卒業論文中間報告

マイグレーションを支援する
分散集合オブジェクト

平成16年9月27日提出

指導教官　近山　隆　教授
田浦 健次朗 助教授

電子情報工学科

30388 高橋 慧

目 次

1 はじめに **1**
1.1 背景と目的 . . . 1
1.2 分散集合オブジェクトの概要 . . . 1
1.3 本研究の貢献 . . . 1
1.4 本稿の構成 . . . 2

2 並列プログラム記述のための既存の手法 **2**
2.1 メッセージパッシングモデル . . . 2
2.1.1 概要 . . . 2
2.1.2 プロセッサの増減 . . . 2
2.1.3 記述性と性能 . . . 2
2.1.4 RMA を追加したモデル . . . 3
2.1.5 Phoenix モデル . . . 3
2.2 分散共有メモリモデル . . . 3
2.2.1 概要 . . . 3
2.2.2 プロセッサの増減 . . . 3
2.2.3 記述性と性能 . . . 4
2.3 分散オブジェクトモデル . . . 4
2.3.1 概要 . . . 4
2.3.2 プロセッサの増減への対応 . . . 4
2.3.3 記述性と性能 . . . 4
2.3.4 Concurrent Aggregates . . . 5

3 分散集合オブジェクトの提案 **5**
3.1 対象とする集合 . . . 5
3.2 記述方法 . . . 5
3.2.1 クラス定義 . . . 6
3.2.2 オブジェクト生成 . . . 6
3.2.3 メソッド呼び出し . . . 6
3.2.4 マイグレーション . . . 6
3.3 返り値の取得とマイグレーション . . . 7
3.4 特徴 . . . 7
3.5 実装 . . . 7
3.6 記述例 . . . 8

4 おわりに **10**
4.1 現状 . . . 10
4.2 今後の予定 . . . 10

1　はじめに

1.1　背景と目的

インターネットの普及により、様々な環境の計算機を接続し、一つの大きな計算資源とする、グリッドコンピューティングが脚光を浴びている。プロセッサの性能は飛躍的に向上している一方で、インターネットに接続されているプロセッサは多くの時間アイドル状態である。そこで、これを並列計算機として計算を行うと、安価に莫大な計算資源を手に入れることができる。

一方で、これらのプロセッサはユーザーからの要求があれば本来の処理に戻る必要がある。またユーザーの判断で不意にネットワークから切断されることもあるし、故障も多い。このため、グリッドコンピューティングの普及のためには、参加するプロセッサ数が計算途中に変化することに対応した計算モデルが必要になる。

並列計算プログラムを記述するためのフレームワークは、これまでに各種提案されている。記述の容易さ・柔軟さと性能はトレードオフの関係にあるが、これから並列計算が多くの成果を上げるためには、記述の容易さが求められていると言える。

しかし、プロセッサ数の変化する環境に対応した、書きやすい並列計算記述のフレームワークはまだ発展途上であり、これまでのグリッド環境による大規模な成果も、仕事の依存性が少ない単純な問題に限られている。

例えば微分方程式を解く際、配列の要素をプロセッサ間で分割して持っている例を考える。各プロセッサは、各要素について上下左右の値から要素を更新していく。プロセッサが増減する環境においては、要素とプロセッサの配分は変化する可能性がある。これを最も普及しているメッセージパッシングモデルで記述する場合、メッセージを送る際に通常プロセッサの指定が必要である。このため、要素とデータの関係が変化した際は、メッセージの送信内容や回数を大きく変えなければならない。

1.2　分散集合オブジェクトの概要

本研究では、大きな配列などの集合を扱う際に、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」を提案する。これは、配列のようにインデックスによって識別されるデータを扱う際、データが複数のプロセッサに分散していても、プログラムの記述上は一つの大きなオブジェクトに見えるものである。

既存の分散オブジェクト技術では、逐次と同様なオブジェクトを記述し、リモートにあるオブジェクトのメソッドを呼び出すことができる。

これを基本に、実体は複数のプロセッサに分散していても、外側からは一つに見えるようなオブジェクトを記述し、リモートからメソッドが呼べるようにする。メソッドの呼び出しは、オブジェクト名・メソッド名と「10 番から 20 番」のようなインデックスの集合を指定する。これにより、要素番号と共通集合を持つ全ての断片オブジェクトに対しメソッドが呼ばれる。

断片オブジェクトは分割・結合が可能で、またメソッド一つでマイグレーション (オブジェクトが他のプロセッサに移動すること) が実現される。これにより、プロセッサ数の増減に応じてオブジェクトの要素配置の配分を柔軟に変更することができる。

1.3　本研究の貢献

従来の分散オブジェクト技術では、「オブジェクト全体」というビューを提供すると、全てのメッセージが一つの「全体オブジェクト」に集中し、性能上のボトルネックとなっていた。

本「分散集合オブジェクト」では、オブジェクトを用いるプログラマに対しては「オブジェクト全体」というビューを提供しつつ、実際はメッセージは直接各断片オブジェクトに届くため、このボトルネックが解消されている。これにより、逐次のオブジェクト指向プログラムと同様に記述でき、メッセージパッシングに劣らない速度を実現する。

また断片のマイグレーションにより、プロセッサ数の増減に対応したプログラムを簡単に記述できる。しかも、マイグレーションによってデータと要素の対応が変化しても、無駄なメッセージによって性能が低下しない。

1.4 本稿の構成

本稿では、まず第２章において既存の並列プログラム記述のフレームワークを説明し、その特徴と問題点を明らかにする。次に第３章で本研究の手法を説明し、最後に第４章で現状と今後の計画について述べる。

2 並列プログラム記述のための既存の手法

本章では、既存の並列プログラムを記述するための以下の手法を説明する。

- メッセージパッシング
- 共有メモリ
- 分散オブジェクト

ここでは各モデルについて、概要・プロセッサの増減への対応・記述性と性能を説明し、問題点について明らかにする。なお、一般に性能と記述性はトレードオフの関係にあるため、まとめて述べることにする。

2.1 メッセージパッシングモデル

2.1.1 概要

メッセージパッシングモデルは、最も基本的で、かつ多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。

プロセッサがアクセスできるメモリ空間は自分のメモリだけ、呼び出せるメソッド (サブルーチン) は自分のメモリ内にあるオブジェクトのメソッドだけである。ここで、相手プロセッサを指定してメッセージを送受信できる send() 関数と receive() 関数が用意されている。send() 関数は、相手プロセッサを指定してメッセージを送信する。送られたメッセージは、相手プロセッサの receive キューに入る。receive() 関数は、キューからメッセージを取り、これを返す。なお、キューを用意したことにより、同時に複数のプロセッサが送信を行っても、受信側が receive() 関数を必要な回数呼べば、メッセージは失われない。

2.1.2 プロセッサの増減

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。プロセッサ台数が変化する環境においては、相手プロセッサを指定する方法が問題になる。計算中にプロセッサが増減した場合、プロセッサが保持するデータも当然変化するが、その場合、新しいデータとプロセッサのマッピングを動的に生成するのは大変である。さらに、増減したプロセッサとの仕事のやり取りも簡単ではない。

2.1.3 記述性と性能

記述は一般に簡単ではない。普通の逐次プログラムには通信という概念が無いため、そのまま並列のプログラムに置き替えるわけにはいかない。

特に問題になるのが、メッセージの受信に関する記述である。プログラム中では様々な種類の通信が発生するが、それらの受信の処理は別の関数に記述するのが自然である。しかし、予め受信するメッセージの順番が決まっていない限り、受信側ではメッセージを受信するまでどの種類のメッセージか判断できない。このため、メッセージの振り分けの処理をプログラムごとに記述する必要が生じる。

一方で、メッセージパッシング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。

2.1.4 RMA を追加したモデル

MPI2[1] では、メッセージパッシングに加えてリモートメモリアクセス (RMA) の機能を備えることで、記述効率を向上させている。リモートメモリアクセスとは、明示的にメッセージの送受信を行わなくても、あるプロセッサが関数を呼び出すことで別のプロセッサのメモリを読み書きできる技術のことである。

MPI2 での実装では、予め一定の仮想のメモリ番地を宣言し、この領域をリモートのメモリに対応付ける。そして、mpi_put() / mpi_get() 関数を通じてこの番地にアクセスすると、予めリモートのプロセッサに待機していた RMA 用のスレッドと通信が行われ、アクセスが可能になる。

基本はメッセージパッシングであり、記述はメッセージパッシングを併用して記述できるため、純粋な分散共有メモリに比べて性能の向上が図りやすい。また、データ要求のメッセージの送受信が不要になり、純粋なメッセージパッシングに比べてメッセージの受信に関する処理が単純化できる。

一方で、メッセージの送受信にはプロセッサ番号を指定するため、プロセッサプロセッサの増減への対応は困難なままである。RMA についても、前もって宣言した番地にしか使えないので、動的にデータとプロセッサの対応が変わることは考慮されていない。

2.1.5 Phoenix モデル

Phoenix モデル [4] では、メッセージパッシングを基本としつつ、メッセージの送信先のプロセッサの指定方法として、番号の集合を用いることにより、プロセッサの増減によらず確実にメッセージの送受信が行えるようになっている。

計算前に決まった番号の集合を考え、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、メッセージが失われることは無い。

このモデルでは、プロセッサの増減に対応できる。また、通信の際に特定の親を持たないので、ルーティングに関するボトルネックは無いが、記述効率は従来のメッセージパッシングから改善されていない。

なお、本研究でのオブジェクトへのメッセージの送受信は、この Phoenix ライブラリを用いて行う。このため、プロセッサ数が増減しても問題なくメッセージの送受信が行える。

2.2 分散共有メモリモデル

2.2.1 概要

分散共有メモリとは、プロセッサがリモートのメモリに対し、あたかもローカルのメモリを扱うように書き込みや読み出しが出来るシステムである。内部構造的には、リモートに相当するメモリ番地にアクセスすると、相当するメッセージがそのデータを持っているプロセッサのところに送信される。各プロセッサではデータのリクエストを受けるスレッドが待機していて、メッセージを受け取ると、要求されたデータを返信する。この仕組みをシステムが行うため、プログラムを記述する上では、他のプロセッサとのデータのやり取りは単なるメモリアクセスになる。

これは専用並列計算機において多く用いられてきた手法だが、グリッド上においては、専用計算機に比べて通信がはるかに遅く、一つのメモリについてのアクセスを何度も繰り返すと効率が悪い。このため、実用的にはページキャッシュを併用するなど、アクセスを高速にする工夫が必要である。

2.2.2 プロセッサの増減

分散共有メモリは、プロセッサ増減への対応が比較的容易なモデルだと考えられる。仮想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、実行時に通信が必要か判断できる。ただし、データの配分の変更と共に仕事の配分を変更するための記述が必要である。

2.2.3 記述性と性能

逐次プログラムとは同時に複数のプロセスが走ることだけが異なるようになり、記述が容易になる。異なる種類のデータの共有は、メモリ番地を使い分けることで分離して記述できるので、内容ごとに関数ほ完全に分離できる。

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セッサにこのモデルでプログラムを書くと、小さな領域へのアクセスが多くの回数発生する。プロセッサ間の通信が高速な並列計算機であればこれは問題にならないが、グリッド環境においては通信路は通常のメモリアクセスに比べ非常に低速である。このような環境で小さなメッセージが頻繁にやり取りされると、メッセージの受信を待ってブロックする時間が長くなり、速度が低下してしまう。ページキャッシュを用いた場合でも、ランダムなアクセスでは事情は同じである。このため、実用的な速さのプログラムを書くには、どのプロセッサがどのメモリ領域を持つのかを意識する必要がある。

2.3 分散オブジェクトモデル

2.3.1 概要

記述効率の向上のためには、オブジェクト指向の考え方を導入する方法がある。これは、逐次のオブジェクト指向プログラミングを基本に、リモートのオブジェクトについて、ローカルのオブジェクトと同じようにメソッド呼び出し (remote method invocation：RMI と呼ぶ) を行えるようにするものである。

呼び出されたオブジェクトを持つプロセッサでは新たにスレッドが立ち上がり、その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。分散オブジェクト記述の規格としては、CORBA[2] がアプリケーションサーバーの接続など、並列計算以外の用途では成功を収めている。

2.3.2 プロセッサの増減への対応

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。

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、プロセッサ数の増減に対応した性能の良いプログラムは記述できない。これを以下に示す。

大きなデータ集合をこのフレームワークで記述する場合は、各プロセッサが部分オブジェクトを持つことになる。ここで、オブジェクトを用いる際、各断片に直接アクセスするか、断片オブジェクトとデータのマッピングを保持する「全体オブジェクト」を作り、これを通じてアクセスするか、二通りのスタイルが考えられる。

直接断片オブジェクトにアクセスする場合は、計算を通じてオブジェクト数は基本的に一定である必要がある。このためプロセッサの増減に対応するには、プロセッサ数に比べ十分に多い、小さなオブジェクトを用意する必要がある。その場合、一回のメソッド呼び出しで可能な処理が少なくなり、メソッド呼び出しが増えて性能が低下する。またプログラマは、どの断片オブジェクトがどのデータを持つのかを把握する必要が生じ、逐次プログラムに比べて記述は煩雑になる。

全体オブジェクトを作る場合は、プログラマにとってはあたかも全要素が全体オブジェクトに存在するように見えるため、記述は楽である。また、プロセッサの増減に対しては、断片オブジェクトを適宜分割したり、統合させればよい。データと断片オブジェクトの対応は、全体オブジェクトが保持する。しかし、このモデルでは全てのメソッド呼び出しが一度全体オブジェクトを経由するため、ここがボトルネックとなる。

つまり、プロセッサの増減には基本的には対応出来るが、大きなデータを扱うオブジェクトでは問題がある。

2.3.3 記述性と性能

記述は逐次プログラムに近く、容易である。また、記述力も高い。

ユーザーが記述する必要があるのは、個々のメソッドとメソッドを呼び出すプログラムだけになる。これは、逐次のオブジェクト指向プログラムとほぼ同じである。呼び出すメソッドの内容はデータの読み書きに留まらないので、記述能力は分散共有メモリよりも高い。

またこのシステムの特徴として、自然な形でデータとタスクの関連性が保たれるという点がある。予めオブジェクトをプロセッサ間に分配しておけば、明示的に各プロセッサでプロセスを立ち上げなくても、それぞれのオブジェクトのメソッドが呼ばれるたびに仕事が自動的に分配される。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。

一方で先にも述べたとおり、プロセッサの増減に対応させるには、記述性や性能が損なわれる。

2.3.4 Concurrent Aggregates

Concurrent Aggregates[3] とは、分散オブジェクト指向においてプロセッサ間に分散するオブジェクトを記述した場合、全体オブジェクトがボトルネックになるのを解消するために考えられたフレームワークである。大きなデータを複数のオブジェクトで分割して持つのは分散オブジェクトと同一だが、ここで全体オブジェクトは存在しない。その代わり、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。

このシステムの長所は、全体オブジェクトを経由することなく全体へのメソッド呼び出しが出来る点にある。このため、全体オブジェクトが全ての断片オブジェクトの場所や配分を把握する必要は無くなる。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。

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 分散集合オブジェクトの提案

本研究では、従来の分散オブジェクト技術を基本としつつ、オブジェクトの識別に整数集合を用いることで、プロセッサの増減に対応し、しかも性能を低下させない「分散集合オブジェクト」モデルを提案する。

3.1 対象とする集合

本フレームワークにおいて扱う集合は、インデックスによって識別される集合とする。これはインデックスと要素が一対一に対応する配列だけに限らず、一対多に対応する集合や分散ハッシュも対象となる。ここで、本システムにおいて作られるオブジェクトは、外部から見ると大きな集合全体に見える。しかし、その実体は各プロセッサに分散した、部分集合を持つ断片オブジェクトの集まりである。それぞれの断片は、保持するインデックス集合によって識別される。

このオブジェクトが分割されるときには、インデックス集合を分割し、それに応じた要素を分け与える。オブジェクトが合併される際には、インデックス集合・要素集合を共に合併させる。

3.2 記述方法

本フレームワークでは、プログラマは断片クラスを記述し、用いる際は仮想的な全体オブジェクトへの参照からアクセスする。全体オブジェクトへの参照はコンストラクタによって得られるほか、断片クラス内部からは whole キーワードによってアクセスできる。

システムの基本となるのは、RMI とマイグレーションである。RMI については、通常のクラス定義を記述し、インデックス集合を引数に取るメソッドを記述すると、RMI 関連のメッセージを処理するコードが自動生成され

る。マイグレーションについては、split()・merge()・serialize() のメソッドを記述すると、join() 関数と leave() 関数が自動生成される。この関数により、そのプロセッサ内に断片オブジェクトを受け入れたり、オブジェクトを他のプロセッサに退避させたり出来るようになる。

以下、これらを詳しく説明する。

3.2.1 クラス定義

プログラマは、断片クラスについて持つデータとメソッドを記述する。この断片クラスのオブジェクトは、自分が保持するインデックス集合を保持する。

コンストラクタは、自分が担当するインデックス集合を引数に取る。このインデックスを元に、プログラマはデータ用の領域を確保するなどの処理を記述する。

一般のメソッドはインデックスの集合を引数として取り、これにより操作の対象となる要素を指定する。メソッドには自分の持つ部分集合に対しての操作を記述する。他の断片のメソッドを呼び出す際は、仮想的な全体オブジェクトへのリファレンスを用いて呼び出す。

また、マイグレートに対応するため、オブジェクトを二つに分割する split()、二つのオブジェクトを併合する merge()、オブジェクトのデータをメッセージとして送信可能なバイト列にする serialize() の三つのメソッドを記述する。

断片クラスの記述後、プリプロセッサにより、コードの補完・及び全体クラスの作成が行われる。

3.2.2 オブジェクト生成

オブジェクトを生成するには、あるプロセッサでコンストラクタを呼ぶ。これにより、各プロセッサで断片オブジェクトが自動生成される。また、呼び出したプロセッサでは仮想的な全体オブジェクトへの参照が返る。なお、プロセッサに対する配分方法は、コンストラクタで指定できる。

3.2.3 メソッド呼び出し

メソッドの呼び出しは、「全体オブジェクト」に対し、メソッド名と操作の対象となるインデックスの集合を指定する。すると、インデックス集合と重なりを持つインデックス集合を持つ全ての断片オブジェクトで、インデックス集合を引数としてメソッドが呼び出される。例えば、[0-10)、[10-20) のインデックス集合を持つオブジェクトに対し、[5-15) を指定してメソッドを呼び出すと、二つのプロセッサ 1 で [5-10)、[10-15) を引数としてメソッドが呼び出される。(カッコは整数の範囲を表す)

「全体オブジェクト」は、まずクラス定義の際にクラス内から参照を得ることが出来る。また、オブジェクトを生成したプロセッサでは、生成時に参照が返る。断片オブジェクトに外部から直接する手段は、基本的に設けない。

なお返り値については色々検討すべき点があるので、まずは返り値をサポートしない形でシステムを構築する。これについては後述する。

3.2.4 マイグレーション

マイグレーションは、join() 関数と leave() 関数により行われる。計算に参加するプロセッサは join() 関数を、計算から離れるプロセッサは leave() 関数を呼ぶ。join が呼ばれた場合は、近くのプロセッサの持つ断片オブジェクトが split() により分割され、serialize() によりバイト列になって該当プロセッサにマイグレートする。leave() の場合は、そのプロセッサのオブジェクト断片が他のプロセッサにマイグレートし、そこにある断片に併合される。

3.3 返り値の取得とマイグレーション

先述のとおり、本システムは当面は返り値をサポートしない。この理由を以下に述べる。

あるオブジェクト A のメソッド m_1 が、実行中に他のメソッド m_2 を呼び出して、その返り値を待っている状態を考える。この間に A をマイグレートする場合は、A は m_1 を実行中のスレッドと共に移送するのが自然である。この場合、m_1 を実行中のスレッドは、m_2 の返事を受け取って処理を続けることができる。

ここで、通常の C/C++のシステムでは、実行中のスレッドを他のプロセッサに移送するのは困難である。このため、オブジェクトをマイグレートするには、オブジェクト内に実行中のスレッドがあってはならない。しかし、通常の返り値をサポートするシステムでは、オブジェクト内には返り値を待って待機するスレッドがほぼ常に存在し、オブジェクトは永久にマイグレートできない。

そこで本フレームワークでは、まずメソッド途中での処理の中断を禁止する。そのような処理は、中断後の処理を別のメソッドに分けて記述させる。また、メソッド呼び出しは非同期的に行われる。つまり、メソッドを呼び出したスレッドは、そのメソッドの終了を待たずに次の処理に移る。

この処理系を用いてアプリケーションを記述するには、返り値オブジェクトを用いる。例えば、あるオブジェクト A のメソッド m_1 が m_2 を呼び出し、その返り値を取得して処理を進めるプログラムは以下のように記述する。

- まずプログラマは m_1 を m_2 の呼び出し部分で m_{1a} と m_{1b} に分割する
- m_{1a} は、m_2 の引数のインデックス集合と「次は m_{1b} を実行する」という情報を持たせた返り値オブジェクト R を生成する
- m_{1a} は R への参照を引数に渡して m_2 を呼び出し、終了する。m_2 は複数の断片オブジェクトで呼び出される
- m_2 は処理を行い、結果を引数として R の特定のメソッド (m_r とする) を呼び出し終了する
- m_r は、m_2 が呼び出された全ての断片オブジェクトからの返り値が取得できたら、この結果を引数に m_{1b} を呼び出す

最終的にはこのシステムを元にして、通常の C++に近い文法で、擬似的に返り値を取れるようなコードを記述すると、上に述べた返り値を持たないシステムに変換される処理系を提供する。

3.4 特徴

本フレームワークの特徴は、インデックスによって識別できる集合を用いる並列プログラムを簡単に記述でき、プロセッサ数の増減に対応できることである。

記述に関しては、一般的な分散オブジェクトモデルに近い。このため、オブジェクト指向の記述ができ、意識せずにデータとタスクの関連性が保たれる。一方で、メソッドの呼び出しはインデックス集合によって行われるが、この際呼び出しのメッセージは一度ある「全体オブジェクト」に届くのではなく、直接各断片オブジェクトに届く。このため、メソッド呼び出しの際に全体オブジェクトがボトルネックにならない。

Concurrent Aggregates と比較すると、全体オブジェクトを持たない点は同様だが、インデックスから直接目的とするオブジェクトにアクセスできる点が性能的に優れている。

マイグレーションについても、オブジェクト自体を分割・併合でき、しかもメソッドの呼び出しは分割の単位を意識せずに行えるので、常にプロセッサ数に適合したデータ配分で計算を行うことが出来る。

3.5 実装

システムの実装には、C++言語及び Phoenix ライブラリを用いる。Phoenix ライブラリでは各プロセッサがバーチャルノードと呼ばれる整数集合を持ち、メッセージはバーチャルノードに対し送信される。本システムではインデックス一つに Phoenix のバーチャルノード一つを対応付けることで、メッセージを該当するインデックスを持つ断片オブジェクトに向けて送信できる。

3.6 記述例

偏微分方程式の解法である、共役勾配法 (SOR) を記述した擬似コードの一部を示す。この例では、大きな二次元のデータに対し、一要素ずつ値を更新する。値の更新には、その要素の上下左右のデータが必要である。全ての要素について更新が終わると、また初めの要素から更新を行う。値の変化が収束したら、処理を終了する。

これを並列処理する際には、配列データを分割し、各プロセッサが自分のデータを更新していくのが自然である。各プロセッサは、値の計算に自分のデータと、自分と他のプロセッサの境界のデータを必要としている。

この例では、まず断片オブジェクトとして SORFraction を記述し、コンストラクタの SORFraction()、計算を行う calc()、自分の持つデータの境界のインデックス集合を取得する get_border_indexes()、インデックス集合に対応する要素を返す get_data() のメソッドを記述する。

```
/* 断片オブジェクトの記述 */
class SORFraction {
  /* 保持するインデックス集合 */
  index_set *assigned_set;
  /* 保持するデータ */
  data_set *data;

  /* コンストラクタ。引数は保持するインデックス集合 */
  SORFraction(index_set *_assigned_set){
    assigned_set = _assigned_set;
    /* データの領域確保などを記述 */
    data = new data_set(assigned_set);
  }

  /* 処理の中心となるメソッド */
  void calc(index_set *request_set){
    for(int t = 0; t < t_max; t++){
      /* 必要になる端のインデックスを計算して、index_set に格納 */
      index_set *border_indexes = get_border_indexes();
      data_set *bound_data;

      /* RMI であることの指定 */
      remote{
       /* get_bound() メソッドを呼び出す。これもこの断片オブジェクト中で定義する。
          whole は全体オブジェクトへの参照を表すキーワード */
        bound_data = whole->get_data(border_indexes, ro);
      }
      /* 自分のデータ (data) と端のデータ (bound_data) から計算を行う */
      ....
      /* プロセッサ間で同期を取る */
      ....
    }
  }

  /* その他のメソッドを記述 */
  ...
```

```
};
```

これがクラス定義において記述する内容である。

そして、計算を命じる一番元のプロセッサでは、以下のようにメインルーチンを記述する。sor-¿calc() の呼び出しにより、全プロセッサで計算が開始される。

```
void main(void){
  /* システムの初期化 */
  init();

  /*  まず作成するインデックス集合を指定する。ここでは [0,1000) の範囲で作成する */
  IndexSet *whole_indexes = new IndexSet(0, 1000)
  /* コンストラクタ。"SQUARE" オプションにより、
     各プロセッサに二次元的な分割で断片オブジェクトが生成される */
  SOR *sor = new SOR(whole_indexes, SQUARE);

  /* RMI であることの指定 */
  remote{
    /* 全体オブジェクトからメソッドを呼ぶ。引数は [0,1000) の範囲を指定 */
    sor->calc(new IndexSet(0, 1000));
  }

  /* システムの後処理 */
  finalize();
}
```

また、その他のプロセッサでは、次のようなメインルーチンを記述する。

なおこれらのプロセッサでは、join() によって断片オブジェクトがマイグレートして、裏でメソッド呼び出しが行われ計算に参加する。よって、このメインルーチンに計算に関する処理は記述されていない。

```
void main(void){
  /* システムの初期化 */
  init();
  /* 計算に参加し、オブジェクト断片を受け取る */
  join();
  /* ユーザ又はシステムからの要求を待つ。
     ユーザが計算から抜けることを要求した場合、LEAVE_REQUEST が返るものとする。
     また、全ての計算が完了した場合も、別の値が返るものとする。
     どちらかのリクエストが来るまで、ここでブロックする */
  request req = get_user_request();

  /* 計算から抜けたい場合は、migrate() を呼ぶ。
     これによりプロセッサ内のオブジェクトが退避する */
  if(req == LEAVE_REQUEST){
    migrate();
  }
```

```
  /* システムの後処理 */
  finalize();
}
```

これにより、マイグレートに対応したアプリケーションが記述出来る。

4 おわりに

4.1 現状

現在、リモートからメソッドが呼び出し可能な分散配列オブジェクトを作成している。これに適当なメソッドを追加することで、マイグレート可能な FFT や SOR のプログラムが記述できる。

4.2 今後の予定

今後は、実際のアプリケーション記述において必要になる機能を考えつつ、システムについての実装を進めていく。

まずは先述の分散配列オブジェクトのテンプレートを完成させ、アプリケーションを記述する。その後、一般的なクラス記述に用いるテンプレートの形にし、またメッセージの受信スレッド、メッセージ定義などを自動生成できるような書式を考え、システムを構築する。

先述したように返り値については当面サポートせず、上記のアプリケーションも返り値を用いない形で記述する。その後、プログラムの変換により返り値をサポートするような書式・システムを考えていきたい。

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)

[4] Kenjiro Taura. Phoenix: A Parallel Programming Platform Supporting Dynamically Joining/Leaving Resources (In Japanese). In IPSJ SIG Notes(HPC-87-24), July 2001.