自律コンピューティングに向けたHPC向け動的負荷分散機構

松原正純† 鈴木和宏† 勝野昭†

IT インフラの拡大によるリソースの運用・管理コストの爆発的な増大に対処すべく，近年，「自律コンピューティング」が注目されている．自律コンピューティングが確立されれば，リソース最適化による効率的なセンター運用や大幅な TCO 削減につながり，そのメリットは計り知れない．しかし，従来の HPC 並列アプリケーションは実行時の並列度が固定されており，自律コンピューティングによるリソース最適化の恩恵にあずかることができない．そこで本研究では，利用可能なノード数に合わせて動的にプロセス数を変えつつ，負荷状況に応じて各ノードに割り当てるデータ量を調整する負荷分散機構を提案する．我々はメッセージパッシングライブラリ LAM/MPI を用いて上記負荷分散機構を実装し，さらに同機構を利用するための専用 API を用意した．この API を用いてプログラミングすることにより，対象アプリケーションにプロセス並列度可変の動的な負荷分散機能を組み込むことができる．複数の PC クラスタを用いて評価した結果，利用可能ノード数および各ノードの性能に従って適切に負荷が分散されることを確認した．

Dynamic Load Balancing in HPC applications for Autonomic Computing

Masazumi Matsubara,† Kazuhiro Suzuki† and Akira Katsuno†

In recent years, “Autonomic Computing” attracts attention as technology which solves the problem of increasing administration cost in IT infrastructure, such as iDC and ASP. The technology can realize efficient administration by optimizing resources, and reduce TCO dramatically. Existing HPC applications, however, can not get the benefit of the resouce optimization because the parallelism is fixed. In order to overcome the problem, we propose a new mechanism which adjusts CPU loads, changing the number of processes dynamically according to the number of nodes. We have implemented the dynamic load balancing mechanism using message passing library LAM/MPI, and prepared API for using the mechanism. We evaluated the performance of the mechanism on heterogeneous PC cluster systems, and got the conclusion that the mechanism can balance CPU loads appropriately according to the number of nodes and each node’s performance.

1. はじめに

近年の IT の浸透により，データセンターなどのインフラが今後ますます拡大すると予想されるが，それにともない，リソース運用・管理コストの爆発的な増大が懸念される．そこで，メーカー各社は「自律コンピューティング」をキーワードとして，TCO の削減，安定したサービスの提供，リソースの最適化に取り組んでいる．富士通では「オーガニック・サーバ」を核とした研究・開発を進めている．また，IBM，サン・マイクロシステムズ，ヒューレット・パッカードはそれぞれ，「オートノミック・コンピューティング」，「N1」，「hp adaptive infrastructure」という「自律コンピューティング」のコンセプトを打ち出している．さらに，他の日本企業からも，「VALUMO」（NEC），「Harmonious Computing」（日立）が発表されている．

この自律コンピューティングを実現するための 2 大要素技術として，自律制御技術，仮想化技術があげられる．自律制御技術とは，システムの状態を監視し，負荷に応じた適切な量のリソースを割り当てたり，故障ノードの回収および処理の引継ぎを自動で行ったりする技術である．また，このとき，稼働中のアプリケーションもしくはシステム全体をダウンすることなく運用を継続させるためには，リソースの仮想化技術が必須である．これらの技術を活用することにより，データセンターや ASP ではより効率的に各種サービスを提供することができる．

しかし，現実問題として，アプリケーションに対して適切にリソースを割り当てられるかどうかは，その

† 富士通株式会社
FUJITSU LIMITED

アプリケーションの特性によるところが大きい．たとえば Web サーバの場合，個々のノードで走るサーバプロセスは他のノード上のサーバプロセスとは独立している．したがって負荷が増大した場合には，待機ノードにサーバを起動して，インターネット入口に置かれたロードバランサに対して追加したノードにも HTTP リクエストを振るように指示することで簡単に対応できる．それに対して，従来の HPC 並列アプリケーションは，最初に起動されたプロセス群だけで最後まで計算処理を続ける場合がほとんどである．これは，長時間特定のノード群を占有すること，そして使用可能ノード数の上限が定まっていることを意味する．この制約は同時に稼働中の Web サーバなど他のアプリケーションの負荷分散の妨げとなり，さらに実行途中で利用可能となった遊休ノードをまったく活用できないため非効率である．

そこで本研究では，実行時の環境に合わせて動的にプロセス数を増減して負荷分散を図る負荷分散機構を開発するすることで，上記問題を解決する．この機構により，自律コンピューティングシステムに比較的容易に HPC アプリケーションを適用することが可能となる．

以降では，まず，自律コンピューティングシステムにおける本機構の位置づけを示し，その後，本研究で提案する負荷分散ストラテジおよびそれを実現する実装方法を説明する．最後に本機構の評価を行う．

2. HPC アプリケーションの課題

図 1 が自律コンピューティングシステムの概念図である．ただし，図 1 には仮想化機構は含まれていない．自律コンピューティングシステムでは，リソース提供者がユーザとの間で交わした SLA（Service Level Agreement）もしくは運用ポリシーにしたがって，ソフトウェア・ハードウェアの構成がリソース管理層によって自動的に決定される．システム起動後は適宜リソース・モニタリング結果がリソース管理層にフィードバックされ，その結果に基づきシステムが再構成される．たとえば優先度の高い Web サーバへの負荷が急激に増加した場合，同時に走っている優先度の低い分子シミュレーションアプリケーションを縮退させ，空いたノードに Web サーバを立ち上げるといった処理を自律的に行う．反対に，Web サーバへのアクセスが減少した場合には，SLA を満たすだけのノードを確保しておき，残りのノードは分子シミュレーションに用いるケースもありうる．

ここで，効率的に全リソースを利用するためには，

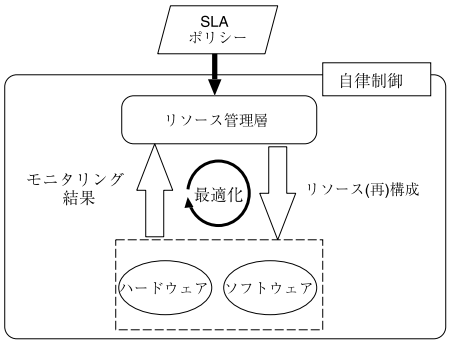

図 1 自律コンピューティング
Fig. 1 Overview of autonomic computing.

分子シミュレーションなどの HPC アプリケーションは，リソース管理層からの指示に従い，適切に使用ノードの追加・削除が行えなくてはならない．縮退に関しては，プロセスもしくはスレッドのノード間マイグレーションにより対応することも可能であるが，伸長の場合は，実行コンテキストがノード台数分なければ割り当てられたノードすべてを活用することはできない．

さらに考慮しなくてはならないもう 1 つの問題として，異機種構成によるノード間の性能格差がある．データセンターなどには様々なマシンが置かれるが，無駄なくそれらリソースを利用するためには異機種構成も念頭に入れなくてはならない．

以上のことから，HPC アプリケーションを自律コンピューティングに対応させるためには，以下の 2 つの条件を満たす動的負荷分散機構が必要である．

(i) リソース管理層からの指示に基づいて使用ノード数，つまりプロセス数を柔軟に変更する．
(ii) ノード間の性能差を考慮した負荷分散を実現する．

なお，本研究では，現時点においては負荷分散機構にのみ焦点を合わせている．したがって，本来の自律コンピューティングシステムならば，モニタリング結果に基づきリソース管理層から自律的にノード数調整の指示が出るはずところを，今回は人間が遊休ノードを判断し，手動で指示を与えるようにする．

3. 負荷分散ストラテジ

前章の課題を踏まえ，本研究ではプロセス数の増減機構を新たに加えた負荷分散機構を提案する．本章では提案する負荷分散機構における上記条件の対処方法を示した後，実際どのようにプロセス数増減に対応するのか，具体例を交えて全体の処理の流れを説明する．

まず条件 (i) について，この条件を満たすためには

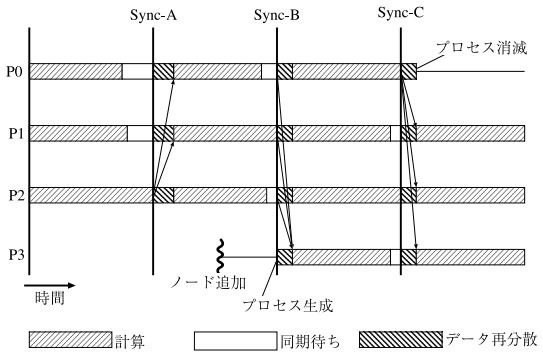

図 2 負荷分散の流れ
Fig. 2 Flow of load balancing.

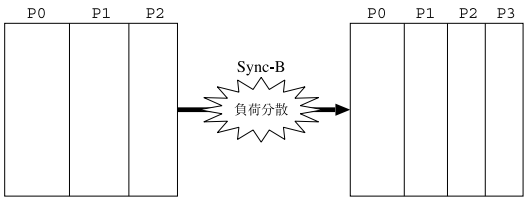

図 3 配列データの分散
Fig. 3 Distribution of array data.

アプリケーション側にリソース管理層からの指示を受け付ける機構が必要となる．ただし，計算途中にノード数の変更を許すと，その後データの整合性を保つのは困難を極める．そこで，ノード数変更のタイミングはアプリケーション側で制御することにする．そのために定期的に同期をとり，その時点でリソース管理層から指示がきていないかチェックするようにする．

条件 (ii) に対しては，文献 7)~9) でも採用されているように，各ノードに割り付けるデータ量を変更することで対応する．具体的には，計算対象となる配列をブロック分割して各プロセスに割り付け，そのブロックサイズを適当に変更することで負荷調整する．よって，本機構で対応できるアプリケーションは領域分割型の並列方式を採用していて，なおかつブロック分割の場合にのみということになる．

以上の対応方法を基に，提案する負荷分散機構の処理の流れを**図 2** を用いて説明する．ユーザプロセス P0-3 は，ノードごとに 1 プロセスずつ割り当てられているものとする．また，P3 が走るノードはアプリケーション実行開始時には利用不可状態であり，途中で追加されるものとする．よって，最初は P0-2 の 3 プロセスで並列実行されることになる．

図 2 では，CPU 性能や他のジョブの影響により，P2 は P0，P1 よりも遅れて第 1 回目の同期ポイント（Sync-A）に到達している．そこで，次の同期ポイントに到達するまでの時間が同じになるように P2 から P0，P1 に対してデータを分散することで負荷調整を図り，そして計算処理を続行する．

また，図では 4 番目のノードが Sync-A と Sync-B の間で追加されている．したがって Sync-B での負荷調整時に，新たにノードが追加されたことを検出し（正確には，リソース管理層から指示があったことを検出する），そのノード上に新たにプロセス P3 を生成してデータを分配する．このときのデータ分散を図示したものが，**図 3** である．図 3 では，2 次元配列データが並列プロセス間で共有されているものとしている．ここで，Sync-A においてすでに負荷調整が為されているため，Sync-B 直前の各プロセスへのデータ割付け量が異なっている点と，Sync-B 後に新規プロセス P3 を含めてデータ分散していることに注意されたい．

さらに，あるノードが非常に高負荷になった場合，もしくは別のアプリケーションに割り当てる必要が生じた場合，リソース管理層からアプリケーションに対して当該ノードを切り離すよう指示されることもある．このとき，負荷分散機構は当該ノード上のユーザプロセスが持つデータを他のユーザプロセスにすべて分散し，その後当該プロセスを終了させる．同期ポイント Sync-C がまさにその例である．

以上のように，同期ポイントにおいてリソース管理層からのノード増減の通知がないかを確認した後に，ノード数に変更があった場合にはそのことも考慮して各ノードの負荷状況に応じて適切にプロセス数の調整およびデータの再分散を実行することで，最適な負荷分散が実現される．

本負荷分散方法を実現するためには同期ポイントや分散対象となるデータの設定をする必要があり，それをどのレベルで実装するか決めなくてはならない．そこで次章では，本機構の実装方法について述べる．

4. 負荷分散機構の実装

4.1 実 装 手 法

本機構を実現するためには，以下の 3 つのアプローチが考えられる．

(1) アプリケーション個別対応
アプリケーションレベル，つまりユーザが書くプログラム中に動的負荷分散のためのコードを埋め込む方法である．この場合，ユーザはアプリケーションの特性およびデータ構造を完全に把握しているため，どのデータをいつ再分散すればよいかということを最も適切に設定するこ

とができる．しかし，裏を返せばそれだけユーザはアプリケーションに精通している必要があり，またプロセス生成・消滅を含めた負荷分散機能をそのつど個々のアプリケーションに埋め込まなければならず，負担はかなり大きい．

(2) 専用ライブラリの提供
動的負荷分散機構をライブラリの形でユーザに提供することで，ユーザへの負担を軽くする．ただしこの場合も，負荷分散対象のデータと同期ポイントに関してあらかじめユーザから指示してもらわなくてはならないため，アプリケーションに対する知識は必要不可欠である．

(3) 並列化コンパイラによる自動組込み
コンパイラによってプログラムを解析し，自動で負荷分散機構を追加しようというものである．ただし，現在のコンパイラ技術をもってしても完全にデータ構造および処理フローを解析することは不可能である．そこで，たとえばOpenMP[1)]やHPF[2)]のように，逐次プログラム中にコンパイラに対する動的負荷分散用の指示文を埋め込むことで解析を助けることを考える．

最終的にはユーザに対する負担が最も小さいコンパイラへの実装が望まれる．ただし今回は実現容易性も考慮し，なるべく簡単に実装・検証ができ，かつ実用にも使えることから，ライブラリレベルでの実装を選択することにした．今回の研究で得られた知見は，今後のコンパイラへの実装へと活かしていく予定である．

4.2 LAM/MPIの利用

PCクラスタなどの分散メモリシステム向けのHPC並列アプリケーションは，一般にMPIやPVMなどの通信ライブラリを用いて作成される．また，前述の負荷分散機構ではノード間でデータ移動が生じるので，これら通信ライブラリと組み合わせるのが都合がよい．そこで今回，MPIの実装の1つであるLAM/MPIを利用して本動的負荷分散機構を追加することにした．今回の動的負荷分散機構ではプロセス数を可変とするため，MPI-2の動的プロセス生成機能が必須であるが，LAM/MPIではこの機能もすでに実装されている．また，LAM/MPI独自のものとして，**表1**に示すノード管理用コマンド群が提供されている．

これらのコマンドを用いると，図2のプロセス数変更は**図4**のようにして実現できる．本機構では，同期ポイントでの負荷調整時にプロセス数の変更が通知された場合は，図4に従って提供するライブラリ内部で適切なコマンドおよびMPI関数を呼び出してプロセス数の調整を行う．

表1 LAM/MPIのノード管理用コマンド
Table 1 LAM/MPI administration commands to control nodes.

コマンド名	機能
lamboot	MPI実行環境の構築
lamgrow	ノードの追加
lamshrink	ノードの削除
wipe	MPI実行環境のターミネート

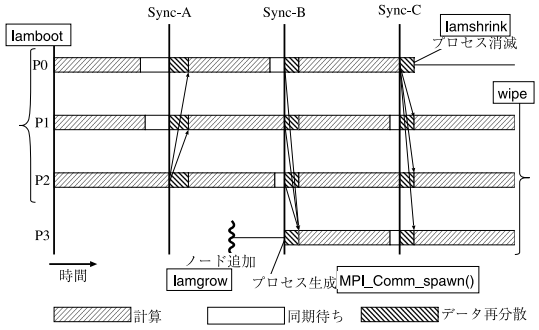

図4 LAM/MPIを用いた実装方法
Fig. 4 Implementation using LAM/MPI.

4.3 負荷情報の取得

本負荷分散機構では各プロセスの内部処理時間を負荷情報として用いる．同期ポイントにさしかかったところで，前回の負荷調整後各プロセスが費やした内部演算処理時間 T を収集し，まず以下の条件式に当てはめる．

$$|T/T_{avg} - 1.0| \geq \alpha \quad (1)$$

T_{avg} は平均内部処理時間，α は閾値である．$\alpha = 0.1$ とした場合には，内部処理時間 T が平均内部処理時間 T_{avg} の ±10%の範囲に収まらない場合に上記条件式は真となる．この結果を過去数回分保持しておき，1ノードでも連続して平均時間を上回るもしくは下回る傾向が続いた場合に，実際にデータ再分散処理が実行される．このように履歴情報を利用することにより，瞬間的な負荷変動によるデータ分散を回避する．

なお，後述の性能評価では $\alpha = 0.1$ と設定し，履歴は3回分保持するようにした．

4.4 API

前述したように，期待したとおりの負荷分散を実現するためには，前もって同期ポイントや共有するデータなどを設定する必要がある．今回はライブラリレベルでの実装のため，これらの設定はユーザによって指示されなくてはならない．そこで**表2**に示すAPIを定め，実装した．**図5**に本APIを使用したプログラム構成例を示す．このAPIを用いてデータの初期化

表 2 動的負荷分散用 API
Table 2 API for dynamic load balancing.

機能	関数
動的負荷分散機構の初期化	`MPI_SD_init(int argc, char **argv, int interval)`
共有データの設定	`MPI_SD_shared_data(void *var, char *tag, MPI_Datatype dtype, int dim, int *arysize, int dir, int blksize, int *sleeves)`
共通データの設定	`MPI_SD_common_data(void *var, char *tag, MPI_Datatype dtype, int count)`
同期変数の設定	`MPI_SD_sync_param(void *var, char *tag, int param)`
コールバック関数の登録	`MPI_SD_add_cbfunc(void *(*func)(void))`
コールバック関数の削除	`MPI_SD_remove_cbfunc(void *(*func)(void))`
負荷調整の実行	`MPI_SD_schedpoint(void)`

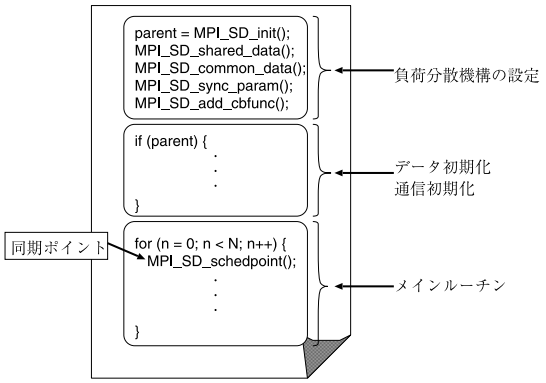

図 5 プログラム構成
Fig. 5 Program structure.

（および通信初期化）を行う前に分散対象データの設定を行う．また負荷再分散を実行したい箇所に同期関数を挿入する．

以降では，各種データの設定，同期ポイントの設定，そしてそれ以外の API の機能について説明する．

4.4.1 共有データの設定（MPI_SD_shared_data）

負荷分散は計算対象となるデータの量をプロセス間で調整することで実現される．そのようなデータのことを本機構では共有データと呼ぶことにする．負荷分散を行う前に，どの共有データをプロセス間で移動させる必要があるのかあらかじめ設定しなければならない．そこで図 5 に示すように，データ初期化部分の前にプロセス間で分散する共有データを設定する．

共有データは多くの場合多次元配列からなる．そこで，本機構では表 2 に示す MPI_SD_shared_data() 関数を用いて配列の属性および分割方法☆を指定する．本関数を呼び出すことにより，`dtype` 型の `dim` 次元配列（`arysize[0]` × `arysize[1]` × ··· × `arysize[dim-1]`）の共有データを `dir` 次元方向に分割して各プロセスにデータを分散する．このとき，初期状態として自プロセスには `blksize` 分のデータを配置することを指示する．本関数により，指定されたサイズ（自プロセス分だけ）のデータ領域が確保され，その領域へのポインタが `var` に返される．また，袖領域を必要とする計算を行う場合には，第八引数 `sleeves` に左右にどれだけの袖領域を持つのかということを設定する．以上により，ユーザは本関数呼び出し後は第一引数の変数 `var` を用いて自由にデータ領域にアクセスできる．なお，同期ポイントでの負荷調整により，各プロセスが持つ共有データの量は変化し，それに合わせたデータ領域が新たに作成されるが，その際変数 `var` の指す先が暗黙のうちに新しく確保されたデータ領域に付け替えられる．これにより，ソースコード上はデータ再分散する前と後で変わりなく，同一変数を介して共有データへアクセスすることができる．

4.4.2 共通データの設定（MPI_SD_common_data）

共有データのように複数プロセスで分散して持つデータ以外に，各プロセスでまったく同じ内容を保持するデータも存在する．その典型がメインループのループ変数である．一般的な並列アプリケーションの場合，各イタレーションごとにプロセス間でデータ通信や同期処理が入るため，全プロセスのループ変数 n は揃っていることが多い．しかし，本機構では実行途中からプロセス数が増加することを許しているため，アプリケーション起動時に生成された初期プロセス群が 100 回目のループを回っているときに新たに生成されたプロセスも，100 回目のループに即座に突入しなければならない．

このようなデータを登録するために，MPI_SD_ common_data() 関数を用いる．表 2 に示すように，引数として共通データへのポインタ `var`，識別タグ `tag`，データタイプ `dtype`，そして要素数 `count` を渡す．本関数により共通データのデータ領域を記録しておき，新規プロセスが最初の同期ポイントに到達したときに既存プロセスのいずれかからすべての共通データの内容を転送し，そしてそれらをタグが等しい共通データ

☆ 現在のところ，対象データが多次元配列であったとしても，一次元方向のデータ分割にしか対応していない．

のデータ領域にコピーする．これにより，最初の同期ポイント直前までは $n = 0$ であったループ変数が，当該同期ポイント直後には $n = 100$ に自動的に修正され，既存プロセスと足並みが揃う．

4.4.3 同期変数の設定（MPI_SD_sync_param）

並列アプリケーションでは，ブロックサイズや要素数など，自プロセスが持つ共有データ量に依存した変数を用いて計算ループのカウント数などを決定している．また，共有データ領域のどの部分を割り当てられたのかによって，スタート/エンドオフセット値などが決まる．したがって，これらの値は自プロセスが持つ共有データに応じて暗黙のうちに変更されなければならない．そこで，データ再分散に同期して変更する変数を設定するための関数 MPI_SD_sync_param() を提供する．

MPI_SD_sync_param() では，同期させる変数へのポインタ `var`，識別タグ `tag`，パラメータの種類 `param` を引数として渡す．`tag` は `MPI_SD_shared_data()` 関数で指定されたタグのどれかでなくてはならず，どの共有変数のパラメータと同期するのかを区別するために使用する．また，同期するパラメータの種類として，共有データごとに各々設定されるブロックサイズ，オフセット値，要素数以外に，特定の共有データには依存しないプロセス ID やプロセス数も指定できるようになっている．

本関数によって設定された変数の内容は，データ再分散が行われて担当データやプロセス数などに変化が生じた場合にのみ変更される．

4.4.4 同期ポイントの設定（MPI_SD_schedpoint）

本機構では，MPI_SD_schedpoint() 関数をプログラム中に明示的に挿入することで同期ポイントを指定する（図 5 参照）．つねに負荷を均等にするためには，定期的にこの MPI_SD_schedpoint() 関数を呼ぶ必要がある．したがって，一般的にはメインループ中に本関数を埋め込むことになる．

この同期ポイントでは，共有データは安定した状態であることをユーザが保証しなければならない．たとえば，共有データを用いて計算している途中に同期ポイントを設けてしまうと，一部の共有データはまだ更新されていないにもかかわらずプロセス間でデータ交換されてしまい，データの整合性がとれなくなってしまう．したがって，計算処理前後もしくはデータ通信前後のいずれかに MPI_SD_schedpoint() を挿入するようにする．ただし，この条件を満たしている限り，プログラムの複数箇所に MPI_SD_schedpoint() を埋め込んでもよい．

MPI_SD_schedpoint() 関数がメインループ中に埋め込まれた場合，毎イタレーション呼び出されることになる（MPI_SD_schedpoint ( ) 関数が複数挿入されていた場合は複数回呼ばれる）．したがって，負荷分散スケジューリングの間隔は数ミリ秒かもしれないし，数時間かもしれない．1 イタレーションあたりどれくらい実行時間を要するかはアプリケーション特性および並列度に大きく依存し，ユーザが正確にこの時間を判断するのは難しい．そのうえさらに，本機構はプロセス並列度を動的に変更するため，1 イタレーションの所要時間は刻々と変化する．同期ポイント間隔が狭すぎると負荷調整のコストが大きくなり，反対に広すぎると実行環境の変化についていけず，期待どおりの負荷分散が行えない．そこで本機構ではこの問題を解決するため，指定された時間間隔で負荷が調整されるように，1 つ前の負荷調整時に次に何回 MPI_SD_schedpoint() が呼ばれたら負荷調整を行うかを決定するようにする．その回数に達するまでは MPI_SD_schedpoint() が呼ばれても何もせずにユーザコードに復帰する．これにより，同期の間隔が狭くても負荷調整コストは抑えることができる．この時間間隔は，初期化関数 MPI_SD_init() 関数の第三引数で指定する．

4.4.5 その他の機能

共有データのブロックサイズなど単純なパラメータではなく，たとえば通信相手などアプリケーション特性に強く依存するものもある．これらは MPI_SD_sync_param() では設定できないため，負荷調整のたびに別途設定し直す必要がある．そこで，通信相手の設定などを行う関数（コールバック関数）をユーザに用意してもらい，そのコールバック関数を負荷調整実行直後に自動的に呼び出すことでこの問題を解決する．そのために，ユーザが作成したコールバック関数を登録するための関数 MPI_SD_add_cbfunc() 関数を提供する．なお，コールバック関数は MPI_SD_add_cbfunc() 関数を複数回呼び出すことでいくつでも設定できる．また，MPI_SD_remove_cbfunc() 関数でコールバック関数を削除することもできる．

4.5 サンプルプログラム

上記の API を利用した具体例として，行列ベクトル積を行うプログラムに本機構を適用したソースコードをオリジナルコードとともに本稿末の**図 6** に示す．図 6 (b) 28〜35 行目が負荷分散のための初期設定部分である．配列 a，b の領域はオリジナルコードでは 7 行目で静的に確保されているが，適用後は 29〜32 行

```
 1 #include <stdio.h>
 2 #include <mpi.h>
 3
 4 #define N           10000
 5 #define LOOP    100
 6
 7 double a[N][N], b[N], x[N];
 8
 9 int
10 main(int argc, char *argv[])
11 {
12    int i, j, k, l;
13    int myrank, npe;
14    int blksize, extra;
15
16    MPI_Init(&argc, &argv);
17    MPI_Comm_size(MPI_COMM_WORLD, &npe);
18    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
19
20    blksize = N / npe;
21    extra   = N % npe;
22    if (myrank < extra) blksize += 1;
23        ・・・
24
25    /* main routine */
26    for (l = 0; l < LOOP; l++) {
27       for (i = 0; i < blksize; i++) {
28          for (j = 0; j < N; j++) {
29             b[i] += a[i][j] * x[j];
30          }
31       }
32          ・・・
33    }
34          ・・・
```

(a) オリジナル

```
 1 #include <stdio.h>
 2 #include <mpi.h>
 3 #include <mpi_sd.h>
 4
 5 #define N           10000
 6 #define LOOP    100
 7
 8  double **a, *b, x[N];
 9
10 int
11 main(int argc, char *argv[])
12 {
13    int i, j, k, l;
14    int myrank, npe;
15    int blksize, extra;
16    int arysize[2], sleeves[2], veclen[1];
17
18    MPI_Init(&argc, &argv);
19    MPI_Comm_size(MPI_COMM_WORLD, &npe);
20    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
21
22    blksize = N / npe;
23    extra   = N % npe;
24    if (myrank < extra) blksize += 1;
25    arysize[0] = N; arysize[1] = N;
26    veclen[0]  = N;
27    sleeves[0] = 0; sleeves[1] = 0;
28    MPI_SD_init(argc, argv, 3);
29    MPI_SD_shared_data(&a, "a", MPI_DOUBLE, 2, arysize,
30                                    0, blksize, sleeves);
31    MPI_SD_shared_data(&b, "b", MPI_DOUBLE, 1, veclen,
32                                    0, blksize, sleeves);
33    MPI_SD_sync_param(&blksize, "a",
34                                     SD_PARAM_BLKLEN);
35     MPI_SD_common_data(&l, "l", MPI_INT, 1);
36        ・・・
37
38    /* main routine */
39    for (l = 0; l < LOOP; l++) {
40       MPI_SD_schedpoint();
41       for (i = 0; i < blksize; i++) {
42          for (j = 0; j < N; j++) {
43             b[i] += a[i][j] * x[j];
44          }
45       }
46          ・・・
47    }
48          ・・・
```

(b) 負荷分散機能組み込み版

図 6 サンプルプログラム（行列ベクトル積）
Fig. 6 Sample Program (Matrix-vector product).

目の `MPI_SD_shared_data()` 呼出しによって動的に割り付けられている．また，メインルーチン内 40 行目に `MPI_SD_schedpoint()` 関数が埋め込まれているが，それ以外の部分はオリジナルコードをそのまま用いることができることに注目されたい．

4.6 処理フロー

本機構を適用した場合の全体的な処理の流れを，**図 7** をもとに説明する．

まず，プログラムの先頭で，動的負荷分散機構を利用するための初期化処理および共有データ・共通データ・同期変数の設定をする．また，必要ならばコールバック関数を登録する．以上により，データ格納領域は確保されるので，その後に通常のデータ初期化および通信初期化を行う．

メインルーチンに入った後は `MPI_SD_schedpoint()` が埋め込まれたところでスケジューリングが実行され，

表 3 実験システム
Table 3 Experiment system.

	CPU	L2 キャッシュ	OS	台数	インターコネクト	備考
クラスタ A	PentiumIII 800 MHz	512 KB	Linux 2.4.18	10 台	100 Mbps	—
クラスタ B	PentiumIII 800 MHz	512 KB	Linux 2.4.7	16 台	100 Mbps	高負荷状態
クラスタ C	PentiumIII 1.4 GHz	512 KB	Linux 2.4.7	6 台	100 Mbps	—

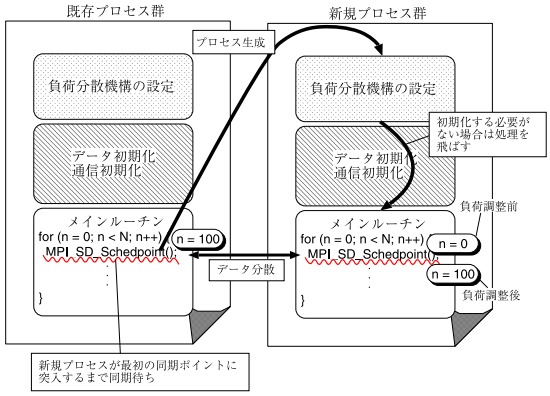

図 7 動的負荷分散機構の処理フロー
Fig. 7 Processing flow of dynamic load balancing.

負荷に不均衡が生じていた場合にはデータが再分散される．このとき，現在使用しているノード以外に使用可能なノードが存在した場合には，そのノードも負荷分散対象とする．そのためにまず当該ノード上に新規プロセスを生成し，そのプロセスが最初の同期ポイントにさしかかるまで，既存プロセス群は負荷調整ルーチン途中で待機する．そして新規プロセスを含めたすべてのプロセスが同期ポイントに入ったところで，共有データの再分散，共通データのコピー，その他パラメータの再設定が行われ，通常の計算処理に復帰する．一方，使用ノード数を減らす場合は，削除対象となるノード上のプロセスへデータを割り付けないようにデータを再分散して，当該プロセスを終了させる．以上の負荷分散処理を適当な間隔で行うことで，システム状態に応じた適切な負荷分散が可能となる．

5. 性能評価

本節では，非圧縮流体解析コードの性能評価に使われる姫野ベンチマーク[5]を用いて，実装した動的負荷分散機構の性能を評価する．姫野ベンチマークとはポアッソン方程式をヤコビ反復法で解くプログラムであり，データ転送は隣り合うノード間でのみ行われる．動的負荷分散機能を適用するにあたり，分散対象となる多次元配列は最初の次元方向にブロック分割して各ノードに分散する．また，データ再分散チェックのための同期は 20 秒間隔でとる．

実験に用いるマシンを**表 3** に示す．各クラスタ内のノード間は 100 Mbps のスイッチングハブを介して接続され，またクラスタ間を接続するために各クラスタ用スイッチングハブをさらに別の 100 Mbps スイッチングハブにカスケード接続している．なお，表中クラスタ B の備考欄に “高負荷状態” とあるのは，レジスタ上で加算を繰り返すだけのプログラムを別に実行することにより，CPU 負荷をかけた状態であることを意味する．このため，クラスタ B 上で姫野ベンチマークを走らせた場合には，無負荷状態時のほぼ半分の性能しか得られなくなる．

5.1 オリジナルプログラムとの性能比較

まず初めに，クラスタ A の 10 ノードを用いてオリジナルのプログラムと動的負荷分散機能を適用したプログラムの性能を比較する．データサイズはあらかじめ用意されているミドルサイズ（$128 \times 128 \times 256$ サイズ）を用いる．なお，今回の実験ではノード数は 10 台で固定されており，またすべてのノードの性能は等しいので，動的負荷分散機能を適用したプログラムであってもデータ再分散は実行されない．

200 シミュレーションステップを実行するのに要した時間は**表 4** のようになった（5 回試行したうち，最良の結果を記載）．ここで，まったく手を加えていないオリジナルプログラムでの測定結果が “オリジナル” であり，“負荷分散適用” とあるのは負荷分散用のコードを追加したプログラムでの測定結果である．“オリジナル” は静的にデータ領域を確保しているので，**図 8** に示すメインルーチン計算コア部分での多次元配列要素のアクセスは一次元に展開されてアクセスされるよう，コンパイラにより最適化される．一方，“負荷分散適用” はデータ領域を動的に確保しており各次元方向のサイズがコンパイル時には不定のため，上記のデータアクセスに関する最適化が施せない．よって，“負荷分散適用” は “オリジナル” より 2 割ほど遅い結果となった．そこで，**図 9** 上部のようなマクロを定義し，図下部のように多次元配列を線形に展開してアクセスするように手動でコードを修正した．その測定結果が “負荷分散適用（アクセス最適化版）” である．この手動最適化により，実行時間は “オリジナル” とほぼ同等になった．

表 4 実験結果（実行時間）
Table 4 Execution time comparison.

プログラム	実行時間（秒）
オリジナル	25.35
負荷分散適用	30.41
負荷分散適用（アクセス最適化版）	25.40

```
for (n = 0; n < NN; n++) {
  . . .
  for (i = 1; i < blksize + 1; i++) {
    for (j = 1; j< jmax - 1; j++) {
      for (k = 1 ; k < kmax - 1; k++) {
        s0 = a[i][j][k][0] * p[i+1][j  ][k  ] +
             a[i][j][k][1] * p[i  ][j+1][k  ] +
             a[i][j][k][2] * p[i  ][j  ][k+1] +
             b[i][j][k][0] * ( p[i+1][j+1][k  ] - p[i+1][j-1][k  ] -
                               p[i-1][j+1][k  ] + p[i-1][j-1][k  ] ) +
             b[i][j][k][1] * ( p[i  ][j+1][k+1] - p[i  ][j-1][k+1] -
                               p[i  ][j+1][k-1] + p[i  ][j-1][k-1] ) +
             b[i][j][k][2] * ( p[i+1][j  ][k+1] - p[i-1][j  ][k+1] -
                               p[i+1][j  ][k-1] + p[i-1][j  ][k-1] ) +
             c[i][j][k][0] * p[i-1][j  ][k  ] +
             c[i][j][k][1] * p[i  ][j-1][k  ] +
             c[i][j][k][2] * p[i  ][j  ][k-1] +
             wk1[i][j][k];

        ss = ( s0 * a[i][j][k][3] - p[i][j][k] ) * bnd[i][j][k];
        gosa = gosa + ss*ss;
        wk2[i][j][k] = p[i][j][k] + omega * ss;
      }
    }
  }
 . . .
}
```

図 8 姫野ベンチマークの計算コア部分
Fig. 8 Calculation core portion of HIMENO benchmark program.

```
（一次元アクセス用のマクロ）
#define ary3(name, x, y, z) \
            (name[(x * YSIZE * ZSIZE) + (y * ZSIZE) + z])


（図7最内ループ最終行の場合の置換）
wk2p = &wk2[0][0][0];
pp   = &p[0][0][0];
ary3(wk2p, i, j, k) = ary3(pp, i, j, k) + omega * ss;
```

図 9 多次元アクセスから一次元アクセスへの置換
Fig. 9 Replacement multi-dimensional access with linear access.

本負荷分散機構を適用するとデータサイズが動的に変わるという性質上，共有データを動的に確保する必要がある．そのため，姫野ベンチマークのように静的に共有データを確保しているオリジナルプログラムに対して性能が下回る場合がある．そのような場合にオリジナルプログラムと同等性能を得るためには，図 9 のような最適化を手動で行わなければならない．

5.2 動的負荷分散機構の評価

続いて表 3 に示す各 PC クラスタを動的に組合せを変えながらベンチマークプログラムを実行することで，負荷分散機構の性能を評価する．本実験では，まずクラスタ（A + B）構成でベンチマークを実行し，その後適当な間隔をあけて（A + B + C）→（A + C）と動的に構成を変える．つまり，使用可能ノード数は 26 → 32 → 16 と変わることになる．なお，前述したように，クラスタ組合せ変更，つまりノード数変更の指示はコマンドラインから手動で指示する．ただし，ベンチマーク自体は一度も止まることなく，測定区間中は継続して走り続けながら自律的に構成が変わることに注意されたい．また，パラメータを調節して，プログラム全体で約 3.6 GB のデータ領域を必要となるように設定してある．

測定結果を**図 10**，**図 11**，**図 12** に示す．どの図も横軸にはシミュレーションステップをとっている．縦軸にはそれぞれ，1 シミュレーションステップあたりの実行時間，個々のノードが担当するブロックサイズ（つまり共有データ量）のクラスタごとの平均，そしてデータ再分散時間をとっている．なお，図 10 の結果にはデータ再分散の時間は含まれていない．また，図上部の範囲指定は，その間でどのクラスタ組を使っていたのかを表している．

ベンチマーク起動時には，クラスタ A，B の各ノードには均等にデータが分散されている．この両クラスタの各ノードには同じ PentiumIII 800 MHz プロセッサが使用されているので，無負荷状態であれば均等に負荷分散されるはずである．しかしクラスタ B には負荷をかけているため，この時点ですでに負荷の不均衡が生じてしまい，その結果 1 ステップあたり約 1.4 秒と，シミュレーション全体を通して最も実行時間を要している．しかし，30 ステップあたりでデータの再分散が行われたことにより不均衡が是正され，1.1 秒程度まで最適化されているのが図 10 および図 11 から読みとれる．

また，150 ステップ目あたりでクラスタ C が追加されている．この時点ではクラスタ C のノードに関する負荷情報はまだ収集されていないため，負荷分散機構は仮の割当てとして，実行中のノードの中で最も担当ブロックサイズの小さいものとほぼ同じサイズのブロックを割り当てて負荷を分散する．同様に，ノード数が減少した直後（350 ステップ目あたり）に各ノードに割り当てられるブロックサイズも厳密ではない．しかし，しばらく実行し負荷情報を収集することにより，各ノードの実効性能に従ってデータが再分散される．

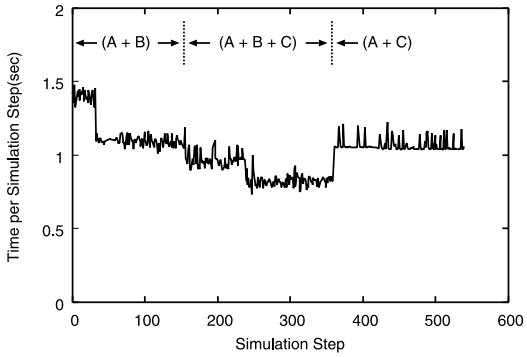

図 10 実験結果（1 ステップあたりの実行時間）
Fig. 10 Elapsed time per simulation step.

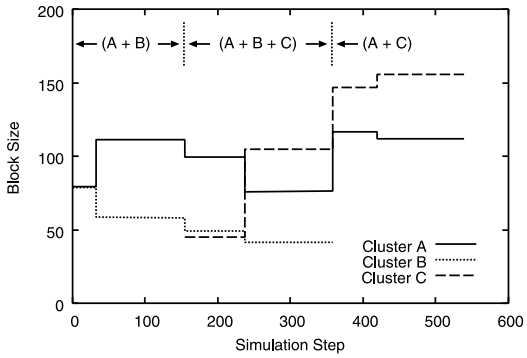

図 11 実験結果（ブロックサイズ）
Fig. 11 Block size.

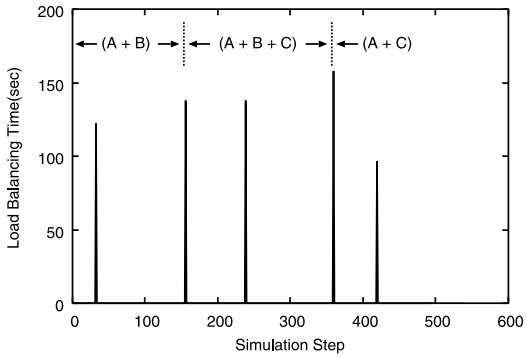

図 12 実験結果（データ再分散時間）
Fig. 12 Data redistribution time.

最終的に（A + C）の 16 台構成でベンチマークが実行される．クロック周波数で比較すると，クラスタ C のノードはクラスタ A のノードの 1.75 倍なのに対して，クラスタ C の平均ブロックサイズはクラスタ A の約 1.39 倍でしかない（図 11）．そこで，同様に姫野ベンチマークを用いて各クラスタのノード単体性能およびクラスタ A のノードを基準とした性能比を求めた（**表 5**）．この結果からも，ノード単体でもクラスタ C はクラスタ A の 1.4 倍程度の性能しか発揮できていないことが分かる．これは，ワーキングセットがキャッシュに収まり切らず，またキャッシュブロッキングなどのメモリアクセス最適化を行っていないので，メモリアクセスがボトルネックになっていることが原因と推測される．つまり，ブロックサイズ比が 1.39 倍になっているのは正常であり，実装した負荷分散機構によるデータ分散は性能に見合って正しく行えているということが実証されたことになる．

表 5 ノード単体性能
Table 5 Each node's performance.

	ノード単体性能（MFLOPS）	性能比
クラスタ A	143	1
クラスタ B	141	0.99
クラスタ C	201	1.41

続いて，データ再分散に要する時間に注目する（図 12）．同期間隔を 20 秒と設定したため，約 20 ステップ間隔でデータ再分散のチェックが入る．負荷調整をする必要がない場合はチェックルーチンのオーバヘッドはほとんどなく即座にユーザコードに戻る一方で，データ再分散が必要となった場合にはその処理に長時間要していることが図から読みとれる．たとえば，350 ステップ目あたりの負荷分散処理には 160 秒ほど要している．この処理時間は交換されるデータ量（本実験では最大で約 3.6 GB）に比例して増大する．つまり，大規模なアプリケーションになるほどデータ分散が長時間に及ぶことになる．実運用時にはノード数の変更は頻繁には生じないと考えられるので，この場合はデータ再分散の時間はそれほど懸念する必要はない．しかし，ノード数が変わらなくても計算負荷が変動する不規則な HPC アプリケーションは多数存在し，そのようなアプリケーションでは頻繁にデータが再分散される可能性がある．したがって，データマッピング方法を現在の単純なブロック分割ではなく，HPF などのように抽象プロセッサモデルを検討するなど，さらなる改良が必要である．また，ノード間のデータ転送は先頭ノードから順次行うという非常に単純なアルゴリズムを採用しており，このことも時間増大の原因となっているので，データ転送アルゴリズムの改善も今後の課題である．

ほかの解決策として，そもそもデータ再分散をしないことでこの問題を回避する方法があげられる．たとえば頻繁に負荷変動が生じることをあらかじめ予測し適当に間引いて負荷調整したり，負荷調整後の性能とデータ交換オーバヘッドのトレードオフを考慮してデータ再分散をやめたりする機能を盛り込むようにすればよい．現在，リソース管理層に関連して，負荷予

測技術やアプリケーションからのフィードバック情報に基づくリソース再構成の研究を進めている．そこで，単にノード数増減の指示だけでなく，負荷調整すべきかどうかの判断材料もリソース管理層から受け取るようにすることで，これらの機能も実現できると考えている．

6. 関連研究

動的負荷分散に関する研究は，個々のアプリケーション特定のものから，ライブラリレベル，コンパイラレベルに至るまで数多く存在する．コンパイラレベルの実装としては，文献 7)，8) がある．前者は，本研究同様，配列データをブロックに分割して負荷分散を実現している．後者は HPF と同じように抽象プロセッサの概念を導入し，物理プロセッサに割り当てる抽象プロセッサ数を動的に調整することで負荷を分散する．アプリケーションレベルで負荷分散を行う研究として，文献 9)，10) などがあげられる．以上の研究はいずれも使用ノード数は固定されている場合を想定している．それに対して，本研究は使用可能ノード数が動的に変わるケースにも対応している．

本研究同様，並列度を動的に変えて負荷分散を実現する研究としては文献 6) がある．文献 6) では，負荷不均衡が生じた場合にプロセス間でデータを再分散するのではなく，空いているプロセッサをリクルートしてきて，負荷の高いプロセスに当該プロセッサを割り当ててスレッド数を増やすことで負荷不均衡を是正する（彼らはこの手法を “Dynamic Power Balancing” と呼んでいる）．文献では，SGI-Cray Origin 2000 上で AMR（Adaptive Mesh Refinement）アプリケーションを実行した場合を実例としてあげている．この手法では各ノードが共有メモリ型マシンでなければならないが，我々の手法は分散メモリシステムをターゲットとしており，適用範囲が広い．

7. おわりに

本研究ではプロセス数可変の動的負荷分散機構を提案し，LAM/MPI を用いて同機構を実装した．PC クラスタを用いた実験から，本動的負荷分散機構はシステム構成，負荷状況に対応して適切にデータ分散していることが実証され，その有効性が確認された．

ただし，本機構だけではユーザへの負担がまだかなり大きい．したがって，今後の課題として並列化コンパイラへの機能組込みが第 1 にあげられる．また，現在は手動で指示しているノード追加・削除をリソース管理層，システムモニタリングツールと連動して自動化することも重要な課題である．

謝辞 本研究は新エネルギー・産業技術総合開発機構基盤技術研究促進事業「高信頼・低消費電力サーバの研究開発」によって行われた．

参考文献

1) OpenMP Home Page: http://www.openmp.org/.
2) HPF Home Page : http://www.crpc.rice.edu/HPFF/.
3) MPI Forum: http://www.mpi-forum.org/.
4) LAM/MPI Home Page: http://www.lam-mpi.org/.
5) Himeno Benchmark: http://w3cic.riken.go.jp/HPC/HimenoBMT/index.html.
6) Huang, W. and Tafti, D: A Parallel Computing Framework for Dynamic Power Balancing in Adaptive Mesh Refinement Applications, *Proc. Parallel Computational Fluid Dynamics '99* (1999).
7) 荒木拓也，村井 均，蒲池恒彦，妹尾義樹：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，並列処理シンポジウム JSPP2002, pp.131–138 (2002).
8) 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.202–212 (1997).
9) 笹生 健，松岡 聡，建部修見：ヘテロなクラスタ環境における並列 LINPACK アルゴリズム，並列処理シンポジウム JSPP2002，pp.71–78 (2002).
10) 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).

(平成 15 年 1 月 24 日受付)
(平成 15 年 5 月 10 日採録)

松原 正純（正会員）

昭和 48 年生．平成 8 年筑波大学第三学群情報学類卒業．平成 13 年同大学大学院工学研究科博士後期課程修了．同年（株）富士通研究所入社．博士（工学）．クラスタシステム，自律コンピューティングの研究に従事．

鈴木　和宏

昭和 41 年生．平成 2 年横浜国立大学工学部情報工学科卒業．平成 4 年東京大学工学系研究科情報工学修士課程修了．同年（株）富士通研究所入社．クラスタシステムの研究に従事．

勝野　　昭

昭和 60 年大阪大学工学部卒業．同年（株）富士通研究所入社．LSI 設計に従事．現在，IT コア研究所に勤務．コンピュータアーキテクチャの研究開発に従事．