Vol. 46 No. 4 情報処理学会論文誌 Apr. 2005

マイグレーションを支援する分散集合オブジェクト

高橋 慧[†] 田浦 健次朗[†] 近山 隆[†]

グリッド環境におけるプログラムは，与えられた資源を最大限活用するために動的なプロセッサ数の増減に対応することが求められている．一方でプログラムの記述においては，プロセッサ数の増減を意識しないような書き方が出来ることが望ましい．

既存の分散オブジェクトモデルでは，オブジェクト指向により簡単な記述が行え，オブジェクトのマイグレーションによりプロセッサの増減にも対応できる．しかし，分散配列などプロセッサ間にまたがる大きなデータを記述しようとすると，データとオブジェクトの対応が動的に変化してしまい，本来のプログラムの記述が非常に複雑になってしまう．

本発表では，大きな配列などインデックスにより識別される集合を用いる際，プロセッサ数の変化に対応した並列プログラムを簡単に記述できる,「分散集合オブジェクトモデル」を提案する．これにより，プロセッサ間に断片として分散しているデータが，プログラマからは一つのオブジェクトとして見えるようになる．メソッドの呼び出しはインデックスの範囲を指定して行うため，プログラムはデータの配分を意識せずに記述できる．実装においては要素と断片の対応を分散保持し，メッセージの集中を防いで性能を損なわないようにした．

我々はこの分散集合オブジェクトの処理系を実装し，その上でアプリケーションを記述して動作の確認と性能評価を行った．

Distributed Aggregate with Migration

Kei Takahashi,[†] Kenjiro Taura[†] and Takashi Chikayama[†]

In Grid environment, it is required to allow to change the number of processors that participates in the computation in order to utilize available resources efficiently. At the same time, object-oriented programming model is desirable for easy description. With distributed object models, a program can allow to change the number of processors by object migration. However, with this model, it is very hard to implement distributed large data structures – such as distributed array – allowing to change the number of processors.

We propose "distributed aggregate model" as an extension of distributed object model. It handles a data structure identified with indices, like distributed array. In our distributed aggregate model, each object can be distributed among processors as fractions, but looks like one object to programmers. Thus a programmer can write programs without worrying about its distribution. Every processor has the conversion table of indices and processors, so messages do not concentrate on one processor.

We implemented this framework, wrote a programming example of partial differential equation and n-body problem. The program operated correctly when the number of processors changes dynamically, and we evaluated its performance.

1. はじめに

グリッド環境においては，利用可能な資源は負荷の増減や故障により変動する．このため，与えられた資源を最大限活用するためにはプログラムはプロセッサの増減に対応する必要がある．プロセッサの増減に対応するためには，動的にデータ・タスクを移動させ，他のプロセッサにデータが移動したことを伝達する必要がある．

現状では多くの並列プログラムはメッセージパッシングを用いて記述されているが，これは多くの場合通信相手の指定にプロセッサ番号を用いるため，プロセッサ数が増減すると動的にメッセージの送信先を変える必要がある．送信先を指定するためにはプログラマがデータとプロセッサの対応を把握するコードを書く必要があり，記述は難しい．

分散オブジェクトではオブジェクト指向の記述が行え，またデータの識別にプロセッサではなくオブジェクトを用いるので，あるオブジェクトがあるプロセッ

† 東京大学
University of Tokyo

サから他のプロセッサに移動しても，メソッド呼び出し先を変更する必要はない．しかし，分散配列などプロセッサ間にまたがる大きなデータを扱う場合には，オブジェクトを分割したり併合したりする必要が生じるので，メソッド呼び出しの対象となるオブジェクトを変更する必要が生じ，記述は簡単ではない．

本発表では，大きな配列などインデックスにより識別される集合を用いる際，プロセッサ数の変化に対応した並列プログラムを簡単に記述できる，「分散集合オブジェクトモブル」を提案する．これにより，プロセッサ間に断片として分散しているデータが，プログラマからは一つのオブジェクトとして見えるようになる．メソッドの呼び出しはプロセッサ番号ではなく，オブジェクトの識別子とインデックスの範囲を指定して行う．このため，プログラマはプロセッサ間でのデータの移動を意識する必要がない．実装においては要素と断片の対応を分散保持し，メッセージの集中を防いで性能を損なわないようにした．

我々はこの分散集合オブジェクトの処理系を実装し，その下で偏微分方程式・N 体問題のアプリケーションを記述して動作の確認と性能評価を行った．

2. 既存の並列プログラミングモデル

ここでは並列プログラム記述のための手法として，メッセージパッシング，分散オブジェクト，Concurrent Aggregate を用いた際の特徴と問題点について述べる．

2.1 メッセージパッシング

メッセージパッシングモデルは，最も基本的で，広く使われている並列計算モデルである．MPI[1] が一例として挙げられる．このモデルでは send() と recv() の二つの関数が用意され，相手プロセッサを指定して通信を行う．

一般的なメッセージパッシングモデルは，プロセッサの増減に対応するのが難しい．一般にメッセージはプロセッサ番号を指定して送られるが，プロセッサ数が変化した場合，動的にメッセージを送信する先のプロセッサを変える必要があり，記述は簡単ではない．

一方で，このモデルは下層で実際に行われる処理に近く，呼び出し自体のコストが小さいため，無駄の無いプログラムを記述することができる．また，ローカルのデータとリモートのデータがはっきりと区別され，ローカルのデータには逐次と同じ速度でアクセスできる．

Phoenix モデル[4] は，メッセージパッシングモデルを拡張したものである．このモデルでは，計算前に物理的なプロセッサ ID とは別の，複数の仮想的なプロセッサ番号 (仮想ノード番号) の集合を考える．そして，この仮想ノード集合を計算に参加しているプロセッサで過不足無く分割する．計算に参加する場合は，他のプロセッサから集合の一部を割り当てられ，離れる場合は他のプロセッサに集合を委譲する．ここで，メッセージの送信はこの番号を指定して行い，送信されたメッセージは指定されたを持つ番号をプロセッサによって受信される．プロセッサが担当する番号は変化し得るが，メッセージの送信先を変更する必要は無いため，プロセッサ数の増減に対応したプログラムを容易に記述できる．

2.2 共有メモリモデル

共有メモリとは，複数のプロセッサが特定の領域のメモリを共用し，ローカルのメモリと同様に書き込みや読み出しが出来るシステムである．アクセスされるデータは場合によってはリモートにあるため，下層で通信が行われる．

このモデルは，プロセッサ増減への対応が容易である．メモリ番地と実際にそのデータを持つプロセッサの対応付けを動的に変えられるようにすると，プログラムの記述を変えずにプロセッサの増減に対応できるようなシステムを構築できる．

一方で，どの番地がどのプロセッサに割り当てられているかを意識せずにプログラムを書くと，離れたプロセッサにあるメモリへのアクセスが多く回数発生する．望ましいのは，離れたメモリに対しての処理はそのデータを持つプロセッサが行うことである．このためにはプロセッサとメモリ領域の対応をプログラマが把握する必要があり，本来の記述の容易さは損なわれてしまう．

2.3 分散オブジェクトモデル

分散オブジェクトはオブジェクト指向を基本に，リモートのオブジェクトについて，メソッド呼び出し (RMI : remote method invocation) が行えるものである．呼び出されたオブジェクトを持つプロセッサではメソッドの処理を行い，必要に応じてその結果を呼び出し元のプロセッサに返す．RMI に加え，一度生成されたオブジェクト自体を他のプロセッサに移動させる，マイグレーションと呼ばれる技術を組み合わせると，プロセッサ数の増減に対応することができる．マイグレーションにより，そのオブジェクトに関してのメソッドの処理もそのプロセッサに委譲されるから，仕事の分担についての記述は変更しなくてもよい．

記述はオブジェクト指向により簡単に行え，自然な形でデータとタスクの関連性が保たれるためという点

がある．外部のプロセスは，オブジェクトのメソッドを通じてしかデータにアクセスしないので，分散共有メモリのようにプロセッサが不必要にリモートのメモリにアクセスすることがない．しかし，単純に RMI とマイグレーションを提供するだけでは，分散配列のようなプロセッサ間にまたがったオブジェクトは簡単に記述できない．これを以下に示す．

大きなデータ集合は，各プロセッサが断片オブジェクトを持つことにより実現される．ここでオブジェクトは複数のプロセッサにまたがって置くことは出来ないため，プロセッサの増加に対応するためには動的に既存のオブジェクトを分割するか，予め各プロセッサがたくさんの小さなオブジェクトを持っている必要がある．前者はプロセッサ増加の前後であるデータを保持するオブジェクトが変わってしまうので，プログラムの記述が難しくなる．後者はプログラムの記述は楽だが，データの分割単位が小さいと実行効率が悪くなってしまうし，分割単位が大きいと任意のプロセッサの増加に対し対応できない．

つまり，プロセッサの増減には基本的には対応出来るが，プロセッサ間にまたがる大きなデータを扱う際には問題がある．

2.4 Concurrent Aggregates

Concurrent Aggregates[3] とは，複数のプロセッサに断片として存在しているオブジェクトに対して，それらを全体で一つのオブジェクトと見てメソッドを呼び出せるようにしたものである．このモデルでは，断片オブジェクトの宣言において，”この操作を他の全ての断片についても行う”という記述が出来るようになっている．外部からある断片のメソッドを呼び出すと，そのメソッド呼び出しが条件によっては他の断片に中継され，最終的にデータ全体にアクセスができる．

このシステムでは，断片オブジェクトが分割・併合・マイグレーションしても，そのオブジェクトを用いる人にとっては，あたかも一つの大きなオブジェクトであるように扱うことができる．メソッドの処理はオブジェクトを持つプロセッサが行うので，タスクの配分は自然に行われる．

一方でこのシステムでは，インデックスによって識別される集合でも，特定の要素に直接アクセスできない．可能なのは，任意の一断片オブジェクトにアクセスすることで，これにより全ての断片オブジェクトへ要素の問い合わせのメッセージが呼ばれる．しかし，配列のようなデータに対し毎回このような操作を行うのは，呼び出しコストが大きすぎる．

3. 提案モデル

本発表では，プロセッサ数が変化する環境において簡単に記述が行え，かつ性能を損なわない「分散集合オブジェクト」を用いた記述モデルを提案する．分散オブジェクトを基本にしているが，オブジェクトはプロセッサ間にまたがって存在でき，また断片のマイグレーションによりプロセッサの増減に対応できる．

3.1 対象とする集合

本システムは，配列やハッシュ表のようにインデックスにより識別される集合を対象とする．並列プログラムにおいては，このデータは断片に分割され，各プロセッサに配置されている．これらは逐次プログラムにおいてはインデックスからデータのアドレスを簡単に計算できるため，高速にアクセス可能である．またプロセッサ数が定まっている並列プログラムにおいても，あるデータを持つプロセッサを簡単に計算できる．一方でプロセッサ数が変化する環境においては，プロセッサとデータの対応は動的に変化するため，データにアクセスするプロセッサはこの対応を保持する必要がある．

3.2 記述モデル

プログラマは，まずクラス定義を記述する．ここで定義するクラスはデータ全体の一部を持つ断片を表し，メソッドもそれに応じて書かれる．その後，プログラマはインスタンスを作成する．外部からはこのオブジェクトは全体で一つに見えるため，プログラマはデータの配分を意識せずにメソッドを呼び出すことができる．またプロセッサの増減に対しては，外部からメソッドを呼ぶことで，断片を退避させたり受け入れたりすることができる．

なおメソッドの同期的な呼び出しは色々検討すべき点があるので，今回の実装では非同期的呼び出しのみをサポートし，メソッドは返り値を持たない．本章ではこれらの詳細について述べる．

3.2.1 断片クラス定義とメソッド呼び出し

本システムで対象とするデータ集合はインデックスを持ち，断片はあるインデックス集合に対応したデータを持っている．例えば，ある断片が [20-30) というインデックス集合を持っていれば，その断片はインデックス [20-30) に対応したデータを持つようになっている．ここで，メソッドはインデックス範囲を指定して呼び出される．すると，指定されたインデックス範囲と重なりを持つような全ての断片に対して，その重なりのインデックス集合を引数にメソッドが呼び出される．例えば，[0-10)・[10-20)・[20-30) という断片

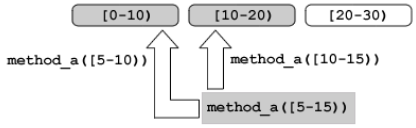

図 1 メソッド呼び出し

が存在するとき，[5-15) という範囲を指定してメソッド $method\_a()$ を呼び出すと，[0-10) を持つ断片では [5-10) を引数に，[10-20) を持つ断片では [10-15) を引数に $method\_a()$ が呼び出される．この様子を図 1 に示す．

ここで，メソッドの呼び出しは一続きの処理として行われ，処理途中にオブジェクトが分割されることはない．また，あるメソッドの実行中に他のメソッドが開始されることもない．メソッドの記述において，自分の断片内に存在しないデータにアクセスする必要が生じたときには，メソッド内から再度別のメソッドを呼び出すことによりデータにアクセスする．これにより，プログラマは断片内のデータと外部のデータを明確に区別してプログラムを記述できる．

実例として，分散配列の指定した要素をインクリメントする擬似コードを以下に示す．この記述によって，配分によらず要素 [15-25) をインクリメントできる．

```
void DArray::increment(IndexSet *is){
  foreach (index <- is)
    data[index]++;
}

int main(){
  ...
  /* DArray のインスタンスを Array とする */
  Array->increment ([15-20));
  ...
}
```

3.2.2 マイグレーション

プロセッサの増減への対応は，マイグレーションにより行う．プログラマは，予めメソッド定義においてその断片オブジェクトを分割する $split()$・併合する $merge()$，バイト列に変換する $serialize()$ とバイト列からのコンストラクタ ($deserialize()$) のメソッドを記述する．これらのメソッドはシステムにより自動的に呼び出される．

本処理系を用いるときには，$join()$・$leave()$ のメソッドを持つ $da :: Manager$ オブジェクトが提供される．処理の実行中にプロセッサが新しく加わった場合は，このプロセッサが $join()$ を呼び出すことにより，既に計算に参加しているプロセッサのうち一つに要求が送信される．このとき，プロセッサの選択は全インデックス範囲からランダムに一つのインデックスを選ぶことにより行われる．$join()$ 要求を受けたプロセッサは自分の持つ断片を二つに分割し，要求元のプロセッサに返信する．

実行中にプロセッサが脱退する場合は，そのプロセッサが $leave()$ を呼ぶことにより，自分の持つ断片オブジェクトが自分以外のランダムなインデックスに送信される．要求を受け取ったプロセッサは，送信された断片を既にある断片と併合したり，それが出来ない時には断片のままで登録する．

システムによって呼び出される $split()$・$merge()$・$serialize()$ はメソッド実行の合間に行われるため，オブジェクトがメソッドの実行中に分割されることはない．

3.2.3 返り値とマイグレーション

今回の実装では，システムは同期的なメソッド呼び出しをサポートせず，プログラマはメソッドを非同期的なメソッドに分割して記述する必要がある．このため RMI のためのメソッドは返り値を持たない．この理由を以下に述べる．

同期的なメソッド呼び出しをサポートするシステムでは，スレッドがメソッド呼び出しの処理を待ってブロックしている状況が頻繁に発生する．ここで，ブロックしているスレッドを持つオブジェクトを分割・併合できるようにした場合，メソッドの実行中にその断片オブジェクトが持つデータ集合が変化することになり，円滑な記述は行えない．ブロックしているスレッドを持つオブジェクトは分割・併合できないことにすると，マイグレーションできる機会は事実上無くなってしまう．

そこで本システムでは，同期的なメソッドの呼び出し側の処理を「呼び出し前の処理」と「呼び出し後の処理」の二つに分割して記述させる．メソッド呼び出しは，呼び出し元のインデックス範囲を引数に行われ，返り値を返すメソッドは，このインデックス範囲に対し返り値に相当する値を指定して「呼び出し後」のメソッドを呼び出す．この様子を図 2 に示す．

これにより，メソッド呼び出しの各切れ目でオブジェクトのマイグレーションが可能になる．またメソッドを呼び出したオブジェクトが返り値を待っているときに分割されても，双方に返り値のデータが送信されるようなプログラムを記述することができる．

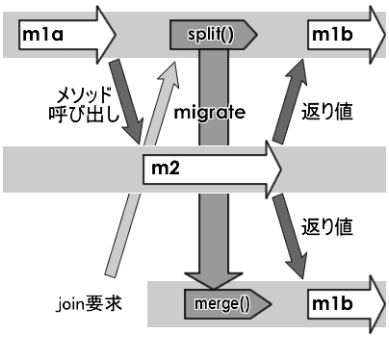

図 2 返り値を挟んでのメソッドの分割

4. 処理系の実装

先に述べたようなメソッド呼び出し・マイグレーションを実現する処理系を，C++で Phoenix ライブラリを用いて実装した．実装にあたっては，スケーラビリティを損なわないよう，一プロセッサに処理が集中しないようにした．

4.1 ルーティング

プロセッサ数が変化する環境においては，要素とプロセッサの対応表を動的に更新する必要がある．一プロセッサがこの表を集中管理すると実装は容易だが，このプロセッサにメッセージが集中し，性能上ボトルネックとなる．このため，本システムではルーティング表を各プロセッサが分散保持し，メッセージは各断片に直接届けられるようになっている．本処理系では，Phoenix ライブラリ[4] がこの処理を行ってくれる．

Phoenix ライブラリでは，計算前にある整数集合を考え，これを計算を担当するプロセッサで過不足無く分割し，メッセージはこの番号を指定して送信される．本ライブラリではインデックス一つに Phoenix の整数番号一つを対応付け，メソッド呼び出しのメッセージは，指定されたインデックス集合のうち一つのインデックスを取って送られる．Phoenix では番号とプロセッサの対応表を分散保持しているので，本ライブラリでもこれを利用することで，メソッド呼び出しのメッセージを直接目的とするプロセッサに送信できる．

4.2 メッセージの配送

メソッド呼び出しはインデックス集合を指定して行われ，各断片で保持するインデックスと重なりを取ってメソッドが呼び出される．メッセージはまず範囲の先頭のインデックスに送られる．受け取ったプロセッサでは，この引数を，自分が保持するインデックス集合との重なりとそれ以外の部分に分割する．そして，重なりを引数にメソッドを呼び出し，残りのインデックス集合を同様に転送する．

我々は同時にツリー状にメッセージが中継されるアルゴリズムも実装した．これは，指定されたインデックス集合がある閾値以上であればこれを分割し，それぞれの先頭のアドレスに送信するものである．インデックス集合が閾値より小さくなったら，メッセージは分割せずに送信する．閾値の選択が適切であれば，この方法は広範囲に対する呼び出しが短時間で行えるが，一回の呼び出しに対し一つの断片が複数回のメソッド呼び出しを受ける可能性がある．

5. アプリケーションの記述例

本システムを用いて，SOR 法と N 体問題の記述を行った．SOR 法は二次元の配列を用い，隣接する断片間でデータを交換し，全要素について残差の集計を行う．N 体問題では一次元の配列を用い，全ての二つの要素のペアについて位置から互いの間に働く力を計算する．

5.1 SOR 法

5.1.1 問題の概要

SOR 法は偏微分方程式を反復法によって解くものである．二次元の配列を，上下左右の値を用いて更新する．端の要素は境界条件に基づいて更新する．全要素の更新が終わったら，残差 (更新された値と元の値の差) を計算して全範囲について集計する．集計した値が閾値未満であれば終了し，そうでなければ再度更新を行う．

このプログラムを並列環境で実行するには，配列要素を各プロセッサに分け与えて，各プロセッサが保持する要素を更新する．プロセッサ間の境界に位置する要素では，更新の際に他のプロセッサのデータが必要になるので適宜要求して更新する．更新が終わったら，残差を 1 プロセッサに集めて集計し，閾値と比較してループを続けるか終了するかを判断する．この擬似コードを以下に示す．

```
while(残差 > 閾値) {
  境界のデータを交換
  保持する配列要素を更新
  残差を 1 プロセッサで集計
}
```

5.1.2 記述方法

以下に，このシステムを用いて記述した擬似コードを示す．プログラムは分散集合オブジェクト *SORMatrix* のインスタンスを作り，全インデックス範囲について *calc*() を呼ぶことにより開始される．*SORMatrix* は必ず矩形領域を持つものとし，一プロ

セッサに矩形でない領域を配置する場合は二つ以上の断片に分けるものとする．

SORMatrix の基本になる関数は *calc*() と *sumup*() である．*calc*() は，その回のループで初めて呼ばれた場合，その断片の端のデータを隣接するプロセッサに送信し，断片が持つ配列の値のうち更新できる部分を更新する．その後，引数のデータを用いて境界を更新し，全範囲の更新が終わったら残差をプロセッサ 0 に送信する．

sumup() は引数にインデックス範囲と残差を取り，それまでにその断片に対し送信されてきたインデックス範囲と残差を集計する．受け取るべき残差が全て到着したら閾値との比較を行い，閾値より大きい場合はループを続ける．この場合は，全インデックス範囲に対し *calc*() を呼ぶ．閾値より小さければ，結果を出力させて終了する．

```
class SORMatrix : public da::Fraction {
  void calc(da::IndexSet *is, da::msg *m){
    if(現在のステップ最初の呼び出し){
      calc(隣接するインデックス集合，自分の境界のデータ)
      内部を更新
    }
    引数のデータを用いて周囲を更新
    if(未更新の領域が存在) return;
    残差を集計・次のステップへ
  }
};
```

5.1.3 性　　能

このプログラムを 100 台を用いた環境でプロセッサの増減のもと正しく動作した．また性能評価のため，行列サイズを一辺 20000 に設定して 4 台から 100 台で実行した．測定環境は CPU が Xeon2.4GHz・メモリ 2GByte のマシンが 1Gbps で接続されたクラスターである．また，データの分割は予め二次元的に均等に割り付けを行った．一回ごとに残差の集計のため全プロセッサが同期されるので，一台のプロセッサについて一回のループにかかった時間から GFlops 値を計算した．これを図 3 に示す．また，この際内部の更新にかかった計算時間 (cputime) とループ全体の時間の内訳も測定した．これを図 4 に示す．

GFlops 値は 100 台までスケーラブルな性能を示した．一台当たりに換算すると，4 台では 576MFlops，100 台では 413MFlops となっている．比較のために，同じ計算処理を用いた逐次版では 590MFlops であったので，通信によるオーバーヘッドは 3 割程度に抑えられている．

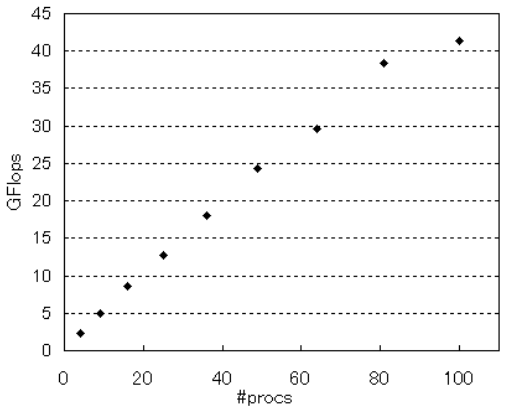

図 3 SOR 法の GFlops 値

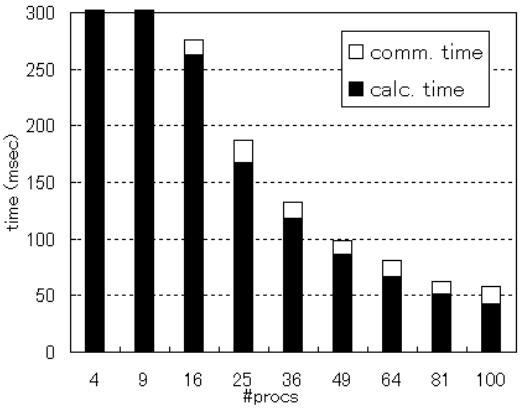

図 4 SOR 法の時間内訳

また，実行中にプロセッサを増加させたときの結果を図 5 に示す．初め 16 台で均等にデータを割り付けて実行していた計算に，44 台のプロセッサを追加した．この時の一ループごとの GFlops 値の変化の様子を図 5 に示す．

参加前は 8.6GFlops だったのが，図で示した部分で 44 台のプロセッサが参加したことで，マイグレーションにより一時的に処理が遅くなっている．その後はルーティングの変更により値は変動しているが，参加後は約 31GFlops で安定し，プロセッサ参加によって 3.6 倍の高速化が図られた．今回の実装では，新しく参加したプロセッサはランダムなインデックスに対し参加要求メッセージを送信し，受信した断片は均等に二分割されて，片方を返す．また，このアプリケーションは全体の性能が最も遅いプロセッサに律速されてしまうため，少ない台数を参加させても性能の向上

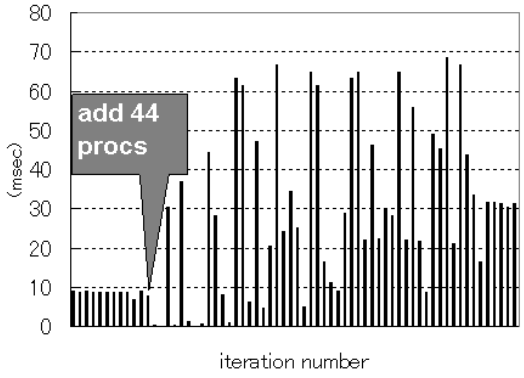

図 5 join 時の GFlops 値変化

は見られなかった．

また，あるプロセッサが参加要求を出してから参加が完了するまでの時間は 7.5 秒で，うちオブジェクトの分割に 2.2 秒・復元に 2.1 秒がかかっている．

5.2 N 体問題

5.2.1 問題の概要

N 体問題は，N 個の物体の間に働く力をシミュレートするものである．全ての物体の組み合わせについて力を計算して，全ての物体について力を計算出来たら速度・位置を更新する．このアルゴリズムは以下のようになる．

```
A[size] : 物体リスト
指定した時間ループ{
  全ての (i,j) について
    A[i] と A[j] の間の力を計算
  時間を進める
}
```

5.2.2 記述方法

このプログラムを本処理系を用いて記述した結果を以下に示す．なお，これは最も簡単なものを示しており，実際に性能測定をしたものは少し異なる．処理は calc() を全領域について呼び出すことにより開始される．calc() は引数に物体の位置・質量のリストを取る．まず calc() がそのステップで初めて呼び出されたときには，その断片が持つ物体のデータを引数に，全てのインデックス集合に対して calc を呼び出す．その後，引数にデータがあれば，そのデータをもとに各物体にかかる力を計算する．その断片が全ての物体のデータを受け取って力を計算したら，時間を進めて位置・速度を更新する．

```
class NBody : public da::Fraction {
  whole_is : 全体のインデックス集合
  own_is   : 保持するインデックス集合
  void calc(da::IndexSet *is, 物体データ){
    if(現在のステップ初の呼び出し){
      calc(whole_is, 保持する物体のデータ)
    }
    引数の物体データを用いて力を計算
    if(更新完了){
      時間を進める
      calc(own_is)
    }
  }
};
```

5.2.3 性能評価

このプログラムを物体数 5000 に設定し，プロセッサ数を 1-49 台の環境で実行し，一回のループにかかった時間から GFlops 値を計算した．

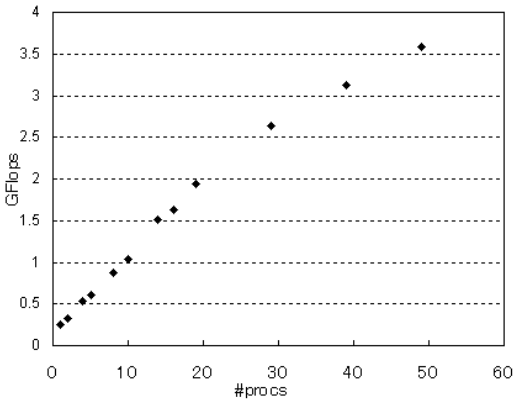

図 6 N 体問題の GFlops 値

GFlops 値は 49 台で頭打ちとなってしまった．一台当たりに換算すると，4 台では 134MFlops，49 台では 73MFlops となっている．これは用いたアルゴリズムにおいて，プロセッサ間の仕事量にばらつきがあるためだと考えている．

6. おわりに

本発表では，分散配列のようなプロセッサ間に分散する集合を扱うプログラムにおいて，オブジェクト指向で記述が行えてかつプロセッサ数の増減に対応する記述モデル「分散集合オブジェクトモデル」を提案し実装した．これによって，プログラマはデータの分散を意識せずにアルゴリズムを記述することができる．プログラマはクラス定義の際に，オブジェクトを分割

する *split*(), 合併する *merge*() などを記述することで，メソッド呼び出し一つで計算への参加・脱退が実現する．またアルゴリズムの記述においては，プログラマはデータとプロセッサの対応を意識する必要がない．

今後の課題としては，プリプロセッサによりメソッドが擬似的に返り値を取れるようにすることが挙げられる．現状の記述ではメソッドの実行を非同期的なものに限定しているので，メソッド呼び出しは返り値を取ることが出来ない．これは，通常の同期的なメソッド呼び出しから機械的に変換できるものだと考えている．このため，プログラマが通常の C/C++に近い文法で記述したプログラムをプリプロセッサによって処理することで，より記述しやすいシステムを構築していきたい．

参 考 文 献

1) The Message Passing Interface(MPI). http://www-unix.mcs.anl.gov/mpi/ .
2) Object Management Group. http://www.omg.org/ .
3) Andrew A. Chien, Vijay Karamcheti, John Plevyak, Xingbin Zbang . Concurrent Aggregates Language Report Version 2.0 http://www-csag.ucsd.edu/papers/csag/external/ca-report.ps (1993)
4) Kenjiro Taura, Toshio Endo, Kenji Kaneda, and Akinori Yonezawa. Phoenix : a Parallel Programming Model for Accommodating Dynamically Joining/Leaving Resources. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2003).
5) Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick: A flexible, optimizing IDL compiler. In Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI), pages 44-56, Las Vegas, NV, June 1997.
6) Charm++. http://charm.cs.uiuc.edu/research/charm/