JAERI-Data/Code
97-012

オブジェクト指向による数値流体計算の並列化

1997年3月

太田高志

日本原子力研究所
Japan Atomic Energy Research Institute

本レポートは，日本原子力研究所が不定期に公刊している研究報告書です。

入手の問合わせは，日本原子力研究所研究情報部研究情報課（〒319-11 茨城県那珂郡東海村）あて，お申し越しください。なお，このほかに財団法人原子力弘済会資料センター（〒319-11 茨城県那珂郡東海村日本原子力研究所内）で複写による実費頒布をおこなっております。

This report is issued irregularly.

Inquiries about availability of the reports should be addressed to Research Information Division, Department of Intellectual Resources, Japan Atomic Energy Research Institute, Tokai-mura, Naka-gun, Ibaraki-ken, 319-11, Japan.

© Japan Atomic Energy Research Institute, 1997

編集兼発行　日本原子力研究所
印　　刷　いばらき印刷㈱

JAERI-Data/Code 97-012

オブジェクト指向による数値流体計算の並列化

日本原子力研究所計算科学技術推進センター
太田 高志

(1997年2月18日受理)

オブジェクト指向による科学計算の並列プログラムの設計を提案し、それが有効な手法となることを圧縮性流体のコード例として示す。

既存の科学計算プログラムの並列化は一般に複雑なものとなり、また各並列計算機固有の構成やコンパイラなどに合わせるために互換性の全くないコードとなってしまう傾向がある。ここでは、オブジェクト指向により、並列処理に関する部分とアプリケーション特有の解法に関する部分を分離するような設計により、コードの保守に関して大きく改善すると共に、互換性の高いコードが書けることを示す。また、ここで提案する指針に沿って2次元オイラー方程式の数値解法のプログラムを設計し、SP2上でその性能を評価した。

日本原子力研究所：〒153 東京都目黒区中目黒2-2-54

An Object-Oriented Programming Paradigm for Parallelization of Computational Fluid Dynamics

Takashi OHTA

Center for Promotion of Computational Science and Engineering
Japan Atomic Energy Research Institute
Nakameguro, Meguro-ku, Tokyo-to

(Received February 18, 1997)

We propose an object-oriented programming paradigm for parallelization of scientific computing programs, and show that the approach can be a very useful strategy. Generally, parallelization of scientific programs tends to be complicated and unportable due to the specific requirements of each parallel computer or compiler. In this paper, we show that the object-oriented programming design, which separates the parallel processing parts from the solver of the applications, can achieve the large improvement in the maintenance of the codes, as well as the high portability. We design the program for the two-dimensional Euler equations according to the paradigm, and evaluate the parallel performance on IBM SP2.

Keywords : Object Oriented Programming, Domain Decomposition, Message Passing, CFD(Computational Fluid Dynamics), Parallel Computing

目　次

1. はじめに …… 1
2. 並列プログラムの分析 …… 3
3. 並列処理の隠蔽 …… 4
3. 1 境界条件の再構成 …… 5
3. 2 領域分割の処理 …… 6
3. 3 並列入出力 …… 7
4. 数値流体解析におけるクラス設計 …… 8
5. プログラムの実際 …… 9
5. 1 メインプログラム …… 9
5. 2 データクラスへのアクセス …… 12
5. 3 継承による解法クラスの開発 …… 13
6. 計算例 …… 15
7. 考察及び結論 …… 18
参考文献 …… 20

Contents

1. Introduction …… 1
2. Analysis of Parallel Codes …… 3
3. Hiding of Parallelization Procedures …… 4
3. 1 Recomposition of Boundary Conditions …… 5
3. 2 Process of Domain Decomposition …… 6
3. 3 Parallel I/O …… 7
4. Class Design for Computational Fluid Dynamics …… 8
5. Writing Programs …… 9
5. 1 Main Program …… 9
5. 2 Accessing to the Data Class …… 12
5. 3 Developing a Solver Class by Inheritance …… 13
6. Example …… 15
7. Concluding Remarks …… 18
References …… 20

1 はじめに

並列計算機というものが計算機自体の成熟と、コンパイラやメッセージパッシングライブラリなどの実行環境も整えられてきたことによって、性能的には大規模計算への利用で現実的な候補となっている。しかしながらプログラムの並列化という作業は、並列化コンパイラを使っても、メッセージパッシングライブラリを使ったとしても、並列に実行する部分を明示的に指示しなければならず、現状では簡単なものとは言えない。一般的な並列処理に関する知識と共に、個別の並列環境についての知識も必要となる。加えて、各機種毎の並列コンパイラや、各種のメッセージパッシングライブラリなどの多様性からそれぞれで書かれたものを互換性の無いものとしている。また、数値解法と並列化に関する処理が互いに入れ込んでいるために、それぞれを独立して開発、保守をすることが非常に難しくなってしまう。

いわゆる構造化プログラミングという手法でプログラムを書くのが数値計算プログラムでは主流であると思われる。構造化プログラムでは全体の処理がさらに下位の小さな処理に分けられ、またその中がさらに下位の処理によって行なわれるという構造になっている。データは上から下のものへと受け渡されて処理が施されて行く (図 1、左)。このようなコードの場合、下位のサブルーチンの方になると、扱っているデータが全体のどの部分に相当するのか、どのようなデータを扱っているのかを認識するのが非常に難しくなってくる。また、そのサブルーチンが汎用的に書かれていて抽象性が高くなると、呼ばれる場所により扱っているデータが異なるような場合も生じるであろう。このようなコードを並列化しようとするには、プログラムの一体どこに並列化の操作を適用したらよいか判断するために、プログラムの内容について熟知している必要がある。特に、既存の非並列のコードを並列化しようとする場合にはそれを書いた本人が行なうか、そうでない場合はそのアプリケーションの内容、アルゴリズムの理解、プログラムの解析などから行なわなければならず非常な困難さを伴うことになる。また、構造化プログラムにもなっていないものの場合は書き直しなどさらに困難になることも考えられる。そのようなコードは問題や計算条件に固有のものとなり、ちょっとした計算条件や手法の変更のために全てを書きなおすということが往々にして起こる。並列環境の標準というものがなかなか確立されないところが一つの原因でもあるが、もともと並列実行を考えて設計されていないコードを後から並列化するために生じる弊害であるとも言えるだろう。

一方オブジェクト指向では、各データがそれに関する処理といっしょにクラス化され、そのクラスに指示 (メッセージ) を送ることにより処理がクラスの内部で行なわれる (図 1、右)。従ってデータ構造を見失う事がなく、領域分割などを行なうにあたって作用させるべき箇所を判断するのが容易になる。このことだけでも並列プログラムを書く場合の利点となりうるが、しかしながら、単純にデータとそれに対する処理をまとめてクラスとするというような設計を行なうとクラスの内部においては構造化プログラムと変わりがなくなってしまい、並列化という観点からはオブジェクト指向の利点を十分に生かしているとは言えない。初めから並列化を考慮してプログラムを設計するのであれば、上に述べたような問題からもアプリケーション固有の解法部分と並列

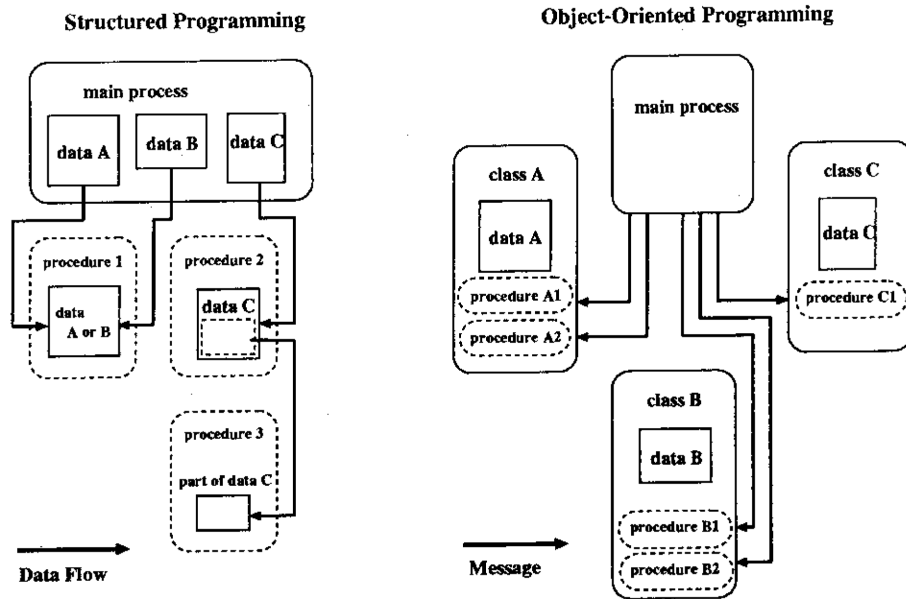

図 1: 手法によるプログラム構造の違い

化に関する処理を分けて開発出来るような仕組みになっているのが望ましい。さらに一歩踏みいれて、オブジェクト指向の情報隠蔽により並列処理部を解法部から隠してしまうことが出来るならば、解法部を書くにあたって並列化を意識せずに、非並列の場合と同じようにプログラムを書く事が出来るだろう。一方、このことにより並列処理部の方に関しても、アプリケーション固有の部分から分離されることにより、並列処理部のプログラムを書くにあたりアプリケーションの内容やアルゴリズムなどを熟知しなくても行なえるようになることが期待出来る。また、オブジェクト指向のカプセル化によって、インターフェースを合わせることによって中身の実装を自由に行なえることから、並列化の実現を自由に選択することが出来る。すなわち、異なる並列環境の計算機に対してもそれに対応する並列処理のクラスを用意することによって、解法部分は全く変更せずに実行することが出来るようになる。このように、オブジェクト指向を利用して初めから並列を意識した設計を行なうことにより、保守性、互換性にすぐれたコードを書くことが出来る。

オブジェクト指向、特に C++を並列プログラムに利用するという試みは多数行なわれている [3] が、それらの殆んどが並列言語やメッセージパッシングライブラリをオブジェクト指向の特徴を採り入れて構成しようというものである。従って用意されている機能はどうしても並列処理に用いられる単独の機能、例えばデータの転送などという下位のレベルのものになってしまうため、並列プログラムを書く際に並列処理を意識しながら書かなくてはならないという意味では従来と同じになるだろう。解法のクラス化などによって並列化を行なうもう一つのアプローチとしては、各並列プロセスのオブジェクト間でデータ転送としてメッセージを交換させるというものが考えられる。しかし、これも同様にデータ転送部を解法クラスの外側でデータ転送の部分を書かなければならず並列を意識しなければならないという点では同様である。また、オブジェクト指向、特

にC++による数値計算のためのクラスライブラリはいくつかのプロジェクトが存在する[6][7]が、それらは数値アルゴリズムの部分の再利用性、保守性を高めようというのが主目的であり、並列に関しては考慮されていない。本研究において提案しているのは、もちろんプログラムの再利用性、保守性を目的のひとつとしてはいるが、アルゴリズムやデータを部品としてクラス化することが目的ではない。本論は、各科学計算に使われるそれぞれのデータ構造について、それを並列処理の機能を内蔵したものとして設計し、普通の型のように用いるようにすると、並列プログラムの開発に様々な利点があるということを主張するものである。また、特定のクラスライブラリを提供するというものではなく、並列プログラムの開発手法についての新たなパラダイムを提案するものである。

本報告書では、数値流体計算において領域分割により並列化を行なうものとして、有限体積法(FVM)による２次元オイラー方程式の解法コードをここで提案する設計指針に沿って開発した。プログラム、クラスライブラリは全てC++によって書き、並列環境としてはMPIを利用している。コードはIBM SP2上で実行し、性能評価を行なった。

2　並列プログラムの分析

本節では、流体解析や構造解析のような分布系の問題を差分法、有限体積法、有限要素法などの手法で解くものを、領域分割により並列化していくものについて考察する。

これらの数値計算プログラムというのは、データとその振舞であるメソッドからなるというクラス構造に照らし合わせて考えて見ると、格子点座標値や物理変数などのデータと、それの処理を行なう数値解法の各手順、処理が、そのままクラスとしての構造にあてはめられることが分かる。領域分割などの並列化に関する手順もデータに働きかける処理としては同じであるためにこのままクラスとしてしまうことが可能である。しかしながら、単に全体のアルゴリズムを並列処理まで含めて一つのクラスとしたのでは、その内部では純粋な数値解法と並列化に関する部分とが入れ込んでしまうという前節で構造化プログラミングの問題としてあげたような状況が起こり兼ねない。もちろん各部分をさらにクラス化することにより混乱を少なくする事が可能であるが、決定的な改善とはならない。

データに関する処理ということで数値解法と並列化を同列に扱って来たが、よくよく考えて見ると数値解法というのはデータの内容に対して処理が行なわれるのに対し、領域分割などの並列化処理はデータの構造というものに対して操作が行なわれる異なる種類の操作として理解出来る。領域分割によって生じる領域境界におけるデータ転送もデータの内容を実際に転送するので混同してしまうが、これもデータのどの部分を転送するかということを行なうという意味からデータ構造に対する処理として捉える事が出来る。この観点からクラス設計を見直すと、データの振舞として並列化の一連の処理を合わせて、領域分割による並列化についてそのクラス自身が自分で対処出来るようなデータのクラスを設計することが出来る。この設計により並列化の処理をデータクラスの内部、数値解法を外部とするような分離を行ない、プログラムを組み立てることが出

にC++による数値計算のためのクラスライブラリはいくつかのプロジェクトが存在する[6][7]が、それらは数値アルゴリズムの部分の再利用性、保守性を高めようというのが主目的であり、並列に関しては考慮されていない。本研究において提案しているのは、もちろんプログラムの再利用性、保守性を目的のひとつとしてはいるが、アルゴリズムやデータを部品としてクラス化することが目的ではない。本論は、各科学計算に使われるそれぞれのデータ構造について、それを並列処理の機能を内蔵したものとして設計し、普通の型のように用いるようにすると、並列プログラムの開発に様々な利点があるということを主張するものである。また、特定のクラスライブラリを提供するというものではなく、並列プログラムの開発手法についての新たなパラダイムを提案するものである。

本報告書では、数値流体計算において領域分割により並列化を行なうものとして、有限体積法(FVM)による2次元オイラー方程式の解法コードをここで提案する設計指針に沿って開発した。プログラム、クラスライブラリは全てC++によって書き、並列環境としてはMPIを利用している。コードはIBM SP2上で実行し、性能評価を行なった。

2　並列プログラムの分析

本節では、流体解析や構造解析のような分布系の問題を差分法、有限体積法、有限要素法などの手法で解くものを、領域分割により並列化していくものについて考察する。

これらの数値計算プログラムというのは、データとその振舞であるメソッドからなるというクラス構造に照らし合わせて考えて見ると、格子点座標値や物理変数などのデータと、それの処理を行なう数値解法の各手順、処理が、そのままクラスとしての構造にあてはめられることが分かる。領域分割などの並列化に関する手順もデータに働きかける処理としては同じであるためにこのままクラスとしてしまうことが可能である。しかしながら、単に全体のアルゴリズムを並列処理まで含めて一つのクラスとしたのでは、その内部では純粋な数値解法と並列化に関する部分とが入れ込んでしまうという前節で構造化プログラミングの問題としてあげたような状況が起こり兼ねない。もちろん各部分をさらにクラス化することにより混乱を少なくする事が可能であるが、決定的な改善とはならない。

データに関する処理ということで数値解法と並列化を同列に扱って来たが、よくよく考えて見ると数値解法というのはデータの内容に対して処理が行なわれるのに対し、領域分割などの並列化処理はデータの構造というものに対して操作が行なわれる異なる種類の操作として理解出来る。領域分割によって生じる領域境界におけるデータ転送もデータの内容を実際に転送するので混同してしまうが、これもデータのどの部分を転送するかということを行なうという意味からデータ構造に対する処理として捉える事が出来る。この観点からクラス設計を見直すと、データの振舞として並列化の一連の処理を合わせて、領域分割による並列化についてそのクラス自身が自分で対処出来るようなデータのクラスを設計することが出来る。この設計により並列化の処理をデータクラスの内部、数値解法を外部とするような分離を行ない、プログラムを組み立てることが出

来るようになる。この分離は二つの処理を完全に分けてしまうものではなく、むしろ階層的なものとなっている(図 2)。それは、データ転送ではやはりその内容である値を送ることになるので、完全に分離してしまうことが単純ではなく、その事によってオーバーヘッドが生じることを恐れるためである。ここで、データ構造とその内容を分けることにより二つの処理を完全に分離してしまい、アプリケーションから独立したデータ型を定義するような設計も可能であるだろう。どのレベルでのクラス化を行なうかということを考えたときに、非常に基本的なクラスだけを用意してそれを組み立てて全体を組み立てるというアプローチも考えられる。しかしながら、本論では並列化に煩わされずに解法アルゴリズムの開発に集中出来るということを念頭に置き、並列処理部分の完全な分離を行なって汎用的なクラスを部品として作ることよりも、より上位でのクラスの設計を行う方針をとった。もちろん今回の設計においても、データクラス内では各部分がさらに下位のクラスによって構成されており、そのレベルでは部品的な汎用クラスが与えられている。従って、例えば流体解析用のデータクラスから流体の変数のクラスを他のアプリケーションの変数のクラスに置き換えることにより、それ専用のデータのクラスを構成しなおす事が可能である。ただし、本論文においてはデータクラスのレベルでの設計をどうすべきかというのが主たる論点である。

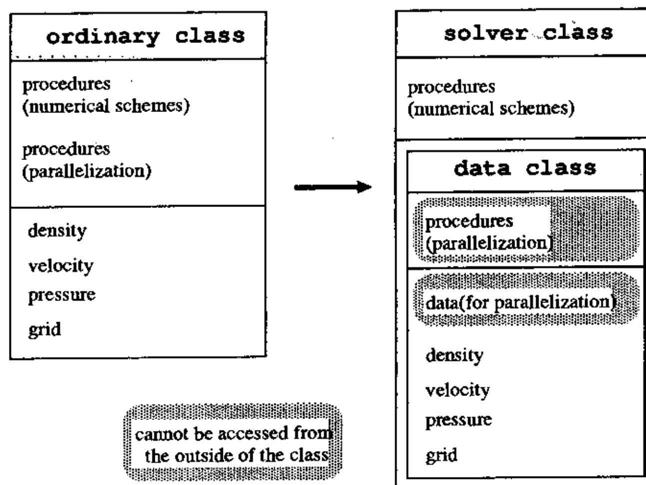

図 2: クラス設計の概念図

3 並列処理の隠蔽

前節のようなクラスの設計指針をたてたところで、どのようにして並列処理部が数値解法のクラスから隠されることになるのか、また、どのようにして並列化を意識せずに並列コードが書けるのかを具体的に述べていく。

領域分割とメッセージパッシングという手法で並列処理を行なう際に、具体的な並列化の為の処理というのは分割された領域のお互いの位置関係や範囲の管理、及び、境界部の値の受渡しで

来るようになる。この分離は二つの処理を完全に分けてしまうものではなく、むしろ階層的なものとなっている(図2)。それは、データ転送ではやはりその内容である値を送ることになるので、完全に分離してしまうことが単純ではなく、その事によってオーバーヘッドが生じることを恐れるためである。ここで、データ構造とその内容を分けることにより二つの処理を完全に分離してしまい、アプリケーションから独立したデータ型を定義するような設計も可能であるだろう。どのレベルでのクラス化を行なうかということを考えたときに、非常に基本的なクラスだけを用意してそれを組み立てて全体を組み立てるというアプローチも考えられる。しかしながら、本論では並列化に煩わされずに解法アルゴリズムの開発に集中出来るということを念頭に置き、並列処理部分の完全な分離を行なって汎用的なクラスを部品として作ることよりも、より上位でのクラスの設計を行う方針をとった。もちろん今回の設計においても、データクラス内では各部分がさらに下位のクラスによって構成されており、そのレベルでは部品的な汎用クラスが与えられている。従って、例えば流体解析用のデータクラスから流体の変数のクラスを他のアプリケーションの変数のクラスに置き換えることにより、それ専用のデータのクラスを構成しなおす事が可能である。ただし、本論文においてはデータクラスのレベルでの設計をどうすべきかというのが主たる論点である。

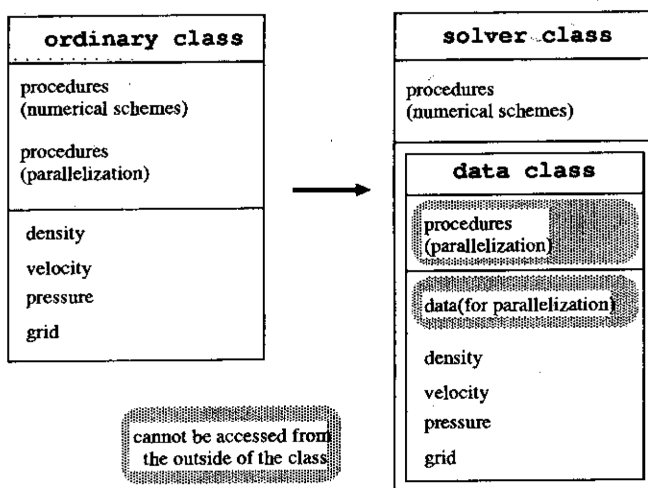

図 2: クラス設計の概念図

3 並列処理の隠蔽

前節のようなクラスの設計指針をたてたところで、どのようにして並列処理部が数値解法のクラスから隠されることになるのか、また、どのようにして並列化を意識せずに並列コードが書けるのかを具体的に述べていく。

領域分割とメッセージパッシングという手法で並列処理を行なう際に、具体的な並列化の為の処理というのは分割された領域のお互いの位置関係や範囲の管理、及び、境界部の値の受渡しで

ある。一般にこのような並列プログラムは、全体の領域分の格子データを並列に実行しているプロセスのうち、一つ、または全部のプロセスで読み込み、それをプロセス数に対応させて分割、それぞれ責任分を確保し計算に移るというような流れをとっている。計算中には、各領域の境界値を隣接の領域を実行しているプロセスと受渡しを行なうという処理も必要である。これらに付随した、領域分割の操作、各領域の位置関係、分割された領域の範囲やデータ番号の管理、メッセージパッシングの送信、受信の処理、などを具体的にはプログラム中で書かなければならず、これらの操作が前節で述べたような構造化プログラムのスタイルの中で、例えば流体などのアプリケーションに依存した解法部分などと入れ込んでしまうと非常に複雑なものとなってしまい、エラーやバグを招く原因となる。

前節で述べたようなオブジェクト指向による設計というのはこれらの操作を異なるクラスとして分離することによって混乱をなくしエラーを少なくしようというものであるが、その結果として並列に関する処理を一方のクラス(基本データ構造)の内部に隠蔽することにより、他方(アプリケーション)のクラスから並列を意識せずに解法を実装出来るようになるものである。このことは、単純にはまずクラス内部への情報遮蔽ということで達成される。それには、データ転送を内部でうまく取り扱う必要がある。また、入力すべき格子のデータや、初期、境界条件を各分割領域毎に用意するようでは、並列化に関する面倒さが解消されたとは言えない。そこで、それに対する対策も必要となる。以上の項目はそれぞれ完全には独立したものではなく関わりあったものであり、これらがまとまって並列化の隠蔽化を達成する。以下でそれぞれについて述べていくことにする。

3.1 境界条件の再構成

領域分割により並列化を行なう場合、全体のデータが分割されその部分が各並列プロセスに割り当てられることになる。多くの場合この分割による格子番号の管理などが面倒な点であり、またエラーを起こしやすいところである。分割された領域は、全領域を同時に管理している場合にこそ、全体の中での位置関係などが問題となる。しかしながら、各プロセスにとっては自分の担当領域が全てであり、それが全体のものであろうと分割された一部分であろうとも、完全な情報を持ったオブジェクトとして変わりが無い。逆に言うとこのように変わりがないようにするのがオブジェクト指向による設計とも言える。

ここで、領域分割がすでに行なわれているとして前節で述べたような構造に照らし合わせて、各個別プロセスではプロセス間のデータ転送がどのように解釈出来るか考えて見よう。他のプロセスとの境界値の受渡しは、転送を行なう相手の番号などが与えられていて特定出来るとすると、そのプロセスの解法部分から見れば単にいくつかある境界条件のうちの一つとして位置付けられるものである。すなわち境界条件として、物理的な条件によって境界領域の変数を与えるのではなく、他のプロセスからもってきた値を与えるということを行なっていると見るのである。領域分割が行なわれた時にそれぞれの部分領域についてそれに対応するデータ転送の相手などを含んだ境界条件の情報が与えられるように出来ると、個別のプロセスはそのプロセスにとって非並列のプ

ログラムの場合と同じように完全な領域とそれに付随した境界条件を扱っていることになる。すなわち解法部分から見れば「領域 ＋ 境界条件」という全く並列処理を意識しない構成となる(図3)。結局、各並列プロセスに、そのプロセスが扱う領域のデータとデータ転送の情報も含めた境界条件のデータを何らかの仕組みで渡すことが出来れば並列化はクラスの内部で処理することが可能となるのである。

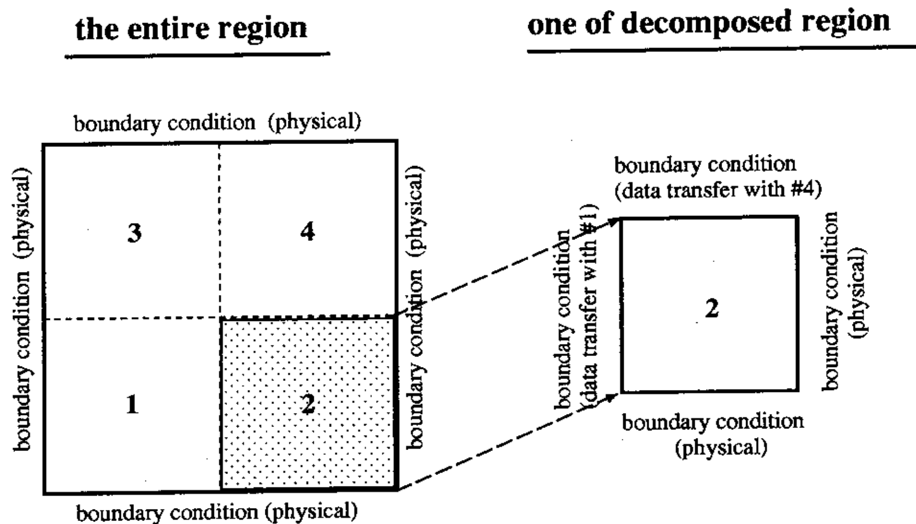

図 3: 全領域と部分領域における境界条件

データ送受信を境界条件の一つとして設計することにはもう一つのメリットがある。従来の構造プログラミングでは、送受信と物理的な境界条件は異なるモジュール(サブルーチン)として構成されるようになっていて、それぞれが順番に実行されるようになっていた。分割数によっては、物理的な境界が無くデータの送受信だけ行なうようになっているプロセスが生じたり、それぞれのプロセスでの物理的な境界とデータ送信を行なう境界の数がばらばらとなり、それぞれの実行でオーバーヘッドが生じる事となる。本設計のようにすることにより、あるプロセスにおいてデータの送信を行なっているときに、本来ならまたされているようなプロセスにおいては物理的な境界条件を実行することが出来るために、そのようなオーバーヘッドを軽減することが期待出来る。

3.2 領域分割の処理

上で述べたような状態にするために、個別のプロセスに対する格子データや、境界条件をそのプロセスの数だけ用意しなくてはならないのでは、並列化を意識しなくてはならないという意味で負担であり、このような設計をした恩恵が半減してしまう。例えば、境界条件としてデータのやりとりをする相手のプロセス番号やデータの位置などを複数のプロセス分だけいちいち自分で考えて用意しなくてはならないのでは、結局並列に関するそういった面倒な処理を陽的に行なわなければならない。また、計算領域データが変わったり、分割数を変える度に新たにそれに対するものを作りなおさなくてはならない。これでは、手間や繁雑さはあまり変わらないといっても

いいだろう。

並列化に手間をとられずにプログラム開発を従来と同じように行なう事が出来るようにということを念頭に置くと、格子や境界条件などのデータも全体のものを一つだけ用意すればいいようにするのが好ましい。そのため、各並列プロセスで扱う部分領域についてのデータというのは全領域に対するデータ類から、クラス内部においてそれぞれのプロセスで処理して構成しなおすようにする。例えば構造格子であれば、各次元方向で均等に分割するという方法で簡単に分割を行なうことが出来る。しかし、そのように単純でなくとも何らかの分割アルゴリズムを用意出来れば、どんなデータ構造であっても全体のデータを読み込んでからそれを処理してそのプロセスが扱う部分領域のデータだけ保持するということは可能である。

さて、内部で格子分割を行なうのであるからどのように分割されているか、お互いの位置関係がどうなっているか、という情報は全てそこで得られる。従って、境界条件の情報も全体領域のものを読み込んだ後、担当部分のものに対応するように変換することも可能なはずである。ただしこの場合、境界が領域の分割により新しく生じる箇所についてはデータの送受信を行なうという境界条件を与えるようにする。このようにして、領域分割などを意識しない全領域のデータとそれに対する境界条件を、個別のプロセスでそれぞれ処理し、そのプロセスに対する領域データと境界条件に作りなおして個別の準備をするという仕組みが構成出来る。

これらの処理はアプリケーション固有の数値解法とは全く関係のない部分であり、データ転送に必要であるために物理変数の情報も必要となるが、データの構造さえ分かれば行なえるものである。従って、本論文での設計におけるデータのクラスの内部で処理は済ますことが可能であり、外から見える必要がない。そのように実装することはオブジェクト指向言語と言われるものを使用することにより簡単に行なえる。このようにクラス内部に領域分割とデータ転送という並列の処理を隠してしまえば、解法のクラスはデータクラスの物理変数を非並列のコードのように利用するだけで、並列のプログラムを書いている事になるのである。

3.3 並列入出力

各並列プロセスでそれぞれ全領域のデータを読み込み処理をすることにしたが、構造格子を対象として考えた場合にはデータのサイズさえ分かれば領域分割を行なう事が可能であるため、個別のプロセスそれぞれで格子全体分のデータを読み込む必要はない。分割領域が確定された後、自分の担当領域だけをファイルから読み込めばよい。これによって、多少とも入力時の時間短小が行うことが出来るのと同時に一時にせよ全体の領域分のメモリを確保することがなくてすむために、非常に大きな領域を多数のノードで処理するときなどに有効であると考えられる。ただし、非構造のデータに関して同様の仕組みをとることは難しいだろう。

出力に関しては各データ構造共通に考える事が出来る。データを出力するのに際してこれを一つのファイルとして行なおうとすると、データを一時にせよ一つのプロセスに集めなければならない。これは一定の時間ステップ毎に出力する場合は余分な通信が生じることとなり性能の低下の原因となる上に、ここでもメモリ確保が問題となる。ある一つのプロセスに全体のデータを確

保するだけのメモリを用意しようとすると、分散メモリ環境下では一つのユニットのメモリの上限で計算の規模が制限されることとなり、大規模計算に対応出来るという分散メモリ環境の利点を消してしまうこととなる。従って、ここでは計算結果を各プロセス独立に担当範囲のものについてのみ出力するような実装としている。この際、個々のプロセスが出力するファイル名をそれぞれ指定しなくてはならないのでは面倒であり、わずかながら並列を意識しなくてはならないことともなるので、「並列に出力する ＋ ファイル名」と指示することにより、(ファイル名＋プロセス番号) というファイルをそれぞれが書くようなメソッドを用意している。

ばらばらに結果が出力されてしまうことにより後処理で結局時間がかかるのではという疑問もあるかもしれないが、後処理が並列実行のものでないのであれば、読み込み処理の方で対応するようにしておけば全体で一つになっているものを読もうが、分かれているものを読もうが、入力にかかる時間はそれほど変わらないはずである。もし、後処理が並列に行なわれるようなものになっているのであれば、データが分割されていることにより並列入力の効果が得られるであろう。もし後処理に関しても同じデータクラスを用いて構成すれば、非常に統合性の良い数値計算の環境を構築出来るのではないかと考える。

4　数値流体解析におけるクラス設計

以上のような方針に沿って、ここでは２次元オイラー方程式を構造格子上で有限体積法により解くコードの設計を行なった。

まず最上位のクラスとしてデータ (流れ場) のクラスと数値解法 (ソルバー) という二つを用意する (図 4)。

ソルバーのクラスは流れ場のクラスをデータとして参照し、計算を行なって行く。その際、参照出来るのは物理的な変数や格子点の座標などのデータのみであって、並列処理に関する部分には直接操作することは出来ない。斜線部の背景で示されたデータ、メソッドが陽的に並列処理に関する部分であるが、全て流れ場のクラスの中に入っており、ソルバーのクラスからは隠されている。ソルバーのクラスからは、[applyBoundaryCondition]、[setDecomposition]、などのメッセージを流れ場のクラスに送ることによってその処理が実行される。流れ場のクラスには格子データ、変数、初期条件、境界条件、メッセージパッシングライブラリへのインターフェースの各クラスが含まれている。このように、下位のクラスになってくると部品としての特徴が強くなっており、実際にいくつかのクラスを置き換えることにより、異なるアプリケーションへの対応も可能になるだろう。さらに基本的なクラスとして２次元の配列などのクラスがあるが、これは構造格子に対する基本的なデータの構造を実現するものである。従って、格子の座標値や物理変数といったデータの内容に当るものは全てこのクラスから派生して作られている。構造格子の場合は２次元配列がデータ構造のクラスとして使用されるが、異なるデータ構造による解法にはそれぞれ対応する基礎クラスを基として同様のクラス構成を実現することが出来る。従って、この設計は２次元オイラー方程式固有のものではなく、領域分割による数値解法一般に適用出来るものと考えている。

保するだけのメモリを用意しようとすると、分散メモリ環境下では一つのユニットのメモリの上限で計算の規模が制限されることとなり、大規模計算に対応出来るという分散メモリ環境の利点を消してしまうこととなる。従って、ここでは計算結果を各プロセス独立に担当範囲のものについてのみ出力するような実装としている。この際、個々のプロセスが出力するファイル名をそれぞれ指定しなくてはならないのでは面倒であり、わずかながら並列を意識しなくてはならないことともなるので、「並列に出力する ＋ ファイル名」と指示することにより、(ファイル名＋プロセス番号) というファイルをそれぞれが書くようなメソッドを用意している。

ばらばらに結果が出力されてしまうことにより後処理で結局時間がかかるのではという疑問もあるかもしれないが、後処理が並列実行のものでないのであれば、読み込み処理の方で対応するようにしておけば全体で一つになっているものを読もうが、分かれているものを読もうが、入力にかかる時間はそれほど変わらないはずである。もし、後処理が並列に行なわれるようなものになっているのであれば、データが分割されていることにより並列入力の効果が得られるであろう。もし後処理に関しても同じデータクラスを用いて構成すれば、非常に統合性の良い数値計算の環境を構築出来るのではないかと考える。

4　数値流体解析におけるクラス設計

以上のような方針に沿って、ここでは２次元オイラー方程式を構造格子上で有限体積法により解くコードの設計を行なった。

まず最上位のクラスとしてデータ (流れ場) のクラスと数値解法 (ソルバー) という二つを用意する (図 4)。

ソルバーのクラスは流れ場のクラスをデータとして参照し、計算を行なって行く。その際、参照出来るのは物理的な変数や格子点の座標などのデータのみであって、並列処理に関する部分には直接操作することは出来ない。斜線部の背景で示されたデータ、メソッドが陽的に並列処理に関する部分であるが、全て流れ場のクラスの中に入っており、ソルバーのクラスからは隠されている。ソルバーのクラスからは、[applyBoundaryCondition]、[setDecomposition]、などのメッセージを流れ場のクラスに送ることによってその処理が実行される。流れ場のクラスには格子データ、変数、初期条件、境界条件、メッセージパッシングライブラリへのインターフェースの各クラスが含まれている。このように、下位のクラスになってくると部品としての特徴が強くなっており、実際にいくつかのクラスを置き換えることにより、異なるアプリケーションへの対応も可能になるだろう。さらに基本的なクラスとして２次元の配列などのクラスがあるが、これは構造格子に対する基本的なデータの構造を実現するものである。従って、格子の座標値や物理変数といったデータの内容に当るものは全てこのクラスから派生して作られている。構造格子の場合は２次元配列がデータ構造のクラスとして使用されるが、異なるデータ構造による解法にはそれぞれ対応する基礎クラスを基として同様のクラス構成を実現することが出来る。従って、この設計は２次元オイラー方程式固有のものではなく、領域分割による数値解法一般に適用出来るものと考えている。

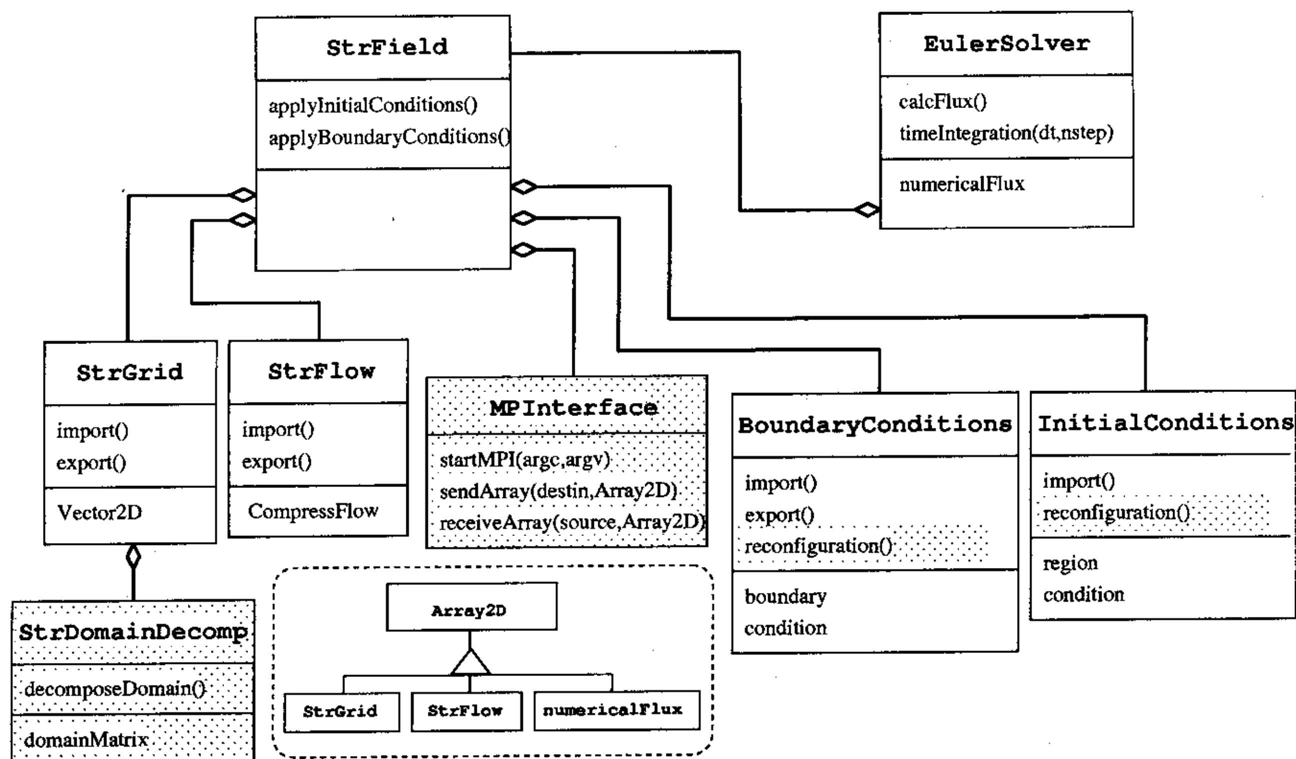

図 4: 流体解析のためのクラス構成

5 プログラムの実際

これらのクラスを用いて実際にプログラムはどのように書かれるのかを見て行きたい。まず、今回設計した流体解析用のクラスを用いるとメインのプログラムをどのように書けるかという具体例を紹介する。これは、全てのクラスが完成した後、一番上位ではそれらをどのように使ってプログラムを構成するのかという一例をあげるものである。次に、これらのクラスがライブラリとして与えられたときに自分自身のコードを開発するためにどのようにそれらを利用すればよいのかを説明する。

ここで説明するものは今回開発したものに沿っているが、これは設計の概念を説明するための一例であり、この設計の決定した仕様を説明しているというようなものではない。

5.1 メインプログラム

まず、非並列のプログラムを以下に示す。

```
#include <iostream.h>
#include "eulerstrfvm2d.h"
```

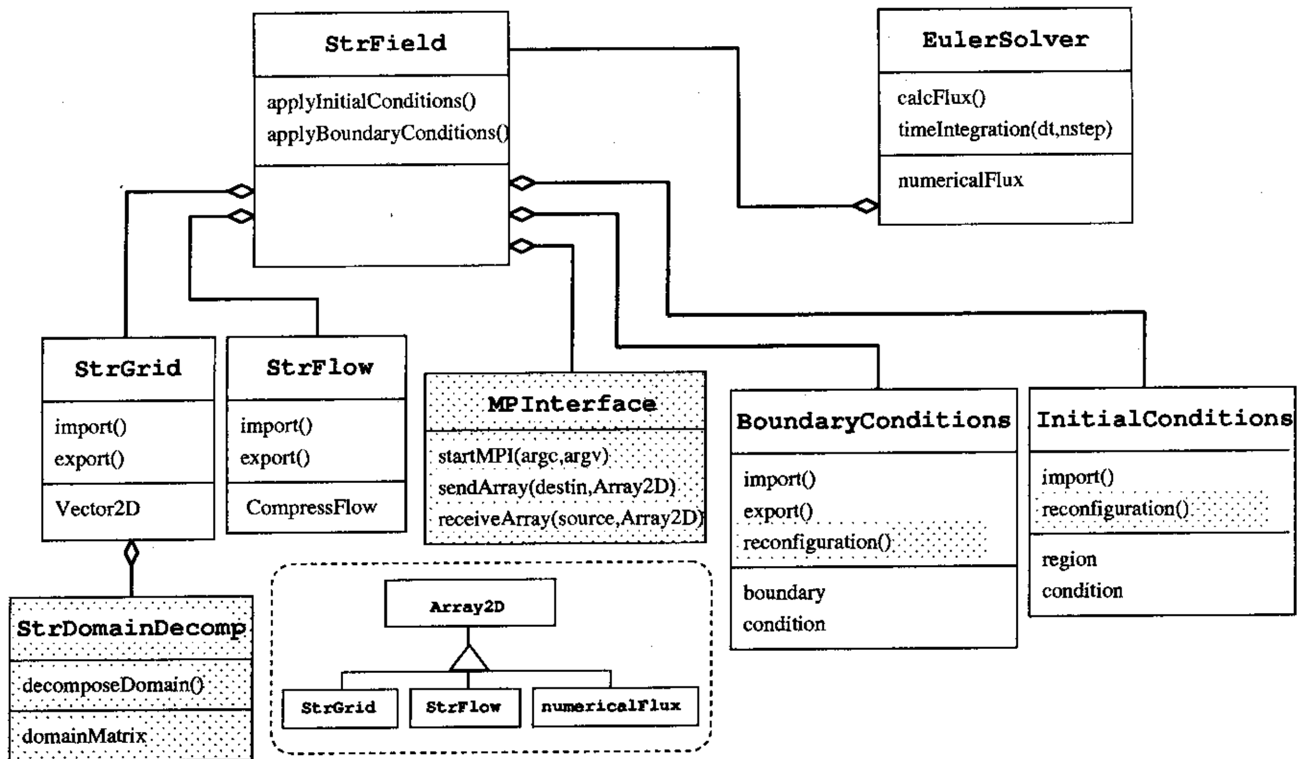

図 4: 流体解析のためのクラス構成

5 プログラムの実際

これらのクラスを用いて実際にプログラムはどのように書かれるのかを見て行きたい。まず、今回設計した流体解析用のクラスを用いるとメインのプログラムをどのように書けるかという具体例を紹介する。これは、全てのクラスが完成した後、一番上位ではそれらをどのように使ってプログラムを構成するのかという一例をあげるものである。次に、これらのクラスがライブラリとして与えられたときに自分自身のコードを開発するためにどのようにそれらを利用すればよいのかを説明する。

ここで説明するものは今回開発したものに沿っているが、これは設計の概念を説明するための一例であり、この設計の決定した仕様を説明しているというようなものではない。

5.1 メインプログラム

まず、非並列のプログラムを以下に示す。

```
#include <iostream.h>
#include "eulerstrfvm2d.h"
```

```
int main(int argc, char **argv)
{
	//-------------------------------
	//	field settings
	//-------------------------------
	StrFvmField2D<double> field;

	field.importGridFile("sample.grid");
	field.importInitialCondition("sample.ic");
	field.importBoundaryCondition("sample.bc");

	//--------------------------------
	//	solver
	//--------------------------------
	EulerStrFvm2D<double> solver(field);

	double dt = 0.001;
	double nloop = 200;

	solver.applyInitialCondition();
	solver.timeIntegrationEulerExplicit(dt,nloop);

	//--------------------------------
	//	file output
	//--------------------------------
	field.exportGrid("sample");
	field.exportFlowField("sample");


	return 0;
}
```

上から見ていくと、まず内部の諸変数をdouble型として持つデータクラスのオブジェクトとして'field'を宣言している。これは、２次元の有限体積法用に構成された流体解析のデータクラスである。このクラスは、各データの入出力のメソッドがサポートされており、上から順に格子、初期

条件、境界条件のファイルを読み込んでいる。これにより、'field' は計算に必要な実際のデータを伴ったものとして用意される。次に、'field' をあてがうようにして、計算手法のクラスのオブジェクトである'solver' を宣言する。'solver' は、'field' に含まれる情報により初期条件を適用し、データを使いながら計算を進めて行く。最後に'field' が結果の出力を行ない計算が終了する。

これに対して並列のプログラムは次のように書ける。

```
#include <iostream.h>
#include "eulerstrfvm2d.h"

int main(int argc, char **argv)
{
        //--------------------------------
        //        field settings
        //--------------------------------
        StrFvmField2D<double>  field;

        field.useMPI(argc,argv);
        int ndivi = 5;
        int ndivj = 2;
        field.decompose(ndivi,ndivj);

        field.importGridFile("sample.grid");
        field.importInitialCondition("sample.ic");
        field.importBoundaryCondition("sample.bc");

        //--------------------------------
        //        solver
        //--------------------------------
        EulerStrFvm2D<double>  solver(field);

        double dt = 0.001;
        double nloop = 200;

        solver.applyInitialCondition();
        solver.timeIntegrationEulerExplicit(dt,nloop);
```

```
//--------------------------------
//        file output
//--------------------------------
field.exportGridParallel("sample");
field.exportFlowFieldParallel("sample");


return 0;
}
```

ここで、イタリックで表されているところが並列のコードとなって追加変更された部分である。まず、'field' を非並列の場合と同じに宣言した後、メッセージパッシングライブラリとしてMPIを使用する宣言を行なう。その後、ここでは２次元を扱っているので、２次元領域をそれぞれの次元方向でいくつに分割するかを指定している。このプログラムの例では５ x ２に領域を分割して１０台による並列計算を行なうことを指定している。その後は非並列と同様に、格子データ、初期、境界各条件のファイルを読み込んでいる。データを読み込むのに用いるメッセージは非並列の場合と同じであるが、[decompose(ndivi,ndivj)] というメッセージを事前に送ることで、各データを読みこむ際にそれぞれその分割領域に合わせたデータの再構築を行ない、'field' を各領域に対応したデータの組からなるものにする。こうして設定された'field' はそれぞれの並列プロセスにおける'solver' から見ると非並列の場合と同じような単独の領域を扱っているようになる。また、時間積分を行なっているところで内部的に'field' を通じて境界条件を適用しているが、メッセージパッシングは条件のひとつとして処理されている。これらの構成によって、'solver' は並列、非並列に関わらず同じデータの扱いをすればよいため、クラスを並列を考えることなく書くことが出来る。従ってリストでも分かるように'solver' に関する部分では非並列のプログラムから何も追加、変更が行なわれない。最後のデータ出力は前節で述べたように各プロセスの担当部分のみを出力するために並列出力用のメッセージを送っている。

5.2 データクラスへのアクセス

ここでは圧縮性流体の計算用にデータクラスが設計されているとして、個別のプログラムを書くとき、つまり計算の手法の部分を書くときにデータクラスどのように利用するのか説明する。

次に示すのはデータクラスの定義の一部分であるが、'grid'、'flow' などがクラスとして内部に定義されており、これらがその名前で参照出来るようになっている。

```
template<class Type>
```

```
class StrFvmField2D {
  private:
    MPInterface _mpi;
      .
  protected:
    //  < reference >
    StrGrid2D<Type> &grid;
    StrFlow2D<Type> &flow;
    StrInitialCond2D<Type> &ic;
    StrBoundaryCond2D<Type> &bc;

    Array2D<Type> &area;
    Array2D<Vector2D<Type> > &normal1;
    Array2D<Vector2D<Type> > &normal2;
    Array2D<Type> &length1;
    Array2D<Type> &length2;
};
```

これらはいずれも２次元の配列のクラスから導出されているのでそれぞれ２次元配列のように、例えばgrid(i,j)、flow(i,j)のように参照することが出来る。これらのクラスはその内部にデータを保持していて、例えば'grid'であれば座標値であるx,yを持っている。それらはgrid(i,j).x、grid(i,j).yのようにして通常の変数としてプログラムにおいて使用出来る。flowはその内部に各流れ場の速度場、圧力などのデータを持っており、同様にして参照することが可能である。これらの変数は外部からアクセス出来るように設定されている。解法のクラスから参照するときにはこれらのクラス内部のデータに上で述べたようなインターフェースでアクセスすることによって、普通のデータを使っているのと同じように記述すればよいのである。

5.3 継承による解法クラスの開発

例えばクラスライブラリを構成する場合、5.2節で述べたようなデータクラスを与えるだけでもよいのであるが、C++のようなオブジェクト言語では継承という仕組みにより自分のクラスを作ることが出来る。これは、あるクラスの機能を維持しつつ、新たな固有の変更を加えたクラスを作り上げる機能である。格子のクラスなどが２次元配列クラスからの導出によりつくられていると述べたのはこのことである。ここでは、格子や流れ場は２次元配列として、そのクラスの特徴を備えながら、格子クラスはデータとしてxとyを、流れ場は速度場や圧力をもつというようになっている。また、何らかのメソッドが定義されていたときに、その部分だけ変えたものが欲しいというときにも、継承するクラスのそのメソッドを書き換えることが出来る。従って、もし解法

のクラスが一つ与えられていたとすれば、そのクラスの数値解法の処理のメソッドを変更したクラスを継承により書くことにより、簡単に様々なアルゴリズムのコードを書くことが出来る。例えば、下がすでにある解法のクラスだとする(一部だけ示している)。

```
template<class Type>
class Solver {
    void applyInitialCondition() {}
    void applyBoundaryCondition() {}
    void eulerExplicit(Type dt, int nloop) {
        (ALGORITHM 1)
    }
};
```

ここで、[eulerExplicit]というメソッドにおいて解法のアルゴリズム(ALGORITHM 1)を実装しているとする。これと異なる手法、(ALGORITHM 2)によるコードを作るときにはこのクラスを継承し、そのメソッド内で異なるアルゴリズムを実装することで達成される。それには以下のようにすればいい。

```
template<class Type>
class DerivedSolver : public Solver {
    void eulerExplicit(Type dt, int nloop) {
        (ALGORITHM 2)
    }
};
```

共通の流体解析という範疇では初期条件や境界条件などのところはそのまま用いることが出来るので、それは基のクラスのものを用いることにすると、アルゴリズムに相当するところだけ書けばよいことになる。従って、この導出されたクラスにはそれらのメソッドの記述はなく、それにより基のクラスのものが用いられることになる。新たなコードを書くための作業はこのように簡単になり、しかも並列化は全く意識する必要がない。

もちろんこれは並列処理部分についても同様のことが言えるわけで、今回の場合では2次元配列などのデータ構造を並列化することに集中して考えればよく、アルゴリズムの違いには注意をはらう必要がない。またもちろん並列化の手法の変更も解法と同様にして行なうことが可能である。

データクラスだけではなく、テンプレートとなるような解法のクラスを用意することで、以上のように簡単に様々な計算手法を試すさいのプログラミングの面倒を、並列処理も含めて軽減す

ることが出来るのである。

6 計算例

第４、５章で設計したコードの性能を IBM SP2 （４８ノード）により実行、評価した。扱った問題は円柱に高速流があたるものであり、図 5(左) に示すように円柱の風上側のみ計算領域をとって行なった。２次元のオイラー方程式をセル中心の有限体積法により解いており、MUSCL と Roe のリーマンソルバー [5] により流束を計算している。また時間積分は単純な陽的オイラー積分によっている。計算格子は４００ｘ３００（１２万点)、６００ｘ５００（３０万点）の２種類について行なった。計算ステップはそれぞれ、２０ループ、１０ループ分について行なっている。図 5(右) に示しているのが計算結果であり、等圧線を表示したものである。

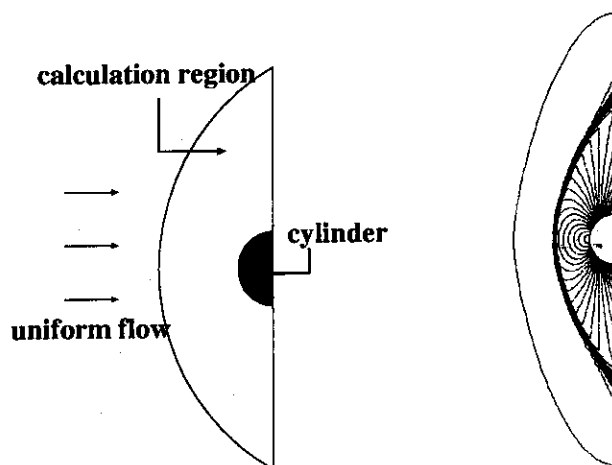

図 5: 左：計算領域、右：計算結果 (等圧力線、一様流のマッハ数 2.0)

実行結果について、ノード数による速度向上と並列化効率をそれぞれ図 6、7に示す。また、経過時間を表 1に示す。

結果はほぼ線形に４８ノードまで上がっていっているのが分かる。並列効率はほぼ９割程度を維持している。これは、主計算の部分だけについて見ている結果で、入出力の部分についてはデータに含まれていない。入出力については３．３節で述べたような並列入出力を行なっている。ノード数が増えるにつれ、扱うデータのサイズは少なくなるので入出力にかかる時間は短くなる。従ってそれらを含んだ経過時間を考えるとさらに効率のよい結果となる。

ることが出来るのである。

6 計算例

第4、5章で設計したコードの性能を IBM SP2 (48ノード)により実行、評価した。扱った問題は円柱に高速流があたるものであり、図5(左)に示すように円柱の風上側のみ計算領域をとって行なった。2次元のオイラー方程式をセル中心の有限体積法により解いており、MUSCL と Roe のリーマンソルバー [5] により流束を計算している。また時間積分は単純な陽的オイラー積分によっている。計算格子は400x300(12万点)、600x500(30万点)の2種類について行なった。計算ステップはそれぞれ、20ループ、10ループ分について行なっている。図5(右)に示しているのが計算結果であり、等圧線を表示したものである。

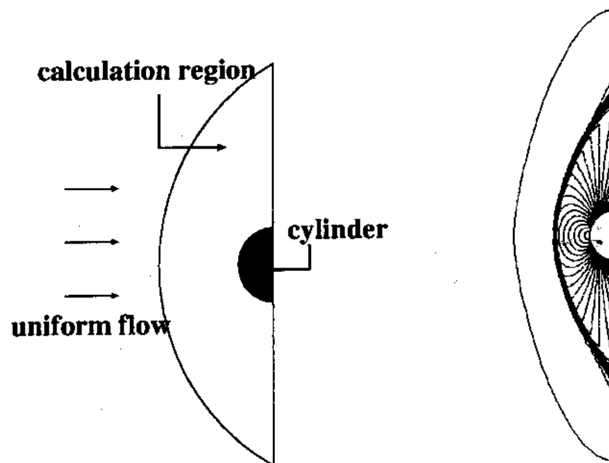

図 5: 左：計算領域、右：計算結果 (等圧力線、一様流のマッハ数 2.0)

実行結果について、ノード数による速度向上と並列化効率をそれぞれ図6、7に示す。また、経過時間を表1に示す。

結果はほぼ線形に48ノードまで上がっていっているのが分かる。並列効率はほぼ9割程度を維持している。これは、主計算の部分だけについて見ている結果で、入出力の部分についてはデータに含まれていない。入出力については3.3節で述べたような並列入出力を行なっている。ノード数が増えるにつれ、扱うデータのサイズは少なくなるので入出力にかかる時間は短くなる。従ってそれらを含んだ経過時間を考えるとさらに効率のよい結果となる。

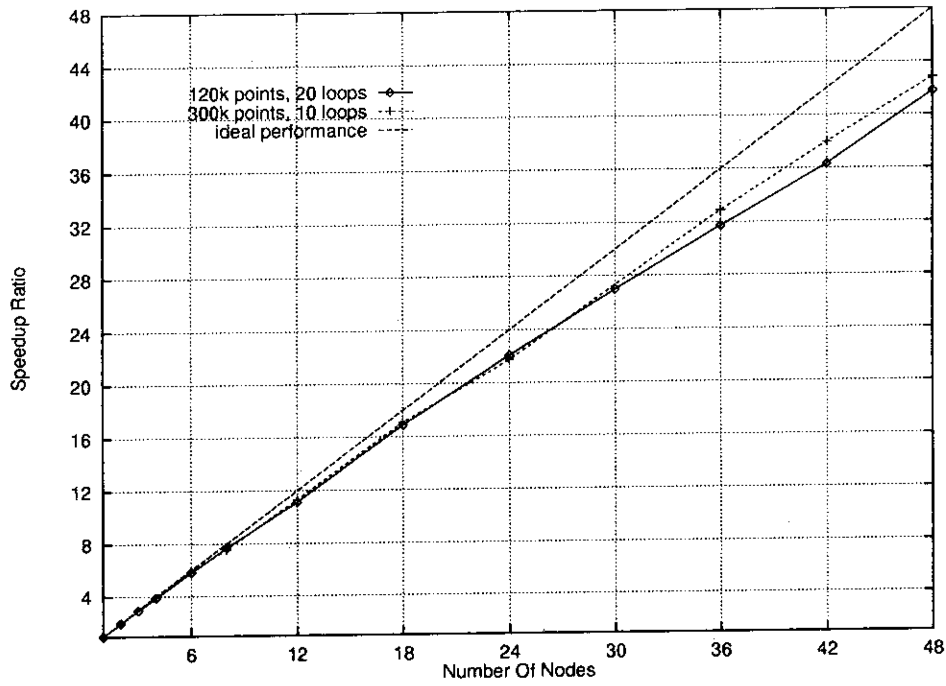

図 6: ノード数による速度向上

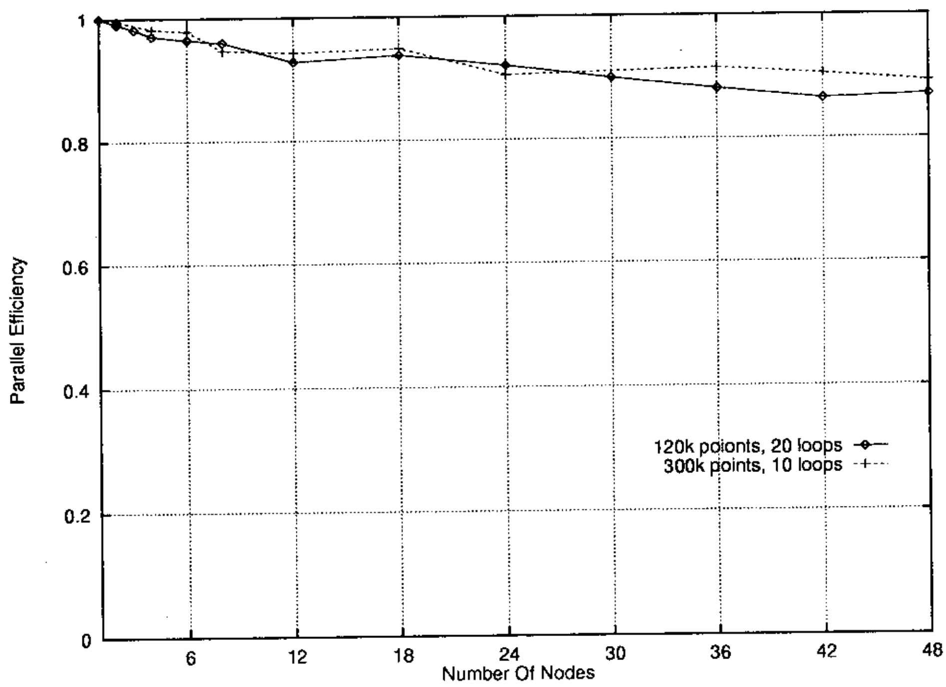

図 7: ノード数による並列効率

表 1: ノード数に対する主計算部分の経過時間

Nodes	*Elaps time of main calculation (sec.)*	
	120k grid points(20 loops)	300k grid points(10 loops)
1	1493.24	1870.26
2	753.84	938.70
4	384.61	476.08
8	194.48	246.83
12	134.10	165.21
18	88.42	109.49
24	67.65	86.08
30	55.38	68.17
36	47.07	56.83
42	41.13	49.28
48	35.77	43.67

7　考察及び結論

オブジェクト指向によるプログラミングの設計思想を提案し、それを圧縮性流体の計算に適用した。SP2 上の性能は良い並列効率が得られることを示している。

ところで、今回取り上げた例題は構造格子上の計算ということで、領域分割の並列計算としては簡単なものである。従って、普通の書き方をしてもこの程度の性能は達成出来るかもしれないし、むしろそのような書き方で細かく性能をあげるように細かく調整することによりさらに性能のよいものにも出来るだろう。しかしながらそのようなものは特定機種で最高の性能を出すようなプログラムとなり、再利用性や異なる環境への互換性は無いだろう。

ここで主張したいことは、ここで提案していることはこの設計によって他よりも優れた並列効率が得られるということではなく、再利用性や互換性を考えた上で、並列化に手間をとられずにプログラムを書きながらもある程度の良い性能が得られるということである。今までの各節の中でも述べたように、この設計指針自体は構造格子に特定したものではない。従って他のデータ構造をベースにしたものでも同様にクラスを構成することが出来る。また、並列化効率を上げる要因とした利点はそこでも生かされるはずである。並列効率を上げるべく並列化のプログラムに非常な手間がかかってしまうのでは、数値計算を行なうという本来の目的からはずれてしまうだろう。数値計算という目的を考えたときに、並列化というのは手段でありそこに時間をとられるのは本意ではないはずである。従って並列化に労力をとられずに良い効率を得られるということは重要であると考える。

さて、効率について述べてきたが実時間はどうであろうか。一般にオブジェクト指向言語はそうでないものに比べてオーバーヘッドがあり実行速度は遅くなる。従って単独の速度では C や FORTRAN で書かれたものには劣るだろう。しかしながらそれらで効率のよいプログラムを書くのには並列化の作業に労力を割かなければならないし、それほど労力をかけなければあまり効率が高くならないだろう。本論文でのやりかたで今回示した程度の効率が得られ、それがさらに多いノード数でも達成されるのであれば、大規模な並列計算においては非常なメリットとなるのではないだろうか。

性能面について述べたが、主目的としては並列コードのプログラミングの労力軽減と互換性の確保であった。労力という意味ではそもそも初めにクラス設計を行ないそれらを実装するところでは並列処理についてももちろん書かなくてはならない。しかしながら並列処理を分離することで並列化プログラミングとしての労力は軽減されると言えるのではないだろうか。それらのクラスがライブラリのようにして用意されたときには計算手法を開発するにせよ、並列化を改良するにせよ従来に比べてどちらも非常に簡単に行なえるようになる。これを、押しすすめるとデータクラスを各種の計算で用いられるデータ型のように提供し、それを用いてコードを記述するだけで並列用のプログラムが書けるというように出来るだろう。従来の並列プログラムを簡単にというアプローチは、並列実行のサブルーチンを提供したり、ライブラリの記述を簡単にしたりというものであった。しかしながら、それらは結局並列化を陽的に意識しなくてはならないというこ

とではそれほどの違いは生じない。また並列に走るサブルーチンをプログラムの任意の場所にいれるのでは並列化のプログラミングという意味ではつぎはぎ的なものになってしまい、かえって全体の並列化を考えるときにプログラム設計が難しくなるかもしれない。並列化を陽に意識しないという意味ではHPFと同じ概念であると言えるかもしれない。しかしながらHPFは言語であるのに対し、本手法はプログラムの設計指針であり、HPFがその言語仕様の枠からはみ出せないのに比べて、様々な対象に適用でき、かつ並列処理に関しても自由に設計を行なえる点で異なっている。

最後に互換性について述べる。C++とMPIという標準的な環境を用いているという意味ではコンパイルをすることによって各計算機でそのまま走らせられることが期待出来る。しかしながら並列計算機というのはさまざまなアーキテクチャが存在し、例えば共有メモリ型のものではMPIが余計なものとなってしまう。しかしながら、ここでも示したように本論文におけるクラス設計では非並列も全く同様に同じクラスを用いて記述出来た。メッセージパッシングによるメモリ分散型の並列環境に対する対応というのは必要なときに行なわれるものであり、共有メモリ型の計算機に対しては非並列のコードとして記述して実行すればよいのである。また、MPIに対応していない計算機環境もあるだろう。この場合もオブジェクト指向によるクラスの設計が助けとなる。メッセージパッシングに関するクラスはインターフェースの部分のみが外のクラスからは利用される部分である。従ってインターフェースさえ一致していればその実装はなんであってもかまわない。今回はMPIを用いたが、同じインターフェースで実装がPVMでも同様に並列プログラムを実行することが出来る。この利点は並列ライブラリの実装のみに関わらず、ここで書かれたクラス全部について言えることである。したがって、一度インターフェースを確立してしまえば、それぞれの部分で様々な手法を用いることが可能である。従って、前の部分でこの設計により並列に手間を取られずにある程度の性能が得られるということを述べたが、各内部の実装をより早い計算が出来るように書き換えて調整するということが行なえる。つまり、並列効率を維持したまま絶対的な計算速度を改善していくということが可能である。この設計は、使用する立場により、簡単に並列プログラムを書く手段として採用したり、性能をあげるために細かい調整に労力を注いだり、という選択を行なえる基盤を与えるものと言える。このような柔軟性はHPFのような『言語』と異なるところであり、またこの設計指針が特定のコードについて述べているのではなく、『概念、パラダイム』であるというひとつのあらわれであると考える。

今回は互換性を示すという意味ではSP2のみでの実行であり実際が伴っていないと言える。他機種での実行は今後示されなければならない項目である。また、本報告書で扱っているのは2次元構造格子という簡単な対象のみであったが、この『設計概念の性能評価』ということでは、3次元や非構造格子などの計算にも適用することが必要である。これらは今後の課題として現在行なっているところである。

参考文献

[1] Bjarne Stroustrup, *'The C++ Programming Language'* Addison-Welsley, 1991

[2] William Gropp, et.al., *'using MPI'* The MIT Press, 1994

[3] Gregory V. Wilson and Paul Lu, *'Parallel Programming Using C++'*, The MIT Press, 1996

[4] Erich Gamma, et.al., *'Design Patterns'*, Addison-Wesley, 1995

[5] Charles Hirsch, *'Numerical Computation of Internal and External Flows'*, John Wiley & Sons, 1988

[6] WWW page of Diffpack, http://www.oslo.sintef.no/avd/33/3340/diffpack/

[7] WWW page of Physica, http://www.gre.ac.uk/ physica/