Distributed Aggregate with Migration
マイグレーションを支援する分散集合オブジェクト

近山・田浦研究室　電子情報工学科 30388 高橋 慧

1　はじめに

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。現在までにも専用並列計算機に匹敵する多くの成果が上がっているが、さらなら普及のためにいくつかの要請がある。

一つは、プロセッサ数の増減に対応することである。現状のグリッド環境は研究所のクラスタが主流だが、複数のクラスタを接続すればより大きな問題を解くことが出来る。しかし、各クラスタを占有できる時間は限られている場合が多く、長期にわたる計算では動的に展開や撤退を行う必要がある。また、耐故障性の確保や、一般のパソコンを計算に参加させる PC グリッドのためにも、プロセッサ数変化への対応が必要である。

もう一つは、記述を簡単に行いたいという要請である。現状では多くの並列プログラムは通信 (メッセージパッシング) を用いて記述されているが、これは下層の処理に近すぎて、簡潔にアルゴリズムを表現できない場合が多い。これに対し、逐次プログラムで実績のあるオブジェクト指向を用いれば、通信を隠蔽し、簡単に複雑な処理を記述できる。

しかし現状では、オブジェクト指向で並列プログラムを記述できる分散オブジェクト技術は存在するものの、プロセッサ数の増減に対応したプログラムの記述は依然容易ではない。簡単に記述できるモデルもあるが、この場合はメッセージが特定プロセッサに集中し、性能の低下を招いてしまう。

本研究では、動的なプロセッサ数の増減に対応し、オブジェクト指向で記述が容易で、メッセージの集中を防ぎ性能が良い記述モデルとして、「分散集合オブジェクトモデル」を提案、実装した。また、この分散集合オブジェクトモデルの下でアプリケーションを記述し、動作の確認と性能評価を行った。

2　既存のプログラミングモデル

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、現在最も広く使われている並列プログラム記述のフレームワークである。代表的な実装としては MPI[1] が挙げられる。これは相手プロセッサを指定して `send()/receive()` 関数を用いてメッセージを送受信するもので、下層の通信 API に近いものになっている。このためは無駄のないプログラムが書ける一方、記述が手続き的で簡単ではない。また、通常のメッセージパッシングではプロセッサを番号で識別するため、プロセッサ数が増減した場合、メッセージの送信先を動的に変えなければならない。

2.1.1　Phoenix モデル

Phoenix モデル [2] はメッセージパッシングの一種だが、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。

2.2　分散共有メモリ

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、ローカルのメモリと同じ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に送信される。各プロセッサでは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メモリアクセスと同等に扱える。

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、異なるプロセッサ数での実行に対応できる。

しかし、一般にプログラマは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記述するため、リモートのデータへのアクセスが頻繁に発生する。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り、小さなメッセージが頻繁にやり取りされると性能が低下してしまう。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が、その場合は記述効率が低下してしまう。

2.3 分散オブジェクトモデル

複数のプロセッサで使えるオブジェクト指向の記述フレームワークとしては、分散オブジェクトモデルが提案されている。並列計算ではあまり用いられていないが、CORBA[3] がアプリケーションサーバーの接続などで成功を収めている。

プログラマはまずクラスを定義する。このクラスのインスタンスをあるプロセッサ上で作り、他のプロセッサにこれへの参照を持たせる。すると、この参照からリモートのオブジェクトのメソッドが呼び出せる。これを RMI (remote method invocation) と呼ぶ。このメソッドの処理は、呼び出されたオブジェクトを持つプロセッサが行う。

プロセッサの増減には、マイグレーションで対応する。これは、あるプロセッサで生成されたオブジェクトを、他のプロセッサに移動する技術である。これにより、計算に参加したプロセッサにオブジェクトを移動したり、脱退するプロセッサが保持しているオブジェクトを退避させたりできる。

しかし、このモデルでマイグレーションやメソッド呼び出しが可能なのは一個単位のオブジェクトに対してであり、複数のプロセッサにまたがったデータを扱う際には、プログラマが多くを記述しなければならない。このため、一般にプロセッサ数の増減に対応するプログラムを容易に記述できるとは言えない。

3 「分散集合オブジェクト」の提案

本研究では、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。

3.1 対象とする分散集合

本研究においては、インデックスで位置を識別できる集合を対象とする。これには、配列・ハッシュ表などが含まれ、大規模なプログラムにおいては頻繁に用いられる。並列プログラムにおいては、各プロセッサが集合の一部を保持し、離れたプロセッサが持つデータにアクセスするにはそのプロセッサに要求を送信する。

このようなデータ構造は、逐次プログラムにおいてはインデックスから簡単にメモリアドレスを計算できる。また、台数が実行前に決まっている並列プログラムにおいても、ある要素を持つプロセッサを簡単に計算できることから、インデックスによるアクセスが高速であるという特徴を持つ。

しかし、実行時に動的にプロセッサ数が変化する環境においては、データとプロセッサの対応は動的に変化する。このため、インデックスから位置を簡単に計算することはできず、インデックスと要素の対応表を保持し、必要に応じて更新する必要が生じる。この対応表の存在が、プロセッサ数増減に対応したプログラムの記述を難しくしている。以後、この対応表をルーティング表と呼ぶ。

3.2 記述モデル

分散集合オブジェクトモデルでは、オブジェクトはプロセッサの中で完結するものではなく、複数のプロセッサに断片として存在することを前提としている。断片オブジェクトはインデックス範囲と、それに対応するデータを保持している。プログラマはこの断片オブジェクトを定義し、その断片が保持する範囲のデータに対する操作をメソッドとして記述する。

次に、プログラマはこのオブジェクトを用いる関数を記述する。外部からオブジェクトにアクセスするには、メソッド呼び出しを用いる。ここで、メソッド呼び出しの際は、個々の断片に対してではなく「全体」に対して行える。メソッドは引数にインデックスの集合を取り、これによってこの引数の集合と重なりを持つ断片でメソッドが呼び出される。これを図 1 に示す。

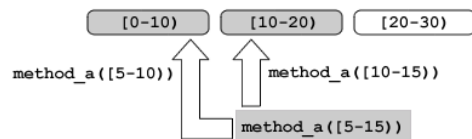

図 1: メソッド呼び出し

プロセッサ増減には、断片が分割・併合することで対応する。プログラマがクラス定義において、予め断片オブジェクトの分割・併合を記述することで、プロセッサへの断片割り当てや、断片の退避がメソッド呼び出し一つで実現される。

3.3 記述例

ここでは、分散配列の指定された要素をインクリメントするメソッドを示す。なお、一部の文法は簡略化のために実際のシステムとは異なっている。

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

int main(void){
…
  WholeArray->increment([100-200));
```

まずメソッド定義では、配列の断片を定義する。data には全体配列に対し、一部の要素が保持されているとする。ここで、メソッド `increment()` はインデックス範囲 `is` を引数に

取る。メソッド内では、この `is` 内の各インデックスを `index` に代入し、`data` の中の相当する要素をインクリメントしている。なお、`is` にオブジェクトが保持しないインデックスが含まれることはない。

これを用いる下のコードでは、配列全体に対し [100-200) という範囲を指定して `increment()` を呼び出している。これにより、この範囲と重なりを持つ各断片で `increment()` が実行される。

3.4 システムの実装

3.4.1 ルーティング表

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 (ルーティング表) を動的に更新する必要があると先に述べた。このルーティング表を比較的容易に実装できる方法としては、一プロセッサ (オブジェクト) が表を集中管理し、全てのアクセスはこのプロセッサを経由して行うものが挙げられる。プロセッサの増減による断片の分割・併合は全てこのルーティング表を持つプロセッサが管理する。

この方法は記述が簡単だが、どのようなメソッドを呼ぶ際にもルーティング表を持つプロセッサにメッセージが送信されるため、ここが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ていて、メッセージは各断片に直接届けられる。このルーティング表の実装には Phoenix ライブラリを利用している。

メソッド呼び出しはインデックス範囲を指定して行われる。ここで、この呼び出しが各断片に中継される基本的な仕組みは以下のとおりである。呼び出しのメッセージはまずインデックス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が、これらにいての詳細は本編 3.5.2 を参照願いたい。

3.4.2 返り値とマイグレーション

今回の実装では、本システムは返り値をサポートせず、プログラマは返り値を持つメソッドを分割して記述する必要がある。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、オブジェクトを移送するには、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移送できない。

そこで本システムでは、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。

先の例では、メソッド m_1 は m_2 の呼び出し部分で m_{1a} と m_{1b} に分割される。m_{1a} は m_2 を呼び出し、直ちに終了する。そして、m_2 は返り値を返す代わりに、自分の保持するインデックス集合を引数に m_{1b} を呼び出す。この際、本来返り値として渡される値は、m_{1b} を呼び出す際の引数として与えられる。この様子を図 2 に示す。

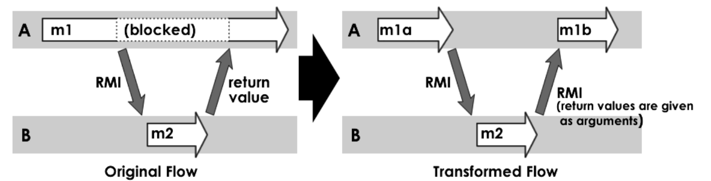

図 2: 返り値を挟んでのメソッドの分割

もしも m_2 の実行中に A がマイグレートした場合でも、m_{1b} は元々の A が保持するインデックス集合に対して行われるので、返り値は正しく移送先のオブジェクトに届けられる。これを図 3 に示す。

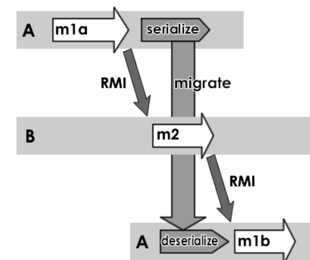

図 3: マイグレーション

将来的にはこのシステムを元にして、擬似的に返り値をサポートする処理系を提供したい。これは、プログラマが返り値を持つメソッドを記述すると、システムがこれを複数の返り値を持たないメソッドに自動変換するものである。

3.5 特徴

分散集合オブジェクトの特徴は、プロセッサ数の増減に対応した並列プログラムを簡単に記述でき、メッセージの集中による性能低下が起こらない点である。記述はオブジェクト指向であり、自動的にデータとタスクの関連性が保たれる。また、要素と断片オブジェクトの対応を各プロセッサが分散保持しているため、メソッド呼び出しは特定のプロセッサを経由するのではなく、直接各断片オブジェクトに届く。このため、メッセージが一プロセッサに集中しない。

プロセッサが増減した場合は、断片オブジェクトを分割・併合することによって対応する。この際、メソッドの呼び出し範囲の指定はインデックス集合によって行われるので、処理の途中にマイグレーションが発生しても、処理を正しく続けることができる。

4 アプリケーションの記述例

4.1 アルゴリズム

本システムを用いて、偏微分方程式の数値解法のプログラムを記述した。二次元の配列をプロセッサが分割保持していて、各要素の上下左右の値を用いて更新していく。ここで、他のプロセッサとの境界部分では、隣の断片とデータを交換する必要がある。交換後各プロセッサは値を更新し、要素が変化した値 (残差) を足し合わせる。全ての要素に渡ってこの変化値を合計し、これが閾値よりも小さければ終了し、大きければ更新を繰り返す。

今回は、端のデータを交換する部分を `send_border()` と `sendback_border()` の二つのメソッドを用いて記述した。前者は、プロセッサの自分の端のデータを引数に、右及び下に隣接しているインデックス範囲に対し `sendback_border()` を呼び出す。後者は、引数のデータを保存した後、送られてきたデータのインデックス範囲に対し、自分の端のデータを引数として次の関数を呼び出す。これにより、`send_border()` を呼んだ断片は、擬似的な「返り値」として隣接するデータを取得できる。

ここで、`send_border()` を呼んだ断片が、隣接する断片の `sendback_border()` の結果を待たずに分割されてしまった例を図 4 に示す。断片が分割されても、`sendback_border()` からの「返り値」は元のインデックス範囲に対し送信されるので、分割された断片双方が返り値を受け取ることができる。

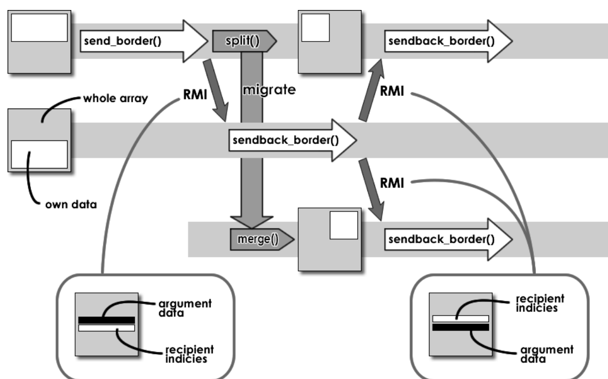

図 4: 偏微分方程式でのマイグレーション例

4.2 性能評価

本プログラムはプロセッサ 81 台の環境で正しく動作し、またプロセッサ 64 台の連続 join に成功した。ここで、全体の配列の一辺を 10000, 20000, 30000、またプロセッサ数を 1-81 台の間で変化させ、一回のループにかかる時間から MFlops 値を計算した。この結果を図 5 に示す。

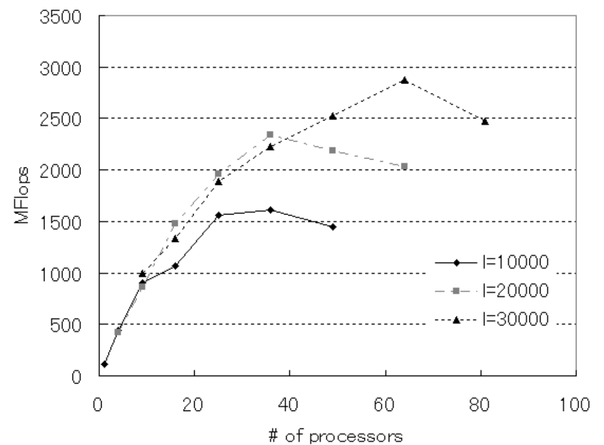

図 5: MFlops 値

結果はプロセッサ数の増加に対し、頭打ちとなる傾向が得られた。これは、残差の合計の部分で特定のプロセッサに処理が集中するような記述を行ったためと推測している。

5 まとめと今後の課題

本研究では、プロセッサ間に分散したデータを取り扱う「分散集合オブジェクト」を提案した。これを用いると、プロセッサ間にまたがる配列のような集合を扱う際、プロセッサ数の増減に対応したプログラムをオブジェクト指向で容易に記述できる。

我々はこの分散集合オブジェクトを記述できるライブラリを実装した。このライブラリを用いて偏微分方程式の数値解法を行うアプリケーションを作成し、正しくマイグレーションが行われることを確認した。

ただ現状のシステムは、メソッドが返り値を取れないという制限のため、必ずしも記述しやすいものにはなっていない。これについては今後の課題とする。

参考文献

[1] The Message Passing Interface(MPI). http://www-unix.mcs.anl.gov/mpi/ .

[2] 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).

[3] Object Management Group. http://www.omg.org/ .