信量と通信スループットから通信時間を推定し，推定される実行速度向上分と比較することで，負荷分散を行うかどうかを判定すべきである．ここで，通信量は再分散にともなう割当て要素数の変更から計算可能である．通信スループットはあらかじめ測定した値を与える，あるいは過去の通信履歴から推定することなどで，得ることが可能であると考えられる．

しかし，異機種システムの場合は通信スループットが同一機種内と異機種間で異なることや，HPF 処理系による通信パターンと同じ通信パターンで通信時間を見積もる必要があることなどから，実装が複雑になる．また，以上を考慮しないような単純な実装では，見積りを大幅に誤ってしまう可能性がある．このため，今回はこのような手法を採用しなかった．このような手法の採用は今後の課題である．

4. 評　　価

以上に述べた負荷分散システムを実装し，評価を行った．実装，評価を行ったシステムは，以下のとおりである：

- SX-4[6]，Cenju-4[3]から構成される異機種システム
- PC クラスタ，Cenju-4 から構成される異機種システム
- PC クラスタの一部の PE に負荷をかけたシステム

異機種システムでは，我々が実現した異機種システム用 MPI ライブラリを利用した．本ライブラリは異機種システムをフラットに結合する．すなわち，たとえば SX-4 1 台，Cenju-4 8 台からなる異機種システムでは，rank 0 の MPI プロセスが SX-4 で動作し，rank 1 から rank 8 までの MPI プロセスが Cenju-4 のそれぞれの PE で動作する．HPF コンパイラは MPI を利用する Fortran プログラムを出力するため，この MPI ライブラリを利用することで，シームレスに異機種システムを利用することができる．

この MPI ライブラリでは，システム内部の通信に，それぞれのシステムで提供されている MPI ルーチンを利用する．システム内部の通信には，SX-4，Cenju-4 ではシステムが提供する MPI ルーチンを用い，PC クラスタでは MPICH を用いた．

また，PC クラスタ単体での評価では，MPI ライブラリとして MPICH を用いた．

以下にそれぞれのシステムでの評価を示す．

4.1 SX-4，Cenju-4 による異機種システム

SX-4 は共有メモリ型ベクトルスーパコンピュータであり，プロセッサは独自のものを用いている．単一プロセッサでのピーク性能は 2 GFlops である．Cenju-4 は分散メモリ型の並列計算機であり，プロセッサは 200 MHz の VR10000 である．プロセッサ間の通信速度はピークで 200 MByte/sec（双方向）である．SX-4 と Cenju-4 は 100 Base-T の LAN で結合されている．

評価に用いたプロセッサ台数は，SX-4 は 1 台，Cenju-4 は 8 台である．

評価には APR Benchmark Suite に収められたベンチマークプログラムの 1 つである grid を用いた．データサイズは 2002×2002 とし，繰返し回数を 500 とした☆．時間発展ループの最初と最後に `!dlb$ calc_load begin`，`!dlb$ end calc_load` を挿入し，`!dlb$ end calc_load` の直下に `!dlb$ balance_load` を挿入した．

表 1 に実行時間（秒）を，**図 4** に Cenju-4 単体で実行したときの速度を 1 とした速度比を示す．

☆ 本プログラムでは，データサイズの一辺の大きさを n としたときに，計算量は $O(n^2)$，通信量は $O(n)$ となる．したがって，データサイズが大きいほど，通信時間の全体に占めるコストが小さくなる．本データサイズは通信コストがあまり大きくなりすぎない程度に設定した．ただし，データサイズを大きくすると，必要なデータ転送量が増えるため，再分散のコストは大きくなる．繰返し回数は単純に全体の実行時間に影響する．

表 1 SX-4，Cenju-4 による異機種システムでの実行時間（秒）
Table 1 Execution time on a heterogeneous system composed of an SX-4 and a Cenju-4 (sec).

SX-4 単体	Cenju-4 単体	SX-4+Cenju-4 等分散	SX-4+Cenju-4 動的分散	SX-4+Cenju-4 手動分散	再分散
381	717	625	319	272	42.6

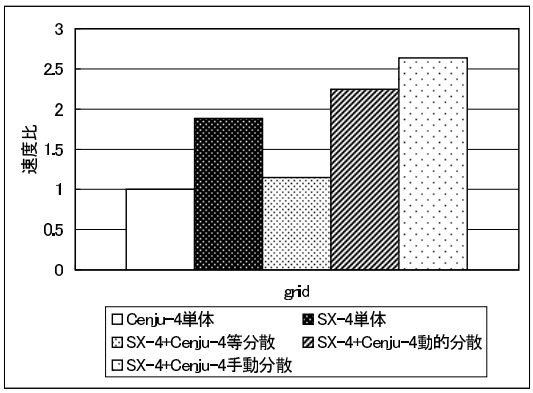

図 4 SX-4，Cenju-4 による異機種システムでの速度比
Fig. 4 Speed ratio on a heterogeneous system composed of an SX-4 and a Cenju-4.

ここで，SX-4 単体は SX-4 1 台での実行時間を，Cenju-4 単体は Cenju-4 8 台での実行時間を示す．また，SX-4+Cenju-4 等分散は，SX-4 と Cenju-4 で（GEN_BLOCK を用いて）同じ分散幅で分散を行った場合を，SX-4+Cenju-4 動的分散は最初は同じ分散幅で分散し，動的負荷分散を利用して負荷分散を行った場合の実行時間を示す．

グラフから，等分散の場合に比べると，動的負荷分散を行った場合の方が，約 2 倍高速であったことが分かる．さらに，単体で実行した場合よりも，異機種システムで動的負荷分散を用いた場合の方が高速であることが分かる．

等分散の場合は，SX-4 単体で実行した場合よりも低速になっている．これは，SX-4 と Cenju-4 の 1 PE あたりの性能差が大きいため，等分散では SX-4 に割り当てられた要素数が少なく，SX-4 の待ち時間が多くなってしまうためである．

本評価では，動的負荷分散によって，1 回だけ再分散が行われた．初期分散幅は PE#0 から PE#7 までが 223，PE#8 が 218 であったが（これ以降，8*223, 218 などと表記する），再分散により，1282，91，90，91，91，91，91，91，84 に変更された（SX-4 の方が PE#0 になる）．再分散時には，SX-4 と Cenju-4 の 1 PE あたりの速度比は（代表的な値を用いると）1282/91 = 約 14.1 : 1 として扱われている．SX-4，Cenju-4 単体での実行速度から速度比を計算すると，$717 * 8/381$ = 約 15.1 : 1 となる．これを見ると，単体の実行速度から計算した速度比と，動的負荷分散で利用された速度比はほぼ同じであり，再分散により適切に速度比が設定されていることが確認できる．若干の差は，Cenju-4 単体での実行時間に通信時間が含まれてしまっていること，また，異機種での実行と単体での実行で 1 PE あたりで使用するメモリ量が異なることなどに起因すると考えられる．

また，初期分散を再分散後の分散幅に指定し，動的負荷分散機構を用いずに評価を行ったのが，SX-4+Cenju-4 手動分散の項である．動的分散に比べると高速であるが，これは，主に再分散にともなう通信が影響しているものと考えられる．

上記の再分散を行うプログラムを HPF で記述し，実行時間を計測したものが表の再分散の項である．再分散時間と SX-4+Cenju-4 手動分散の時間を足すと，ほぼ SX-4+Cenju-4 動的分散の時間になることから，再分散にともなう通信がオーバヘッドの多くの部分を占めることが確認できる．

4.2 PC クラスタ，Cenju-4 による異機種システム

本節では PC クラスタ，Cenju-4 による異機種システムでの評価を示す．ここで用いた PC クラスタは，Pentium III 866 MHz の CPU を持ち，Gigabit Ethernet で接続されている．OS は Red Hat Linux 6.2 である．PC クラスタと Cenju-4 は 100 Base-T の LAN で結合されている．PC クラスタ 4 台，Cenju-4 4 台の構成で評価を行った．

評価に用いたプログラムは，前述の grid に加え，SPEC ベンチマークの tomcatv，NCAR ベンチマークの shallow を用いた．

本評価では，grid のデータサイズは 2002×2002，繰返し数は 100 とした．指示文の挿入部分は先ほどの評価と同じである．

また，tomcatv ではデータサイズを 1536×1536，繰返し数を 300 とした．実行時間の計測は，主に実行されるループに関してのみ行うよう指定した．`!dlb$ balance_load` は時間発展ループの最後に挿入し，繰返しの 5 回に 1 回のみ実行するよう，IF 文を挿入した．

shallow ではデータサイズを 2000×2000，繰返し数を 300 とした．指示文は時間発展ループの最初と最後に挿入し，`!dlb$ balance_load` は同様に繰返しの

表 2 PC クラスタ，Cenju-4 による異機種システムでの実行時間（秒）
Table 2 Execution time on a heterogeneous system composed of a PC cluster and a Cenju-4 (sec).

プログラム	PC 単体	Cenju-4 単体	PC+Cenju-4 等分散	PC+Cenju-4 動的分散	PC+Cenju-4 手動分散	再分散
grid	155	297	143	108	103	1.6
tomcatv	178	372	195	140	132	8.5
shallow	185	290	163	145	136	9.8

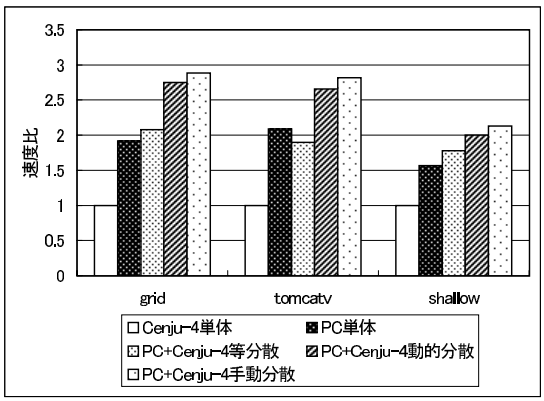

図 5 PC クラスタ，Cenju-4 による異機種システムでの速度比
Fig. 5 Speed ratio on a heterogeneous system composed of a PC cluster and a Cenju-4.

5 回に 1 回のみ実行するようにした☆．

表 2 に実行時間を，**図 5** に Cenju-4 単体で実行したときの速度を 1 とした速度比を示す．

図，表の意味は先ほどの評価と同様である．ただし，「PC」は，PC クラスタ 4 台を，「Cenju-4」は Cenju-4 4 台を表す．

本評価においても，前節での評価と同様の傾向を示す．すなわち，いずれの場合も等分散の場合に比べると，動的負荷分散を行った場合の方が高速である．さらに，単体で実行した場合よりも，異機種システムで動的負荷分散を用いた場合の方が高速である．

grid や shallow では等分散の性能が PC クラスタ単体の性能を上回っているが，tomcatv では，前節での評価と同様，等分散の性能が PC クラスタ単体の性能を下回っている．これは，tomcatv の場合，PC クラスタと Cenju-4 の 1 PE あたりの性能差が grid や shallow より大きく，2 倍以上となっているためである．

PC クラスタと Cenju-4 の台数は同じであるため，PC クラスタと Cenju-4 の両方を用いた場合は，Cenju-4 に全体の半分の仕事が与えられる．ここで，Cenju-4 の性能が PC クラスタの半分以下である場合，その半分の仕事を実行するのにかかる時間は PC クラスタ単体で全体の仕事を実行するのにかかる時間よりも大きくなる．このため，tomcatv のみ等分散の性能が PC クラスタ単体の性能を下回っている．

手動分散は動的分散よりも若干高速であるが，通信のオーバヘッドが小さかったため，性能差は小さい．

ここで，再分散時間は SX-4 と Cenju-4 で構成されたシステムよりも小さなものになっているが，これはプロセッサあたりの速度比が PC クラスタと Cenju-4 で構成されたシステムの方が小さいため，再分散における通信量が小さいこと，また，システムの違いにより，異機種間の通信スループットが PC クラスタと Cenju-4 で構成されたシステムの方が高かったことが原因だと考えられる．

また，プログラムによって再分散時間が異なるが，これは速度比が異なることのほか，再分散を行った配列の数が異なるため，再分散における通信量が異なることによる．tomcatv および shallow では，3.1 節で述べた形で，複数の配列を `!hpf$ align` 文により整列したうえで利用している．負荷分散時にはこれらのすべての配列が再分散されるため，配列の数により再分散に必要な通信量が大きく異なることになる．

等分割の場合と動的負荷分散を行った場合の速度比は，grid で約 1.3 倍，tomcatv で約 1.4 倍，shallow で約 1.12 倍であった．

grid の初期分散幅は 7*251，245 であり，これが再分散により 305，327，327，328，180，179，179，177 に変更された（PC クラスタ側が PE#0 になる）．tomcatv の初期分散幅は 8*192 であったが，再分散により 270，270，269，260，117，117，118，115 に変更された．また，shallow の初期分散幅は 250*8 であったが，再分散により 295，309，309，304，195，197，198，193 に変更された．いずれの場合も再分散は 1 回だけ行われた．

次に PC クラスタと Cenju-4 の 1 PE あたりの速度比について，単体での実行速度から計算した速度比と，

☆ tomcatv，shallow においても，grid と同様，問題サイズの一辺の大きさを n としたときに，計算量は $O(n^2)$，通信量は $O(n)$ となる．データサイズについては grid の場合と同様，通信コストがあまり大きくなりすぎない程度に設定し，繰返し回数は，それぞれのプログラムの実行時間が同じ程度になるよう調整した．

表 3 負荷をかけた PC クラスタでの実行時間（秒）
Table 3 Execution time on a PC cluster shared with another job (sec).

プログラム	負荷なし	継続負荷		断続負荷	
		負荷分散 OFF	負荷分散 ON	負荷分散 OFF	負荷分散 ON
grid	155	308	185	206	180
tomcatv	178	337	226	233	219
shallow	185	360	233	245	227

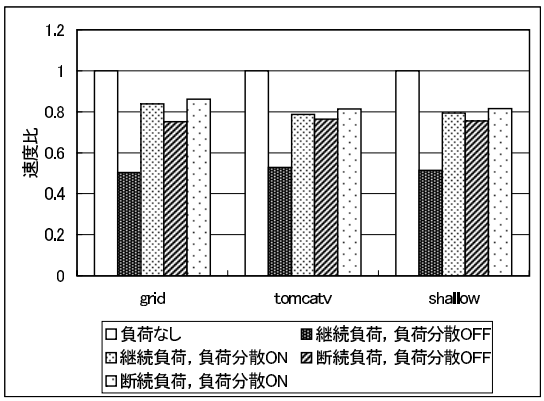

図 6 負荷をかけた PC クラスタでの速度比
Fig. 6 Speed ratio on a PC cluster shared with another job.

再分散時に仮定された速度比を比較する．再分散時に仮定された速度比の計算には，代表的な分散幅を用いる．grid では，実行速度からは $(297*4)/(155*4) = 1.92 : 1$，再分散時は $327/179 = 1.83 : 1$ であった．また，tomcatv では，実行速度からは $(372*4)/(178*4) = 2.09 : 1$，再分散時は $270/117 = 2.31 : 1$ であった．また，shallow では実行速度からは $(290*4)/(185*4) = 1.57 : 1$，再分散時は $309/197 = 1.57 : 1$ であった．いずれの場合も，再分散により適切に速度比が設定されていることが確認できる．

4.3 負荷をかけた PC クラスタ

本節では，PC クラスタ単体を用い，PE の 1 つに負荷を与えた場合の評価を行った．これは，複数のユーザで共用される計算機でクラスタを構成した場合に，ほかのジョブの影響でプロセッサごとの性能が変わる場合を想定した評価である．

評価に用いたプログラムは前節の評価と同じである．また，PC クラスタも前節の評価のものと同じである．**表 3** に実行時間を，**図 6** に負荷がない場合の速度を 1 とした速度比を示す．

ここで，継続負荷とあるものは，1 PE に継続的に負荷を与えたもの，断続負荷とあるものは，20 秒負荷を与え，20 秒負荷をなくすという周期を繰り返したものである．ここで，負荷は無限ループを実行するプログラムを別プロセスで起動することで与えた．断続負荷の場合の方が，再分散が多く必要なため，動的負荷分散システムにとっては厳しい環境となる．

まず継続負荷の場合であるが，動的負荷分散を行わない場合は，負荷がかかっている PE の速度が約半分となることで律速となり，いずれのプログラムでもほぼ半分まで速度が低下している．これに対し，動的負荷分散を行った場合は，負荷を移動することで速度低下を抑えている．動的負荷分散を行わない場合に比較して，grid で約 1.7 倍，tomcatv で約 1.5 倍，shallow で約 1.5 倍高速となった．ここで，grid の初期分散は 3*501, 499 であり，これが再分散により 269，578，578，577 に変更された（PE#0 に負荷を与えた）．tomcatv の初期分散は 4*384 であり，これが 253，436，430，417 に変更された．また，shallow の初期分散は 4*500 であり，これが 313，564，566，557 に変更された．

また，断続負荷の場合でも動的負荷分散を行った場合の方が高速である．grid，tomcatv，shallow いずれの場合も 13 回再分散が行われたが，grid で 1.14 倍，tomcatv で 1.06 倍，shallow で 1.08 倍高速となった．

5. 関 連 研 究

負荷分散は並列処理においては基本的な問題であり，古くから多くの研究がなされている．ここでは，分散メモリ計算機において，数値計算プログラムを対象とする関連研究との比較を行う．

文献 1) では，逐次プログラムをループやサブルーチンなどのマクロタスクに分割し，それらを自動的に負荷分散，スケジューリングする．文献 8) ではこの手法を用い，異機種並列環境で負荷分散，スケジューリングを行っている．我々の手法がプロセッサに割り当てるデータ量を変更することで負荷分散を実現しているのに対し，本手法はマクロタスクを単位として負荷分散を実現しているという点で異なっている．

文献 5) で示された手法では，仮想的に多くのプロセッサがあるものとしたうえで，逐次プログラムを HPF と同様に並列化する．仮想プロセッサを実プロセッサにマッピングする際に，負荷や速度に応じて割り当てる仮想プロセッサの数を変えることで，動的な

負荷分散を行っている．この手法では，コンパイラの並列化部分が単純になるなどのメリットはあるが，配列ごとやループごとのような，きめ細かい負荷分散の指定は行うことができない．我々の手法では，指示文で異なるIDを用いることで，プログラムの部分ごとに異なる負荷分散を行うことも可能となる．

アプリケーションレベルで負荷分散を行う研究も多く行われている．たとえば文献7)では，MPIプログラムレベルでプログラマが明示的に記述する方法での負荷分散が提案されている．本提案手法は我々の手法と類似しており，繰返しごとに各プロセッサの実行時間を計測し，それに基づいて各プロセッサの計算範囲を増減することで負荷分散を行う．しかし，アプリケーションプログラマがこのような負荷分散を明示的に記述するのは，プログラマに対する負担が大きいと考えられる．

文献2)では，有限要素法を用いるプログラムに対し，負荷分散を行うライブラリを提供している．ライブラリ化することで適用範囲を広げてはいるが，有限要素法以外のプログラムには適用できない．

6. おわりに

本研究では，HPFを拡張し，動的に負荷分散を行うシステムを実現した．本システムをSX-4，Cenju-4から構成される異機種システム，PCクラスタ，Cenju-4から構成される異機種システム，およびPCクラスタ上に実装，評価を行い，高い速度向上が得られることを確認した．

今後の課題として，有限要素法のような，均一な環境においても負荷分散が必要なアプリケーションに対し，本手法を適用していくことがあげられる．

また，より大規模なアプリケーションに対し本手法を適用し，ループごとに異なる負荷分散が必要な場合の本手法の有効性を確認することも，今後の課題としてあげられる．

参考文献

1) Kasahara, H. and Yoshida, A.: A data-localization compilation scheme using partial-static task assignment for Fortran coarse-grain parallel processing, *Parallel Computing*, Vol.24, No.3–4, pp.579–596 (1998).
2) Maerten, B., Roose, D., Basermann, A., Fingberg, J. and Lonsdale, G.: DRAMA: A Library for Parallel Dynamic Load Balancing of Finite Element Applications, *Euro-Par'99*, LNCS 1685, pp.313–316 (1999).
3) Nakata, T., Kanoh, Y., Tatsukawa, K., Yanagida, S., Nishi, N. and Takayama, H.: Architecture and the Software Environment of Parallel Computer Cenju-4, *NEC RESEARCH & DEVELOPMENT*, Vol.39, No.4, pp.385–390 (1998).
4) 財団法人高度情報科学技術研究機構：High Performance Fortran 2.0公式マニュアル，シュプリンガー・フェアラーク東京 (1999).
5) 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による非均質計算環境向けコード生成手法，並列処理シンポジウムJSPP'97，pp.205–212 (1997).
6) 井上政信，大和田刻明，古井利幸，片桐 勝：SX-4シリーズのハードウェア構成，NEC技報，Vol.48, No.11, pp.13–22 (1995).
7) 熊谷泰幸，木下浩三，岸 達也，佐々木隆，伊藤 聡：自動負荷分散の実装とその評価，並列処理シンポジウムJSPP2001，pp.75–76 (2001).
8) 小出 洋，笠原博徳：メタスケジューリング—自動並列分散処理の試み，*bit*, Vol.33, No.4, pp.36–41 (2001).

(平成14年1月29日受付)
(平成14年5月16日採録)

荒木 拓也（正会員）

1971年生．1994年東京大学工学部電気工学科卒業．1999年同大学大学院情報工学専攻博士課程修了．博士（工学）．同年，NEC入社．現在インターネットシステム研究所勤務．プログラミング言語の実装，特に最適化，並列化等に興味を持つ．

村井 均（正会員）

1971年生．1996年京都大学大学院工学研究科情報工学専攻修士課程修了．同年NEC入社．現在，地球シミュレータセンター技術員．並列スーパーコンピュータ向けの言語環境の研究開発に従事．

蒲池　恒彦（正会員）

1964 年生．1988 年九州大学工学部情報工学科卒業．1990 年同大学大学院総合理工学研究課情報システム学専攻修了．同年 NEC 入社．現在，インターネットシステム研究所主任．並列計算機アーキテクチャ，自動並列化コンパイラ，並列化支援システム等に興味を持つ．

妹尾　義樹（正会員）

1961 年生．1984 年京都大学工学部情報工学科卒業．1986 年同大学大学院修士課程修了．同年 NEC 入社．インターネットシステム研究所にてスーパーコンピュータの研究開発に従事．並列処理アーキテクチャ，分散メモリマシンのための並列化言語環境，並列アルゴリズムに興味を持つ．現在インターネットシステム研究所研究部長．工学博士．1988 年本会論文賞受賞．