An Object-Oriented Framework for Scientific Computations

- advantages in parallel computational fluid dynamics -

科学計算のためのオブジェクト指向フレームワーク

- 並列流体計算における利点 -

Takashi Ohta

太田 高志

Abstract

Parallel computing has become a general approach for scientific computing, and is considered one of the major computing environments of the future. Though appreciating its performance, the current situation seems to lack an appropriate methodology and philosophy that can take real advantage of parallel computing. The algorithm of a parallel computing scheme is often a merely parallelized version of what was originally designed for the sequential computation, and so are the related processes like grid generation and visualization.

Having the above situation as a background, it is definitely desired to establish a new paradigm for parallel computing, and this is the objective of this work. We propose a programming design and an object-oriented framework for building calculation codes. The framework is also expected to cover the entire analyzing process, since it is necessary to redesign the whole process as it fits to the new paradigm brought by the parallel computing. The framework forces a code to have a certain program structure, which benefits various aspects of the parallel computing and the other analyzing processes. In addition, the framework offers programming templates for the parts of CFD codes and acts as an infrastructure. Were many codes built based on a same designing concept, these codes could be used interchangeably, and become a sharable asset. The object-oriented features greatly help to realize this aspect.

This thesis explains the programming paradigm that we propose, and the object oriented programming framework that is designed to realize the concept. Usages of the framework and the merits of the design are also presented. The testing calculations show that this approach works very well with the parallel computing, and the examples demonstrate that the framework benefits in constructing an integrated system in a seamless way.

The object-oriented approach brought great changes in programming paradigms in various application fields, and benefits greatly in many aspects. However, it has not brought significant influences into scientific computing so far, and CFD is not an exception. Therefore, this work also aims to examine the benefits that an object-oriented approach can bring, especially to parallel CFD.

Contents

List of Figures

Chapter 1

INTRODUCTION

1.1 Parallel CFD

The computational approach has gained a major position in analyzing nonlinear problems ever since so-called supercomputers came into the scene. In aeronautics, CFD (Computational Fluid Dynamics) plays a significant role in analyzing an aerodynamic phenomenon. The approach is especially useful when the phenomenon cannot be examined by experimental approach. It also contributes the theoretical research by offering supporting calculations. As the importance is increased in the research process, CFD is expected to deal with more practical problems. In order to meet such requirements, and with the progress of computer hardware and numerical algorithms, the required computing power both in the calculation performance and the memory size are constantly increasing. When we consider a CFD case, even a calculation with 10,000,000 grid points is not enough for the detailed analysis around a complete aircraft configuration. In addition, coupled simulation of different physical systems is becoming a realistic target [1, 2]. Along with such demands for large-scale calculations, the required computing performance will be enormous.

In order to meet these requirements, parallel computing becomes a practical candidate for such large-scale calculations because of its performance and the memory size, especially when considering a memory-distributed architecture. Parallel computers will indeed meet such criteria, and will be one of the major platforms for scientific computations in the near future. The situation seems favorable to parallel computing in that respect. However, the coding requires special knowledge and techniques for parallel computing. Acquiring such new concepts and knowledge requires a considerable effort and that would not be an ideal situation for a researcher who only wants a powerful computing environment. Though various tuning techniques for the parallelization of specific code fragments are often mentioned, it seems that no solid methodology exists regarding designing parallel code, nor is there a philosophy that takes real advantage of parallel computing.

The main problem regarding the parallel computation is that writing a parallel program is often referred to as difficult and tiresome. This might originate in such facts as the following: With a message passing library [3, 4], the procedures that achieve parallel computation, such as data decomposition and distribution, data transfer, synchronization of processes, and data gathering, must be written explicitly in detail. It leads to unexpected errors, incorrect calculation results, or deadlocks of multiple processes, unless these procedures are designed carefully. Even with a so-called parallel compiler that claims to parallelize code automatically, inserting the directive lines, which indicate where and often how operations are to be parallelized is indispensable in order to obtain decent performance. The directives generally lack portability, so that these lines must be rewritten with a different machine or a different compiler. Besides these tasks, often the algorithm should be redesigned as to expose parallelism explicitly. These programming constraints force a researcher to do extra labor in a field other than the area of his actual interest, unless his research area is parallelization itself.

Such additional labor for parallelization would be rewarded if the objective is to pursue the peak performance for a particular algorithm regarding a particular problem, and use it repetitively in the same computing environment. However, it is often required to modify code in the course of research, along with changes of the algorithms and the conditions. Various schemes will be introduced for verification, and that often leads to rewriting of an entire program. A structural programming approach would partly help such modifications by making each procedure a module. However, parallelization harms the modularity since the program lines for parallel computing are inserted intertwined with the native code. This also harms the code's reusability, and even the portability on different platforms, since it is tightly coupled with a certain parallel environment. Since parallel computing requires special knowledge, the programming task is often done by an expert in parallel computing. Though an expert knows a lot about the parallelization, he must learn the algorithm and the details of a certain application in order to understand where and how to modify the code. This process is inefficient and costs the expert a great deal of extra labor. At the same time, it is often observed that the parallelization task is carried out by searching for the parallelizable parts without knowing what these parts are responsible for or what meaning the code actually has. This makes parallel programming a mere mechanical task but not a research objective. This is observed as the approach commonly employed for developing a parallel code.

1.2 Objectives and History

When we consider the above situation of CFD and its significance in aeronautics, it is definitely desired to establish a new paradigm for parallel computing in order to improve the situation, especially as regards to the design of parallel code. At the same time, since parallel computing aims at large-scale calculations, the pre- and post-processes should be prepared appropriately in order to deal with such a large volume of data. Users should be well aware that parallel computing affects not only the programming for the main calculations but also these supporting processes. The processes that are originally designed for a sequential computation, without consideration of dealing with a large scale problem and parallelized data, are inadequate for parallel computation in many respects. Therefore, the objective of this work is to propose a paradigm for programming design, and construct a programming framework that can serve as an infrastructure for the whole CFD process using parallel computing.

As for the programming approaches that are currently practiced, we decided that the difficulties like the ones mentioned in the former section are owing to the *parallelization* process itself. We regard this approach as the main cause in preventing a generalized methodology for parallel programming from being devised, since every parallelization tends to be specific to particular code, making it a mere mechanical task but not a creative art. Furthermore, these parallelized codes often lack the reusability and portability within various environments. Aside from these problems, this approach requires users to develop two different programs, one for a sequential environment for the beginning, and one for parallel calculation after the sequential version is completed. Furthermore, another program must be prepared whenever a different parallel environment is used. With this approach, the programming lines for the actual numerical algorithm and the ones for the parallelization are tightly coupled. This mixed-up coding harms the modularity, portability, and reusability of the code. Therefore, it would be beneficial if the numerical and parallelization algorithms are separated as distinct modules.

Taking this idea of separating the parallelization procedures from the writing of the numerical algorithm as the basic concept, we have been working with an object-oriented approach for parallel scientific computation [5, 6, 7, 8]. The research aims to improve the solutions by proposing an alternative programming paradigm. The approach targets CFD algorithms that are parallelized using the data parallelism concept, such as the domain decomposition technique. This approach guides the code to be written so that its numerical scheme and parallelization procedures are separated and realized as different classes, the former as a solver class and the latter as a data class. The procedure for the communication between the parallel-running processes is encapsulated and concealed from the outside of the data class. This makes the data class look as if it is a mere data structure to the solver. Though the concept is as simple as that, it benefits various aspects in the process of developing parallel scientific codes, and achieves good parallel performance [5]. Though the concept has proven itself, developing code according to this style requires other knowledge of this approach rather than the knowledge of specific parallelization techniques that is needed using other parallelization approaches. A programming framework that guides code into compliance to this approach is necessary for easily receiving the maximum benefits. In order to design such a framework, the design principle is somewhat modified by my earlier work mentioned above, to allow more flexibility and reusability. This thesis is, therefore, not only describing the programming concept, but also introducing the design of the object-oriented framework.

The framework is designed to offer a certain structure by the combination of its classes. Since it separates the numerical and the parallelization procedures, and covers the most central parts needed for composing code, parallel program can be written with just the same amount of labor required for developing a program for sequential computing. At the same time the framework supports developing parallel code, it also makes the program module for the main calculation that is portable even for use with a sequential computation environment. The coding for a calculation scheme can be used for parallel computation when it is assembled with the modules for parallel computing. For realizing sequential computation with the same code, it can be done by removing these modules for parallel computing and there is no need to modify the coding for the scheme itself. The other merit is that the framework can offer itself as an infrastructure for building CFD codes, regardless of whether parallel or sequential calculation will be used. Such an infrastructure acts not only for the convenience in programming, but benefits more in sharing research idea as programming achievements. Such a framework will contribute in making the numerical approach to be processed more smoothly and systematically. By preparing the framework, it is expected that the researchers can concentrate in their main concerns such as numerical schemes and physics, and are not disturbed by the detail of the programming. Thus, it is important for the CFD research to have a framework like the one explained above. The target of the framework in this thesis is CFD programs with FDM or FVM, that are parallelized by techniques that decompose the whole calculation load globally, as in the domain decomposition technique. Though the base concept can be expanded to broader fields of scientific computations and parallelization approaches, this thesis limits its focus to the above cases. On the contrary, what this framework does not intend to achieve is pursuing the peak performance of a program. Though the framework is not intentionally designed to spoil that aspect, and though actually a trial work to improve the performance is included, it is not a main objective of this work. This programming paradigm and the framework are not intended for achieving the most superior performance possible. It may be better to apply the parallelization process and the fine tuning in the traditional way if the goal is pursuing the peak performance of a certain program or an algorithm on a certain machine. The objective here is at variance to such attitude. This approach is proposed for the research work where the changes in schemes, conditions, and computing environments occur often. It is also considered an aid for the search for genuine parallel algorithms. As the design concept of the framework, it does not aim to be a general, low-level class library for parallel programming. Such attempts only provide other elements for programming in the aforementioned environments, such as message passing libraries. The framework proposed here is not for a general use, but for a specific application, CFD in this case, and supplies the components that are sufficient to build a complete program. This characteristic of the framework allows researchers to concentrate only on a CFD scheme, without having to focus on the parallelization task.

1.3 State of the Art

There are many reports on CFD with parallel computing [9, 10], and these seem to be categorized into several groups. Some use parallel computing for carrying out large-scale calculations [11, 12, 13]. These reports contribute to promoting parallel computing as a practical platform by showing its performance and results. Research in parallel algorithms forms another major group, such as for matrix solvers, flow solvers, grid generation and so forth. There are a great deal of research on parallelization for basic numerical algorithms like linear algebra [14]. One of the more interesting examples involves a flow solver for an unstructured grid, explaining how to make the algorithm efficient for data access and communication [15]. Much work has been carried out in domain decomposition techniques and load balancing, especially with unstructured grids [16, 17]. Adjusting the calculation loads dynamically between parallel processes with an adaptive grid/mesh technique is a typical theme in this category. These projects are each dedicated to a specific algorithm of their own and each addresses a specific aspect of parallel computing. There are several attempts for applying an object-oriented approach to scientific computing, outside of the domain of CFD [18, 19, 20]. Many are dedicated to designing a new parallel language by extending C++ [21, 22], or a class library [23, 24]. These approaches aim to provide easier and more convenient programming elements for parallel computing, by taking advantage of the object-oriented features. There is also intensive work on building a framework for scientific computing, such as POOMA [25, 26]. This framework covers a broad range of aspects of scientific computing including parallel computing. It implements many abstract concepts of scientific computation as classes. These projects have benefits for parallel programming by providing convenient and even intelligent tools for the parallelization of codes, such as an array class that supports parallel execution by itself, like the array in HPF (High Performance FORTRAN). Among the other projects mentioned above, POOMA is the most similar to this work in the concept and approach. Though the characteristic as an object-oriented framework is similar, its intention and concept is different. POOMA intends to support scientific applications generally. Thus, the classes tend to be at higher levels of abstraction. As a result, these classes behave rather like fundamental parts for developing a program. Therefore, writing a CFD code with POOMA requires a deep understanding of POOMA's concepts, and a researcher must consider how to build the code with the newly introduced parts of POOMA.

Our approach puts more emphasis in defining a program's entire structure rather than the implementation of the parts, and the object-oriented feature is mainly used for this purpose, while the programming details are rather untouched. The uniqueness of the approach described in this thesis, compared with other researches, can be explained as follows. First, the difference from the various attempts in parallel computing with CFD is that each of these deals with solely a specific problem of each own, while this work is proposing a general concept towards the use of parallel programming. Second, other projects with object-oriented approaches are intended to provide ways that are more convenient, or tools for programming, whether these are done by providing a new language or a library. Our approach does not tread a similar path as providing the components to be used in programming, but it offers a design of the program's entire structure, which is constructed by the combination of the classes of the framework. As a summary, other related works can be regarded as the works that have the parallelization approach as their common basic principle or focus. These works are variants of the parallelization of certain codes, or improved ways or tools designed by the parallelization principle. Therefore, when developing CFD code by any of these approaches, there is no difference in the design of the parallel code. It is done by modifying the sequential code by using the language or the library as the parallelizing tool. The differences of these approaches are the differences of the kinds of tools they offer for that purpose. We do not take this *parallelization* approach but propose to design parallel code from the start. Our approach and the framework are designed to do parallel programming in that manner. At the same time, we decided to have the classes as coding templates corresponding to the concepts that appeared in typical CFD programs, such as a grid system, flow variables, boundary conditions and so forth. This treatment makes programming significantly easier, and also makes the framework a common infrastructure by which the modules are developed as compatible. This feature is another merit that this approach can bring to CFD programming.

Chapter 2

BASIC CONCEPTS

The most common way for preparing a parallel program is to parallelize an existing sequential program. Even when writing a new program from scratch, we generally follow the process of developing a sequential version first, then parallelize it. With the parallelization approach, every necessary procedure for parallel computation is inserted into an existing sequential program as they are intertwined with the original program, as shown in Fig. 2.1. Consequently, modified program is obtained as a parallelized program. This approach seems to have the following problems. First of all, it needs parallelization task to modify the sequential program. This means

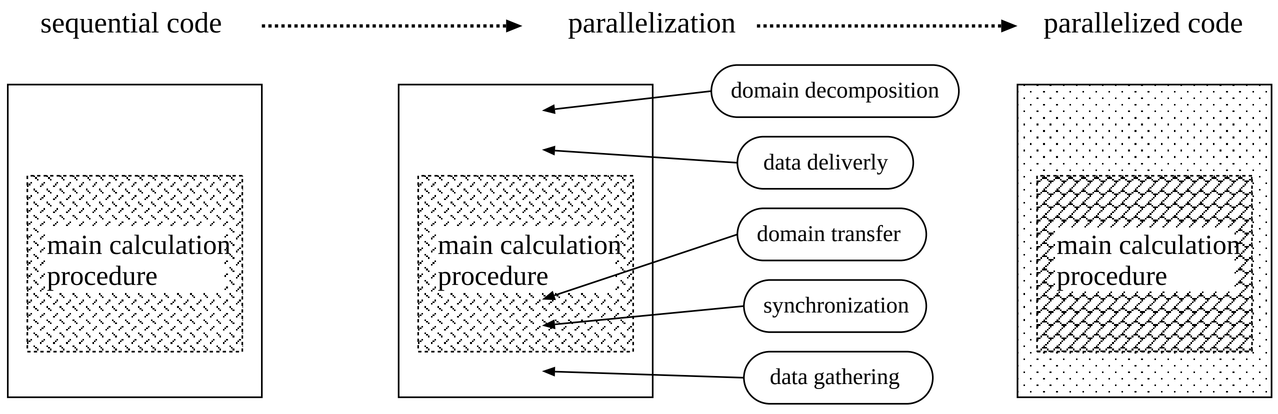


Figure 2.1: Traditional *parallelization* approach

that an additional programming effort is required for parallelization. In addition, knowledge of parallel computation in general and of a particular environment such as a message passing library are also required. Since a parallelization task tends to be specific to each program, it is difficult to reuse the parallelization effort for other programs. Whenever the necessity of parallelizing a different program has arisen, another parallelization task must be begun from the beginning. Second, two versions are produced for the same calculation, one for the sequential, and the other for the parallel execution. Hence, when some modification is made to the calculation algorithm, two programs should be simultaneously updated if both of them are in use. It also requires the consistency of the two versions. Third, and the most serious problem is that the algorithm was designed for a sequential computation. A parallelization of a loop or any part of the program does not alter the algorithm as a parallel algorithm, but only makes it one that is runnable in parallel. A carefully parallelized program can gain enough speedup in comparison with its sequential version, but the approach interferes with the possibility of devising a superior fundamentally parallel algorithm.

With such problems as a background, the following arguments are naturally deduced regarding more desired features for parallel programming. To begin with, it is preferable not to parallelize a program, but to design a program as parallel program from the start. However, at the same time, it is desired that the same code can run on both sequential and parallel environments without preparing two different versions for each. Such a unified code should not be achieved by switching between two algorithms with an if-clause, since that is not different from preparing two separated programs. In the course of this argument, the expected design of the algorithm has emerged as one that is originally designed for parallel computing, and includes the sequential execution case as a special case when it runs as one process. In addition, it is preferable that the procedures necessary for parallel execution, such as the domain decomposition and data transfers, or anything for parallelization, are designed as separated and reusable modules rather than that these are embedded directly into the program. In order to realize the requirements above, we decide to introduce the following principles in designing programs. First, let the main calculation algorithm be entirely parallel by assuming parallel execution from the start. Second, the procedures required for parallel execution should be independent modules separated from the program of the main algorithm. This condition leaves the program for the main algorithm intact and separated from programming for parallelization procedures such as data transfer, though the algorithm itself is prepared for parallelism. Third, the module for the variables and the data should be independent from any of the calculation procedures. These principles enable the same programming for the main calculation to run on both parallel and sequential environments. The program runs in parallel when it is assembled with the modules for parallel execution, and runs sequentially without them.

We explain how these principles are realized, by taking domain decomposition as an example case. The domain decomposition technique generally requires procedures for a decomposing the region, for regional information management, for data transfer by message passing and for the synchronization of multiple processes at appropriate time. In order to isolate the main algorithm from these procedures, we regard the data transfer procedure as one of the boundary conditions, together with the other mathematical conditions. This treatment lets the calculation algorithm not noticing explicitly the existence of the procedure for data transfer. If a unified interface can be defined to invoke the procedures for applying various boundary conditions, including the one for data transfer, the main algorithm is abstracted so it manages its responsible region and the boundaries only. The synchronization is automatically taken when all the data transfers are completed, in every time applying the boundary conditions. Now the domain decomposition procedure is regarded as a kind of pre-process of the main calculation. Here, we define a field as a module that groups the grid data, the variables, and the boundary conditions. The domain decomposition is regarded as a process that prepares an appropriate field for parallel computation, which consists of a decomposed region and the boundary conditions for it. A sequential calculation is naturally included as the case when the calculation field is equal to the entire region. When this field module is designed not being specific to a particular procedure, procedures can be implemented as separated program modules. This means that the program for main calculating algorithm and the program for a parallelization procedure can be implemented as separated programs. These modules work cooperatively by taking the field module as a medium of communication.

Now let us generalize the above approach to a broader concept so it can cover cases other than the use of the domain decomposition technique. With any scientific computing applications, it is significantly advantageous to design the algorithm as it has a parallelism as the whole. That parallelism may require recomposition, redistribution, and transfers of the data, during the calculation. If these kinds of procedures for parallelization can be abstracted together with a procedure used in sequential computation so that both of them are represented by a unified concept, it is possible to create one procedure that can cover a certain part of both parallel and sequential calculations. Treating both the data transfer and the mathematical boundary conditions by a unified concept is an example of such an abstraction. Once such an algorithm can be designed, the distribution of the calculation loads has became a pre-process of the main calculation. Now, pre-process is not merely a process of grid generation, as the word often implies, but is a process that prepares whole information and variables required for a calculation, as they suit for the algorithm's abstraction. In the above case, the field module is expected to have a calculation region and the information of the boundary processes. This structure of the field, together with the algorithm of a unified concept, can produce a program that is available to both parallel and sequential calculations. Now, pre-process is not merely a process of grid generation, as the word often implies. It is regarded as a process that prepares all information and variables, to form a proper field module for the calculation. It includes the process of the decomposition of the calculation load when it is for parallel calculation. The program's structure by this approach is depicted as Fig. 2.2. The pre-process for sequential calculation prepares a field of the data and variables of the entire region, while the process for parallel calculation generates a field for the decomposed region. Though the instances of the field modules are different, their structure are the same, therefore, can be handled by the same calculation program.

It should be understood in the course of the explanation that the strategy employed here is not aiming to help the parallelization task. The objective is to offer a certain program structure, using which a parallel program is naturally produced. The problems mentioned at the beginning of this chapter, such as the parallelization process itself, the working costs for parallelization, the consistency of two versions of programs for parallel and sequential calculations, and the desire for a parallel algorithm, are all covered by this approach. What is significant with this approach is to find a proper abstraction for the algorithm, and make a proper design for the field module. These are the essentials of the approach.

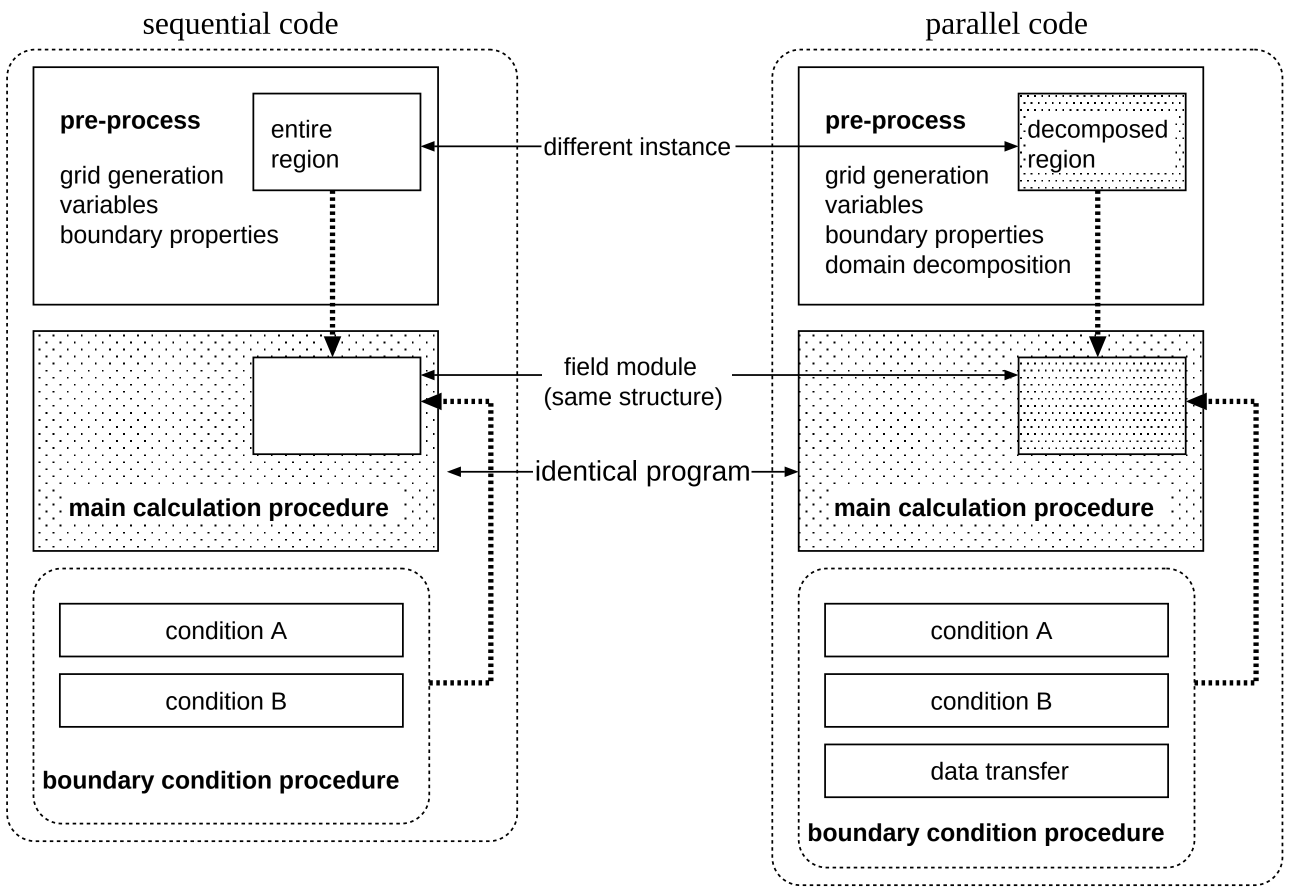


Figure 2.2: Architecture of sequential and parallel programs by this approach

Chapter 3

FRAMEWORK

3.1 Necessity of Framework

In the former section, an approach for the programming of parallel CFD code is proposed. However, without any support, it would be difficult to develop a program so that it complies to the concept. Researchers must analyze the target algorithm throughly and find a proper abstraction of it, since the concept does not guide the programming for each particular case. In order to receive all the benefits of our approach, modules should be well-designed especially when considering the reusability. It would be harder than typical parallelization approach. Therefore, a programming framework that offers the core modules to compose the fundamental architecture, is highly desired.

A framework is also desired for forming a common base to which programs are designed. By serving as the common infrastructure, it assures compatibility of every module. It is observed often that researchers develop their own programs not considering the compatibility or reusability of the code. This makes it difficult for other researchers to examine a new scheme or calculation results. The researcher who wants to use a new scheme must write the program by consulting research papers. It is even observed that an individual researcher rewrites his own code completely in order to employ a new scheme or conditions. Thus, it results that no programming work previously done is effectively used. This happens because there is not a standardized guide for the programming. If such a standard comes to exist, and many fragments of program are supplied, researchers can easily reuse other's achievements for writing his own program. That situation would reduce the individual's programming effort to complete a program from scratch, so one can concentrate to an important task like designing of numerical algorithm. A need for a framework is found here again.

Therefore, we design the framework that supports programming of CFD code, by basing on the concept discussed in the former chapter. This work is carried out for making the framework a sample to examine the benefits and drawbacks of such arrangement.

3.2 Principle for Framework Design

We employ object-oriented approach since we think the concept is most naturally realized by it. We set following principles in designing the framework.

First, classes will not form a low-functionality, general-purpose library for parallelization. The framework aims to provide an entire architecture for programming, but not to provide parts for parallelization. Classes are prepared so that they correspond to the typical subjects that appear in CFD code, such as grid and flow variables. The program's architecture is defined by the combination of the classes, and object-oriented methodology is fully used here. Writing a CFD algorithms, however, is not restricted to any programming paradigm. For parallel computing, detailed preparations for parallel execution are offered by the framework. Therefore, researchers are not disturbed by the programming for parallelization, and can concentrate in developing a CFD algorithm itself.

Second, the classes should be well-designed to offer flexibility in forming a program. At the same time the framework does not aim a general purpose library, it is not prepared for composing a particular program of fixed structure. The flexibility should be achieved by a flexible composition of the architectures, but not by assigning a variety of functions to each individual class.

Third, the classes of the framework are prepared in a layered structure. The classes of the level of higher concept are built on top of the lower classes. Only the lowest classes access directly to low-level libraries or system functions. This setting has advantages in two aspects. The difference of the system environment like message passing library for parallelization can be absorbed by the implementation of lower classes. This achieves the portability of the classes of higher level. In addition, though the framework is designed for a particular research field, that is for CFD in this case, lower classes can be general enough to provide the base of a framework for different research fields. A framework for a different scientific calculation would be built by replacing the higher classes only.

Last, the classes corresponding to the subjects in CFD code are provided as the templates for programming. A specific implementation will be written into a proper place of the program by using such a template. A template also guides the interfaces that should be prepared and used in that implementation. These templates impose actual restrictions in the implementation of programs, being prepared as abstract classes for inheritance.

3.3 Framework

3.3.1 Overview

The framework consists of many modules. These modules will be cited as ”class” in the following sections, corresponding to object-oriented concept. The word ”object” is used for representing a specific instance of class.

The core structure of the framework is given by the relations of procedure classes and data class. This relations are depicted as Fig. 3.1. The central class of the structure is for data, which is represented as ”Field” in the figure. The procedures are also realized as classes. Each procedure class does its work by processing an object of Field class.

For example, the solver procedure applies main calculation and boundary conditions to a

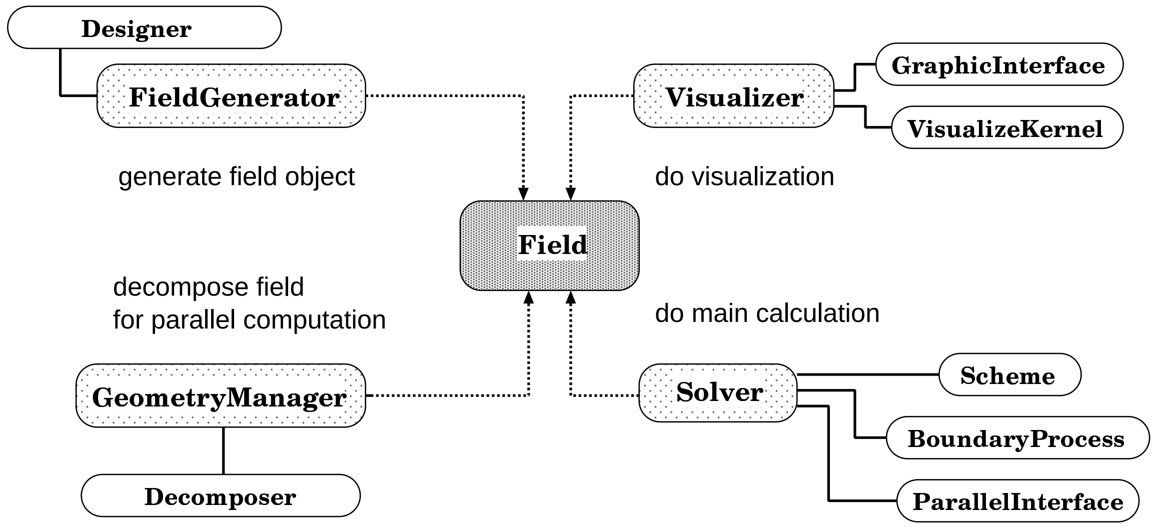


Figure 3.1: Conceptual overview of the framework's structure

data object, adopting the classes of scheme, boundary processes, and interface to a parallel environment. "Generator" generates an object of "Field" class, by giving specific data to it. "Geometry Manager" is responsible to the decomposition of the calculation domain for parallel computing. And, "Visualizer" does visualization with data objects of calculation results.

The works of these procedures are isolated so that they do not affect the other procedure directly. The effect of each procedure can be observed only as a transition of a data object. Therefore, this arrangement realizes a separation of calculation algorithm and parallelization detail, as they are implemented in separated program segments. Thus, this arrangement offers a core structure that suffices the concept of our approach.

The framework supports building such structure by the relation of classes. A particular program is built by choosing and composing proper classes for the purpose. Figure 3.1 depicts only the conceptual overview of the core structure. The entire framework consists of several sub-framework, each of which covers an individual facet of CFD program or parallel computation. These are explained in the following sections.

3.3.2 Field

Field class is the central structure of the framework. An proper design of the class is significant to the realization of the framework. As the data module for CFD, it is expected to have grid data and flow variables. Since we demand a data module has all the necessary data and information to carry out calculation, information on boundaries of a region is also chosen as a member of Field class. The information on a boundary should contain the region information and the process applied to it. A region information consists of the information such as range and position for specifying a boundary.

There are two subjects that should be considered with parallel computation. These are the treatment of data transfer and domain decomposition. For unifying the procedures processed in both parallel and sequential calculation, data transfer is treated as one of the boundary processes. For data transfer, the information on corresponding region is given in a region information. There is no special treatment to a Field object concerning domain decomposition, since the structure of it remains the same even after the decomposition. Only difference brought by the decomposition is the difference of the type of boundary process; now the Field object has data transfer as its boundary process. These data and information provide the enough ingredient for carrying out a CFD calculation. An object of Field class also becomes the target of the procedures like grid generation and visualization.

There are cases where the entire region is composed of several domains. This happens when the target region has a complicated geometry that is difficult to be realized by single structured grid. It is often observed that a program is developed particularly to a certain field composition. In such cases, the program must be rewritten for a different composition of the field, though a same numerical algorithm is employed. This framework offers a function of handling both single and composite regions by one program. The arrangement is shown by the diagram[27, 28] in Fig. 3.2. As both SingleField class and CompositeField class are derived from the same ancestor, Field class, they can be accessed by the same interface to a Field object. This enables writing one program that handles various composition of the field.

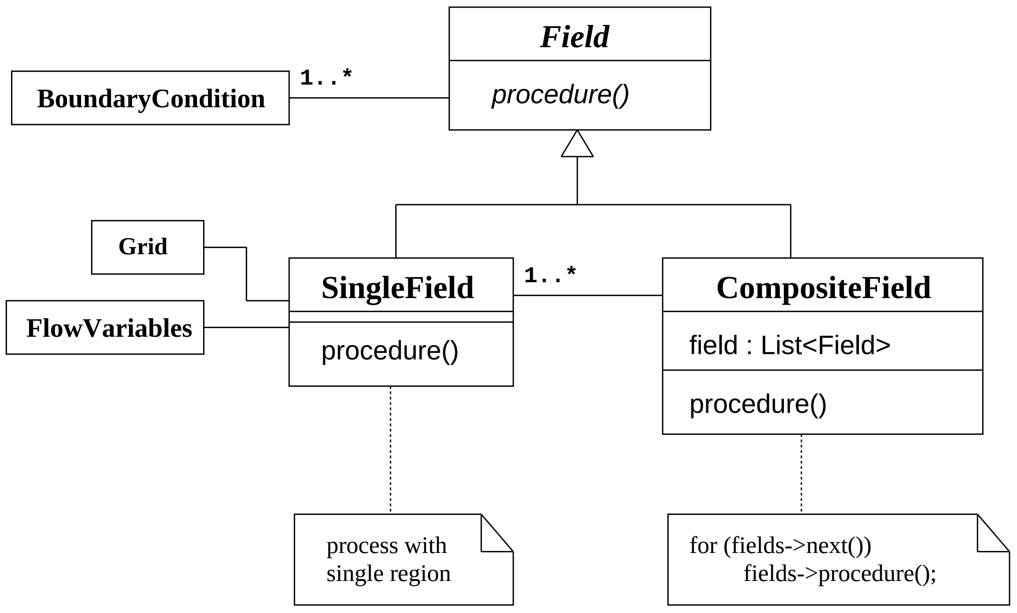


Figure 3.2: Field class design

Example implementations of Field class and SingleField class are presented as follows.

```
class Field {
  public:
    bool isComposite();


    virtual int  numberOfField() = 0;
    virtual int  numberOfRegion() = 0;
    virtual void accept(FieldUser *user) = 0;


    int         numberOfCondition();
    void        setCondition(BoundaryProcess* c);
    Conditions*  getCondition();


  protected:
    bool       compoiste;
    Condiitions  conditions;
};



class SingleField : public Field {
  public:
    virtual Grid* generateGrid(const int& sz1,
             const int& sz2, const int& sz3) = 0;
    virtual Grid* generateGrid(const int& s1, const int& e1,
             const int& s2, const int& e2,
             const int& s3, const int& e3) = 0;


    virtual VariableField* generateVarfield(const int& sz1,
             const int& sz2, const int& sz3) = 0;
    virtual VariableField* generateVarfield(const int& s1, const int& e1,
             const int& s2, const int& e2,
             const int& s3, const int& e3) = 0;


    Grid*        getGrid();
    VariableField* getVarfield();


    int  numberOfField();
    int  numberOfRegion();
    void accept(FieldUser *user);
  protected:
    Grid          *grid;
    VariableField  *vfld;
};
```

Usually, these Field classes are not used directly in a program. A Field class for a particular calculation is prepared by inheriting any of these classes. In that inherited class, specific implementations to handle particular grid system and flow variables are made within the defined interfaces.

3.3.3 Field User

Since most of procedures are expected to use a Field object, it is convenient to prepare global methods for accessing. These methods will offer the way of accessing the data and the variables that are encapsulated in Field class. In the former section, several classes belonging to the Field category are introduced, for managing the fields of different compositions. In order to handle these classes in a unified manner, the difference of the treatments must be handled within the accessing interfaces. FieldUser class is designed to provide such interfaces. Once FieldUser is accepted by a Field object, it generates pointers for accessing the data members of a Field object. Thus, a FieldUser object and its derived classes can access the data members of Field, and can process their duty by using these data.

Just like Field class's case described in the former section, FieldUser is not used directly. It acts as a programming template for implementing procedures as independent classes. A procedure that is going to use a Field object will be realized as a class that inherits FieldUser (Fig. 3.3). Specific work of a certain procedure is implemented in each derived class.

The following is a sample code showing how a Field object accept a FieldUser.

```
Field *field = new CFD_Field;

Solver *solver = new CFD_Solver;

field->accept(solver);
```

First, instances of Field and Solver classes are generated. Solver is a class derived from FieldUser. It appears as CFD_Solver in particular in this example. Then the field object is associated to the solver object, by being accepted by it. One same field object can accpet more than one FieldUser.

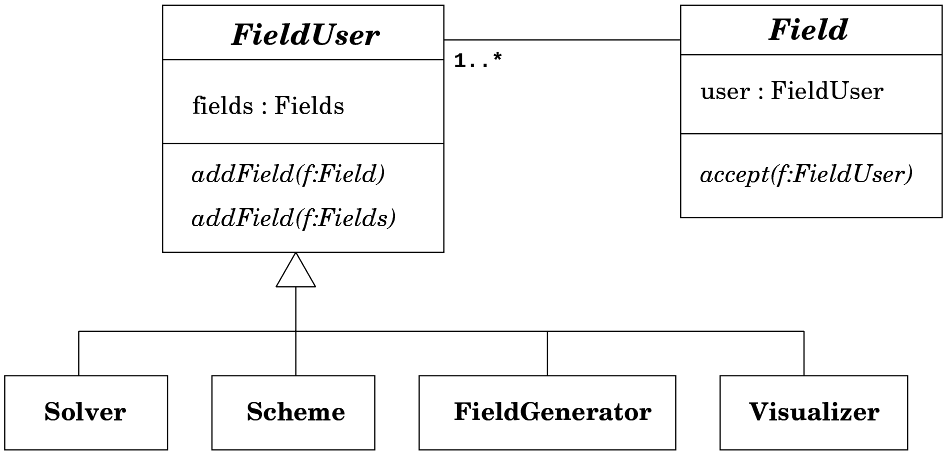


Figure 3.3: Mechanism for using Field class

3.3.4 Classes for Calculation

Two classes, Solver and Scheme, are prepared for composing a main calculation procedure.

Scheme is the class where specific calculation algorithm is implemented. It is designed to handle only one single field, since this treatment makes the implementation easier. Solver employs a Scheme class that matches to the kind of field object. If the target object has a composite region, Solver generates a Scheme object for each Field object of the single region that is included in it, and binds every pair of the Scheme and Field objects (Fig. 3.4).

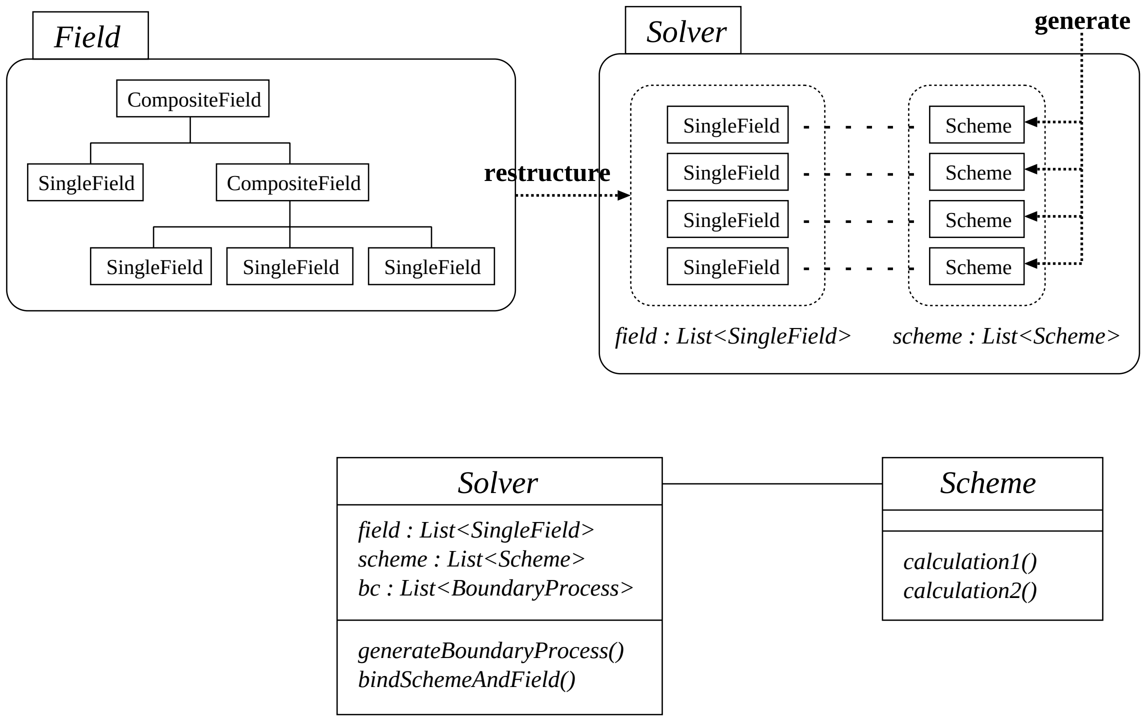


Figure 3.4: Classes for main calculation procedure

Solver also generates the objects for boundary processes, according to the information provided by a Field object. Classes for the boundary processes are derived from the same ancestor and inherit one same interface to apply their processes (Fig. 3.5). Therefore, Solver does not need knowing the detail of each of them in order to use various boundary processes. This is one of the advantages of the object-oriented programming called polymorphism.

The following program fragment is an example to show how the boundary processes are used.

```
/*-----------------------------------------------
  setting of boundary process objects
 -----------------------------------------------*/
typedef map<int,BoundaryProcess*> Processes;
```

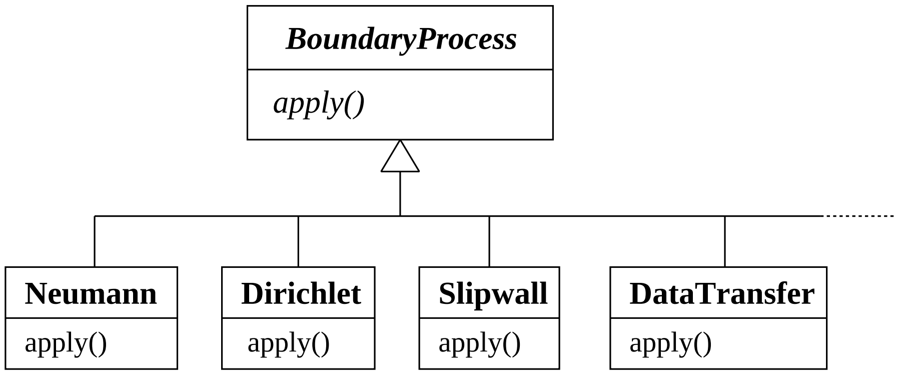


Figure 3.5: Classes for bundary process

```
Processes bndprc;


Neumann     *bp1 = new Neumann;
Dirichlet    *bp2 = new Dirichlet;
Slipwall     *bp3 = new Slipwall;
DataTransfer *bp4 = new DataTransfer;


bndprc.insert(Processes::value_type(0,bp1));
bndprc.insert(Processes::value_type(1,bp2));
bndprc.insert(Processes::value_type(2,bp3));
bndprc.insert(Processes::value_type(3,bp4));


/*---------------------------------------------------
  usage of boundary process objects
 ---------------------------------------------------*/
Processes::iterator itr;
for (itr=bndprc.begin(); itr!=bndprc.end(); itr++) {
  BoundaryProcess *bp = itr->second;
```

```
    bp->apply();
  }
```

Solver class performs the calculation process by evoking proper methods of Scheme class. It is also responsible to the setting of necessary parameters for calculation.

Both Solver and Scheme are prepared as the inheritances of FieldUser class. For a particular application program, responsible classes will be prepared by inheriting each of Solver and Scheme. For example, IncSolver and IncScheme would be derived for a calculation of incompressible flow problem.

3.3.5 Classes for Parallel Execution

Classes provided for parallel computation can be categorized into three groups. These groups are parallel environment, domain decomposition, and data transfer.

We decided to use existing environment like message passing library to realize parallel computation. There are several such libraries that are widely used, such as MPI and PVM. We do not want our framework and programs to depend on one particular library, since it harms portability. Therefore, we prepare abstracted interfaces in order to absorb the different usage of different libraries. A specific interface class is prepared for a particular library, by inheriting the abstract class where these abstracted interfaces are defined. This arrangement is shown in Fig. 3.6. A program is written by using the interfaces offered by the abstract class, instead of using a particular library directly. Using a different library will be done only by changing the parallel interface object, and this change will not affect the program since every interface class has the same interfaces.

The pre-process for parallel computing includes the procedure for domain decomposition, and classes are prepared for it. Two sub-procedures are carried out in this stage. One of the procedures is the decomposition of the region, and the other is the rearrangement of the boundaries. Sub-framework for this procedure is depicted as Fig. 3.7. All the decomposition task

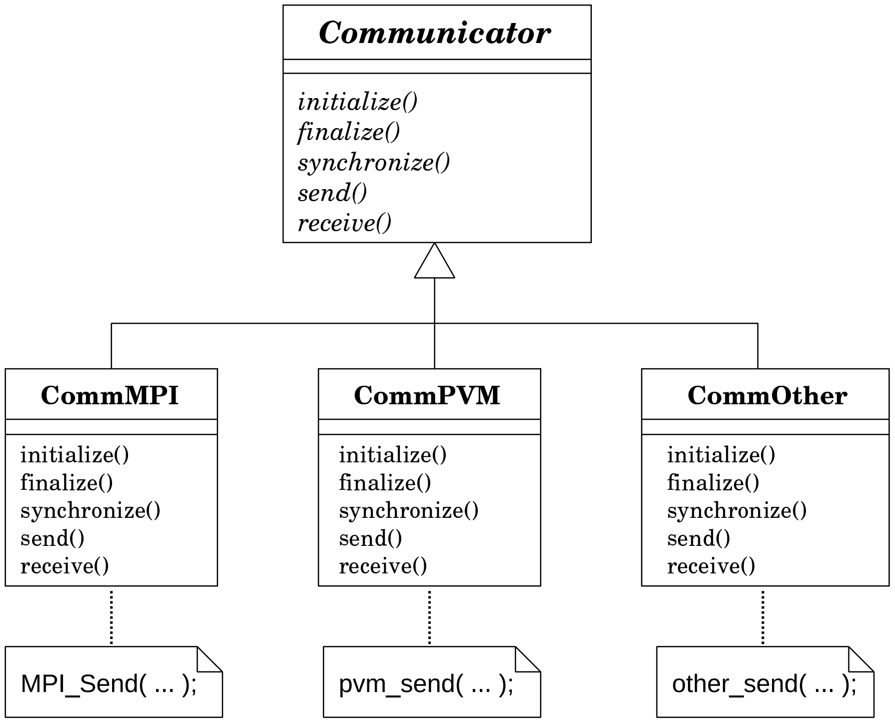


Figure 3.6: Abstract interfaces to parallel environment

is performed under Decomposer class. Decomposer assigns single Field object to Extractor, which is responsible to the actual decomposition task. This Extractor class produces a field object for a decomposed region by extracting a part of the entire region.

After the decomposition work, the rearrangement for the boundaries are carried out. A decomposed region is expected to have boundaries for data transfer, while it also inherits some of the original boundary conditions. In Fig. 3.8, the entire region has four boundary conditions. When this region is decomposed into nine subregions, each of subregion A and B will also have four boundary conditions. In these subregions, the boundary condition for data transfer substitutes some of original conditions. Thus, a field object that is identical in a structural view but is different in contents from the one for the entire region is produced.

Data transfer is performed by the class that is responsible to that procedure. The class is implemented by inheriting the BoundaryProcess class. This enables handling the procedure for data transfer together with the other conditions, without knowing what is exactly performed in each procedure. The data transfer procedure is implemented by using the functions of a message passing library. However, the implementation is done by using the abstract interfaces prepared by the framework, but not by calling the library's routines directly.

These are the classes that are incorporated into the core architecture to realize parallel computation of a particular code. The existence of these classes do not affect the use and the implementation of the other classes for the core architecture.

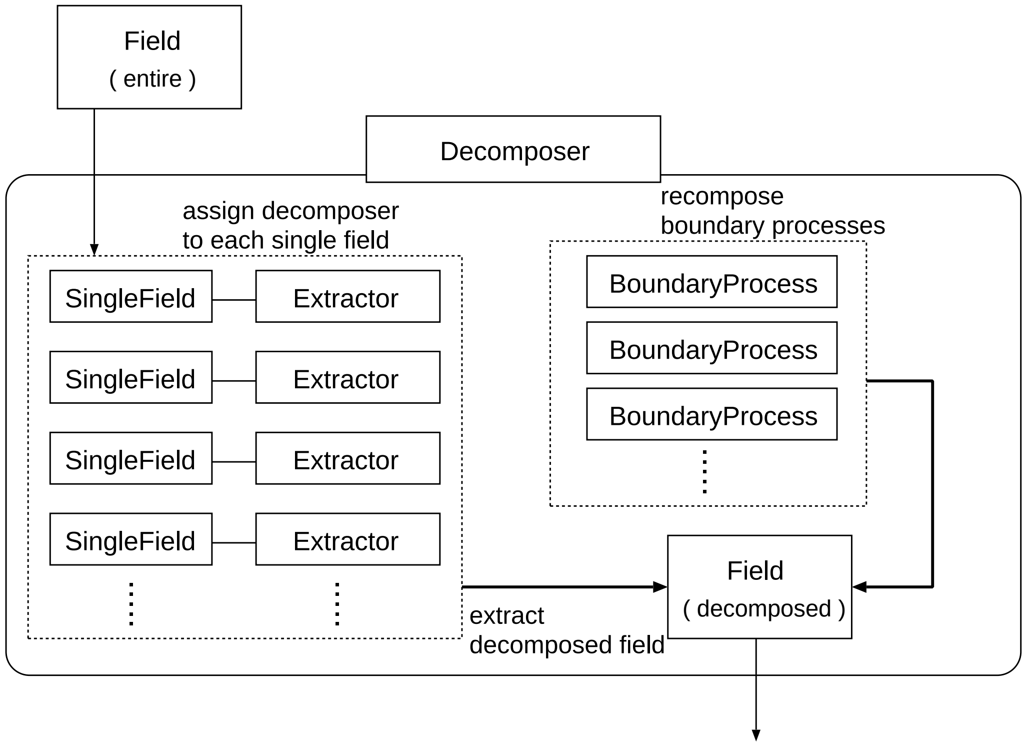


Figure 3.7: Arrangement for domain decomposition

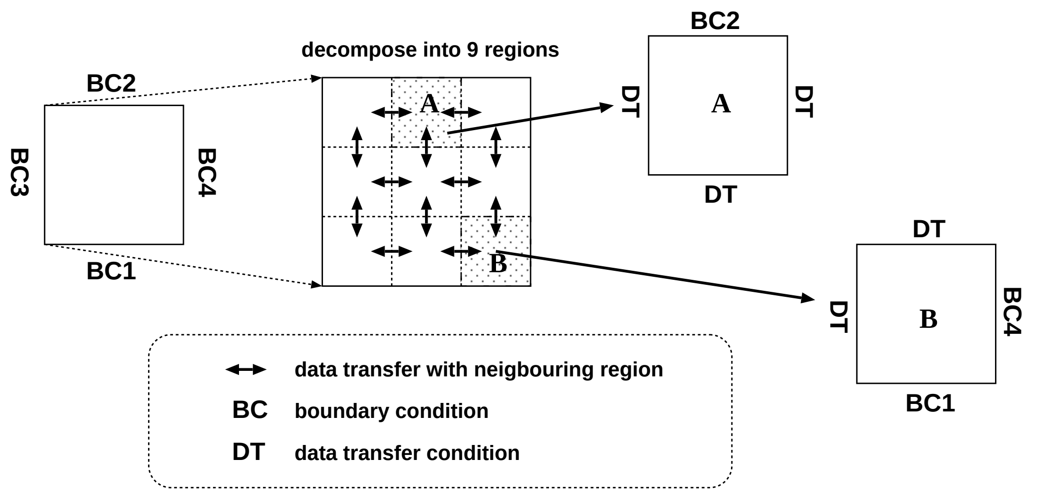


Figure 3.8: Rearrangement of boundary conditions

3.3.6 Classes for Pre-Process

In this framework, pre-process is not confined to a grid generation task only. It is the process for generating a proper field object for a particular calculation. Generation of a field object consists of following procedures.

- definition of field's geometry
- allocation of variables
- definition of boundaries
- assignment of boundary conditions

FieldGenerator and FieldProducer are the classes offering the template to implement a pre-process for particular geometries and conditions. FieldGenerator is the class that defines and generates a grid by itself and FieldProducer imports a grid data to compose a field object. Field-Generator has the definition of a grid's geometry, therefore, grids that are different on parameters can be produced by one same class. This arrangement is superior to ordinary grid generation approaches in regard of reusability. Being different from delivering a raw grid data, a generator class can produce a variety of grids that are different in parameters and sizes. FieldProducer is prepared for the migration of existing grid files into the framework.

A Generator for a particular geometry and particular conditions is implemented as a class derived from FieldGenerator, and similarly a Producer class for a particular grid file is derived from FieldProducer. In Fig. 3.9, two generator classes are derived for two different geometries. One class is for a region around a circular cylinder, and the other is for a rectangular region. The grid generation method and the definition of boundaries particular to the geometry are implemented in each class.

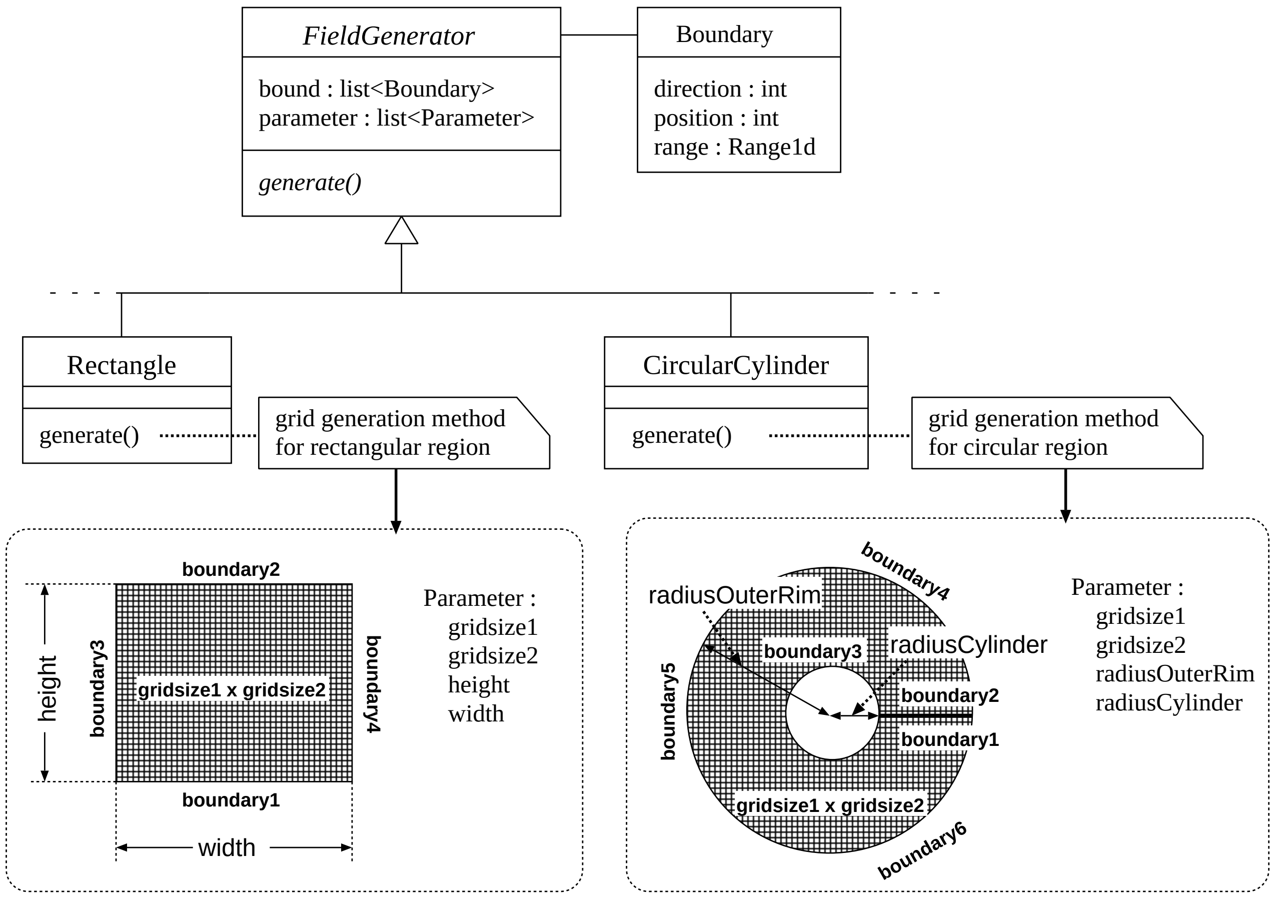


Figure 3.9: Generator classes for particular geometries

FieldGenerator provides the common interfaces to various different generator classes, for dealing with geometric parameters and grid generation methods. An example of the interfaces of FieldGenerator class is shown below.

```
class FieldGenerator {
 public:
  typedef Parameter<int>      IParameter;
  typedef Parameter<float>    FParameter;
  typedef map<int,IParameter*> IParams;
  typedef map<int,FParameter*> FParams;

  void initialize();
  void generate();
  void finalize();
  int  numberOfFloatParameter();
  int  numberOfIntParameter();
  void generateIntParameter();
  void generateFloatParameter();

  Field*     getField();
  IParameter* getIntParameter(const int& key);
  FParameter* getFloatParameter(const int& key);

 protected:
  virtual void defineGeometry() = 0;
  virtual void prepareParameter() = 0;
  virtual void prepareField() = 0;
  virtual void setBounds() = 0;
```

```
    virtual void setConditions() = 0;
    virtual void setInitialCondition() = 0;

    Field     *field;
    IParams   *iparam;
    FParams   *fparam;
  };
```

The parameters are stored as a list of Parameter objects, which are devised for providing a common way in accessing to the various type of parameters. This arrangement enables preparing a single GUI (Graphical User Interface) environment that can serve to various generator classes. An example of such GUI system is shown in Fig. 3.10.

Here, the system that offers a GUI for the grid generation procedure can receive various kind of generator classes, and offers a specific GUI environment for each of them. As the figure shows, when the system receives the generator for airfoil, it prepares a particular environment for manipulating the procedure for that configuration. When a generator for a grid around a three-dimensional circular cylinder is imported, another set of interface is generated according to the parameters prepared in the class. This is achieved by preparing parameter objects that store the information on the interface type of each parameter. In each generator class, parameter objects are prepared according to the geometry it will generate. The GUI system parses the parameter's list, and generates the proper interfaces (Fig. 3.11).

There is an extra support for domain decomposition especially for ones performed on a distributed memory environment. Instead of the general approach that generates the whole data at the beginning and delivers a decomposed part to each parallel-running process afterward, it is the approach that generates local decomposed data in each process. In order to realize the above approach, the domain decomposition procedure is not applied to a concrete grid data directly. We extract the necessary attributes to generate a grid data, and the decomposition procedure is applied to these, instead of the data itself. For a structured grid, the grid range is sufficient for such attribute. For other data structures, appropriate attributes should be found for employing the same arrangement. When a decomposition is applied to a global range, a range for a local region is obtained. Grid data is generated according to the local attributes. Thus, the grid data for the entire region is not generated on one local process, where only a part of entire memory is available. Now the grid generation process is performed in parallel. This prevents the calculation from being limited by the amount of a local memory size.

The approach explained in this section is not for devising a generic method that covers grids of all kind of geometries. Rather, our approach is in the opposite concept. We know by our experience that it is difficult to design such a good generic method. Even if such an ideal generic method were devised, it still would be difficult to generate a grid of particular topology. Such a method tends to be a higher abstracted system and requires several steps to connect the method with a particular problem. Instead, it is easier to find an appropriate and rather straightforward approach for each special problem, and it is convenient to use such a method than a generic one in many cases. Therefore, we believe it is better to offer a common template for implementing each special grid generation method, than to build a grid generator that covers all. Though the example shows only the classes for specific geometries, this does not mean that every class for the pre-process should be so. It could be a genric method that uses a parabolic equation or imports a CAD data for grid generation. The objective of this framework is to define the common interfaces and provide the base by which various methods can be used in a same manner, since it is difficult to cover everything by one method only.

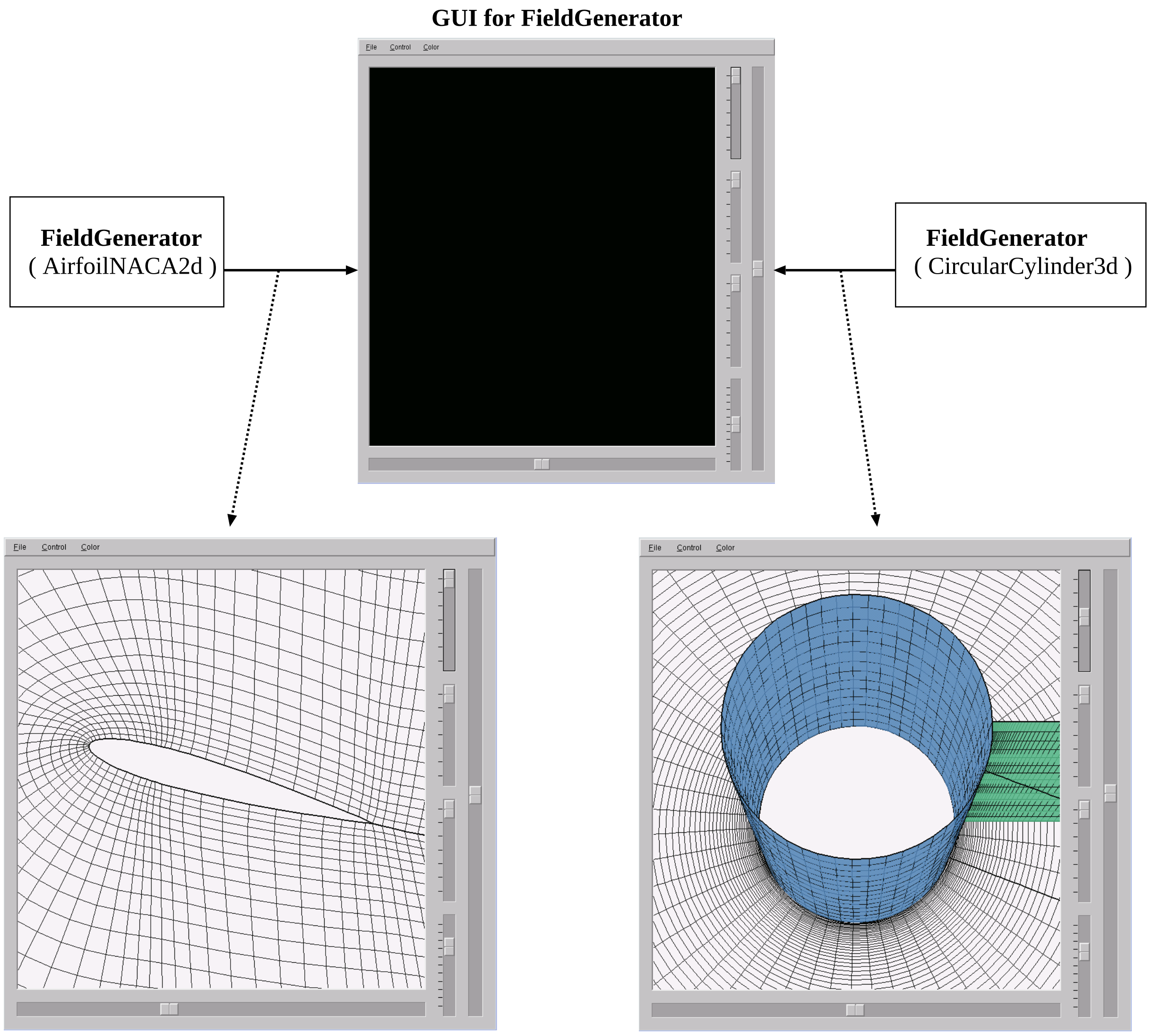
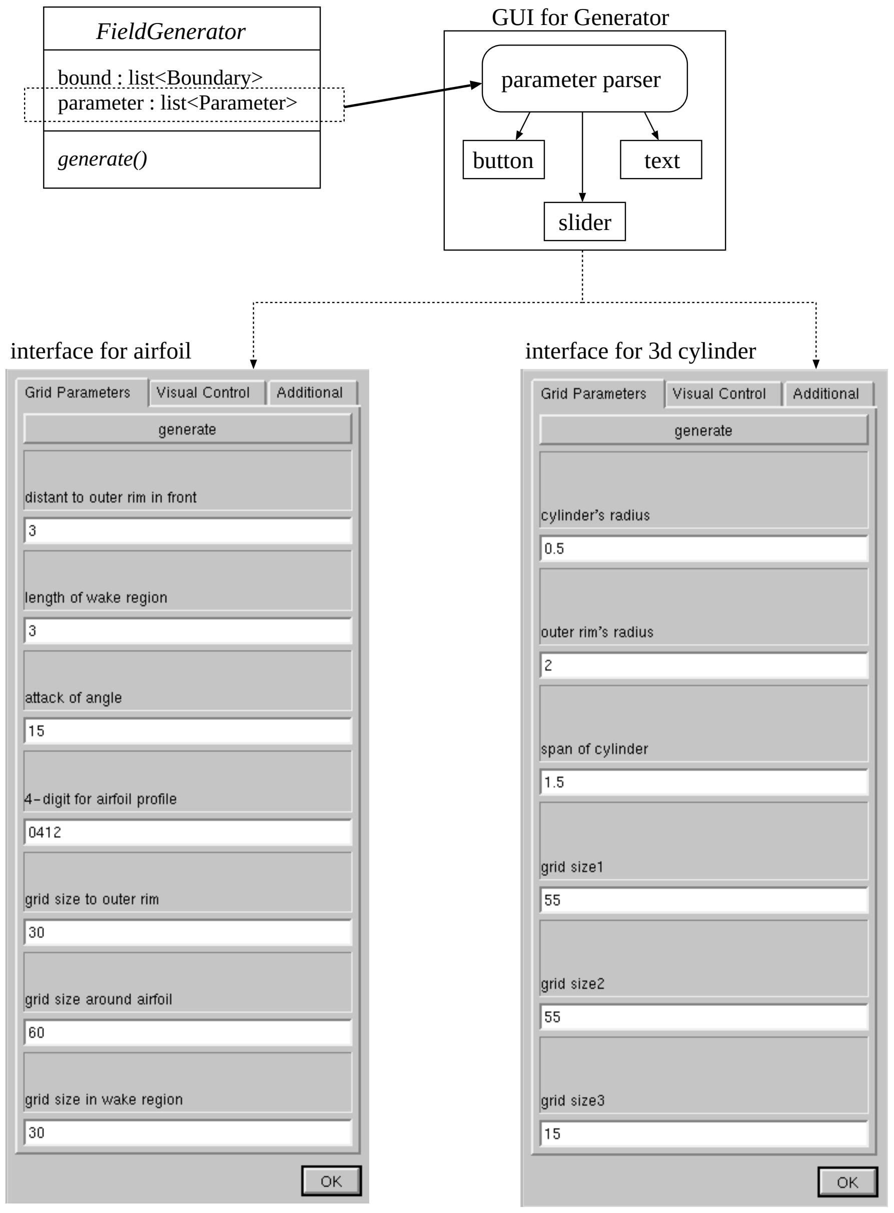


Figure 3.10: GUI system for generator



interface for airfoil

distant to outer rim in front: 3
length of wake region: 3
attack of angle: 15
4-digit for airfoil profile: 0412
grid size to outer rim: 30
grid size around airfoil: 60
grid size in wake region: 30

interface for 3d cylinder

cylinder's radius: 0.5
outer rim's radius: 2
span of cylinder: 1.5
grid size1: 55
grid size2: 55
grid size3: 15

Figure 3.11: Dynamic generation of interface

3.3.7 Classes for Visualization

Though there is no particular arrangement prepared for visualization process, the supporting framework can also be built on the same architecture with the calculation and the pre-process. Each of the procedures for the visualization is implemented by inheriting VisualizeScheme class, which itself inherits FieldUser class. Displaying of velocity vectors and generation of contour lines are the examples of such procedures. The framework is depicted as Fig. 3.12.

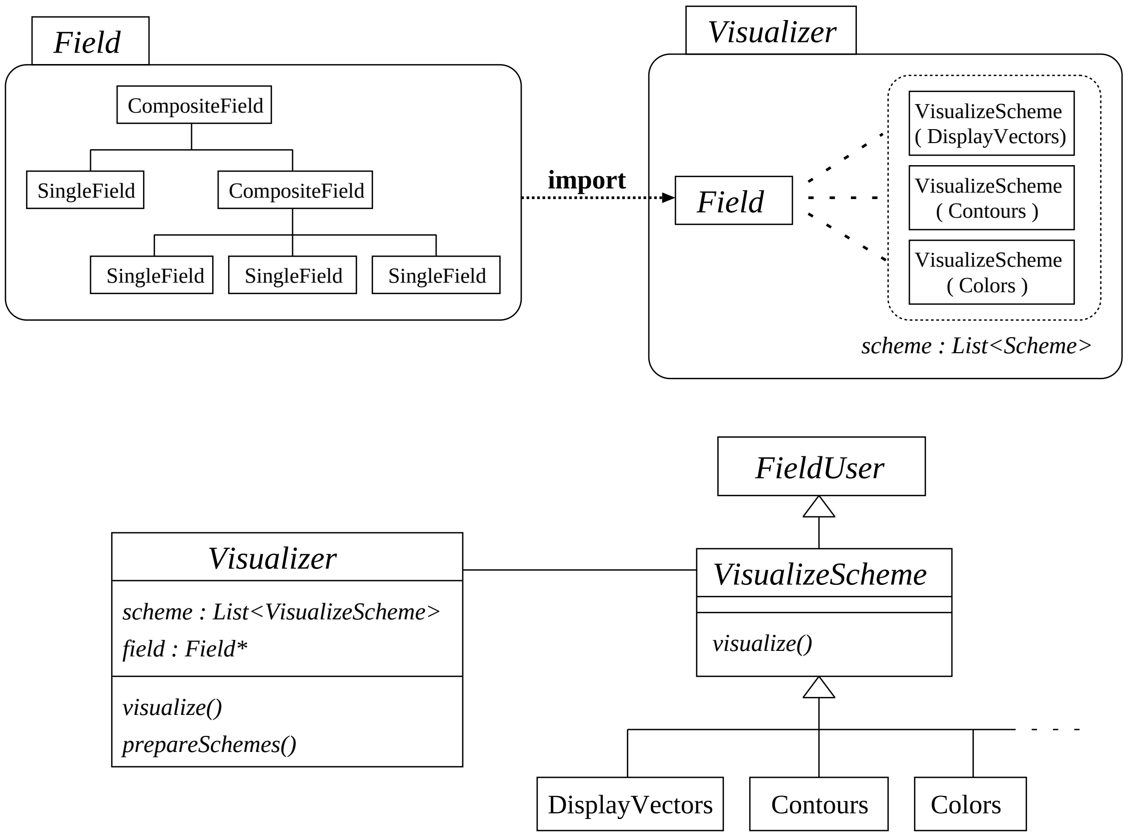


Figure 3.12: Framework for visualization

Visualizer is the class that manages the visualization process. Visualizer prepares necessary VisualizeScheme objects. For visualization, Visualizer imports the field objects that are going to be visualized, and assigns them to each VisualizeScheme object. Visualizer initializes a low-level graphic environment; OpenGL [29] is currently used. Each VisualizeScheme processes its own visualization procedures and draws its result by using the functions provided by the prepared graphic library.

Taking this simple relation as the core architecture, the visualization system can be composed flexibly. Two design examples of such system are shown here. One system is a common visualization system that imports a calculation result and visualizes it. An example of such system is shown in Fig. 3.13. The system generates necessary visualization schemes dynami-

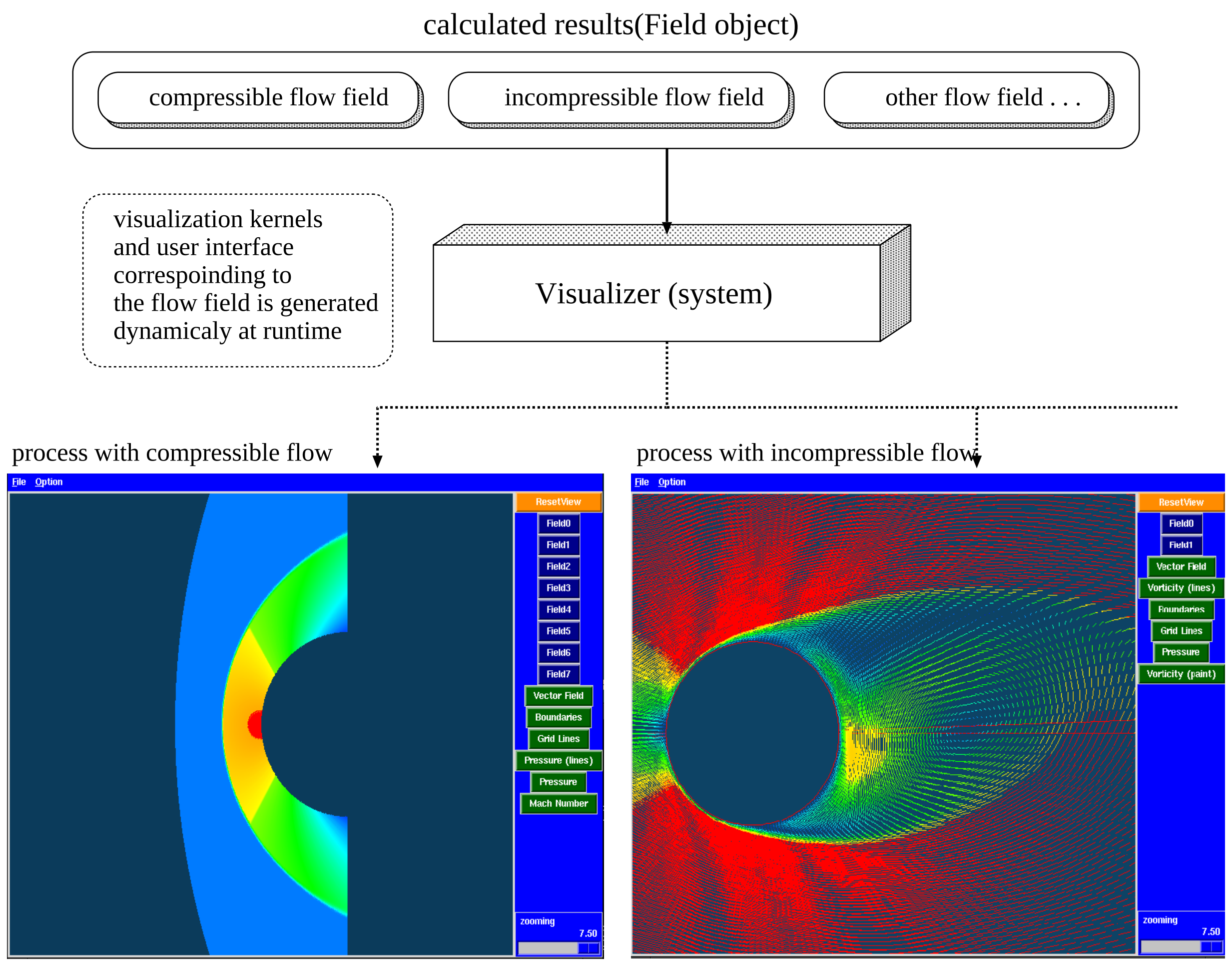


Figure 3.13: Visualization System with GUI

cally, according to the type of imported field object. There are Field objects that are different at the data structure or the variables. For example, there are different classes of the field objects for incompressible flow and compressible flow simulations. The GUI is also generated dynamically so that it matches the functions that are prepared for the type of the imported field object. In the figure, two different images are shown. The left one is a visualized image of a compressible flow field, and the right one is an image of an incompressible flow field. For a compressible flow field, the visualization schemes are prepared for visualizing pressure, Mach number, and density field. The schemes for pressure and vorticity are prepared for a visualization of an incompressible flow field. The scheme for displaying velocity vectors is employed by both cases. The user interface is generated so that it matches the prepared functions. That means the necessary buttons only are displayed in the control panel.

Another possible architecture is a real-time visualization system. All the calculation processes handle an object of Field class for processing their jobs. Therefore, if it is possible to prepare a field object that can be used simultaneously by both calculation process and visualization process, a real-time visualization system can be realized. It is most easily achieved by packing solver and visualization procedures into one solid system. However, such system would be insufficient by the following reasons.

- every solver that wants the facility must implement the visualization routines
- the system will lack portability
- the arrangement harms a flexible design of the systems
- calculation process is made to wait during the visualization

It is possible to design the calculation and visualization as independent processes. These processes are made to cooperate through the file written out to disk storage. Such a system lacks the characteristic of the real-time visualization, since writing and reading a file take a much longer time than using the data on memory An ideal real-time visualization is expected to have the following features.

- independent calculation and visualization processes

- no interference to calculation process
- no file I/O
- no drastic modification required for calculation program

In order to meet the above requirements, use of a shared memory can be introduced. The architecture of the system is depicted as Fig. 3.14. Here, a field object is allocated on a shared memory region. Both the solver and the visualization processes can access the field object as if it is allocated solely for their own usage. The visualization process can attach and detach the

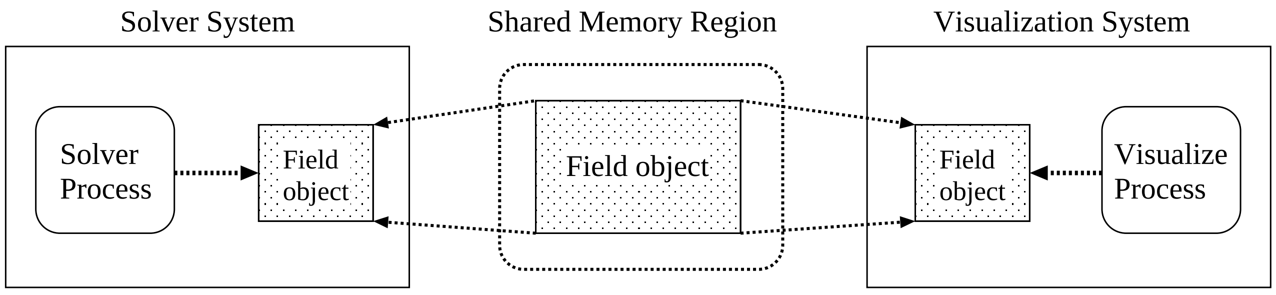
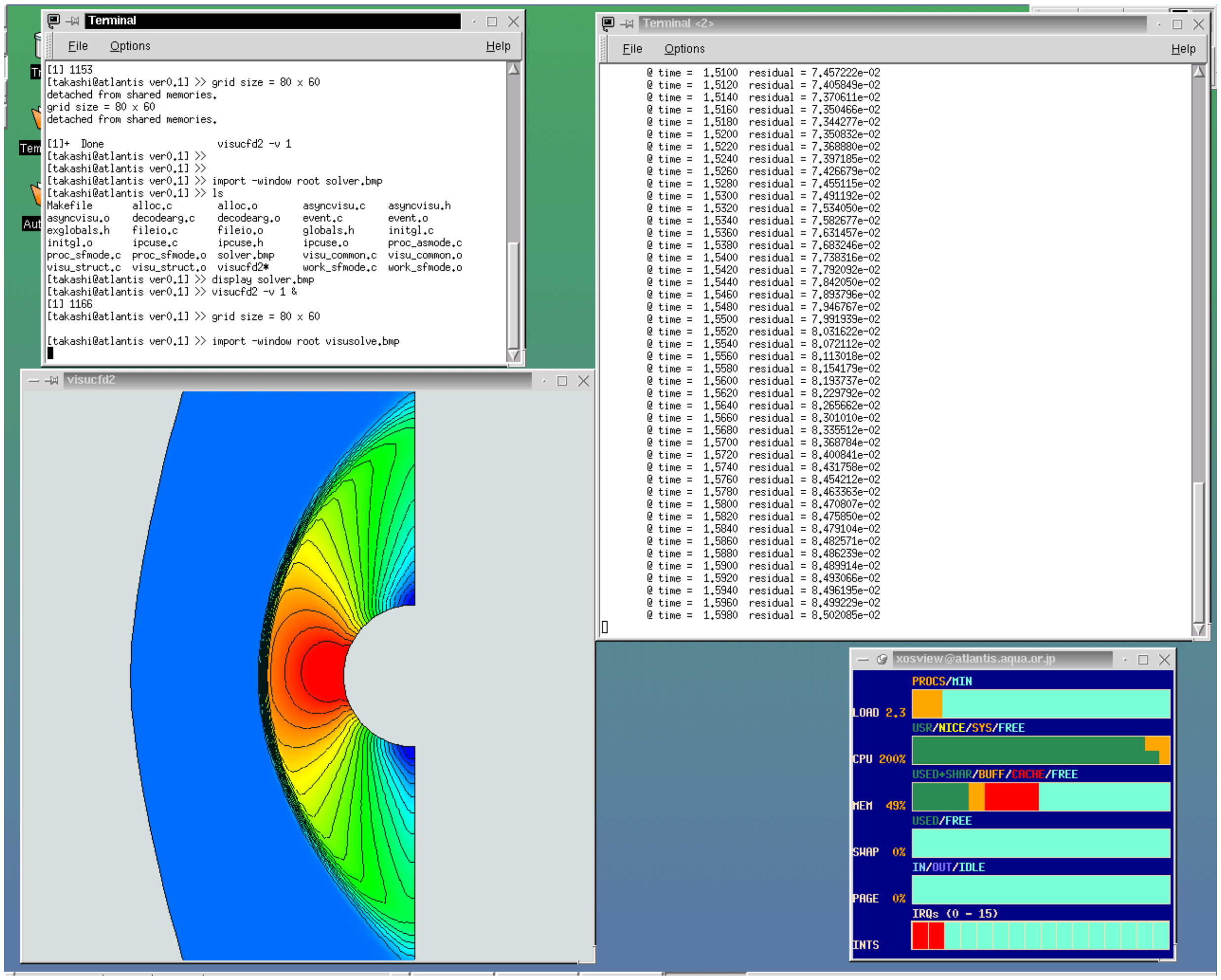


Figure 3.14: Visualization System with Shared Memory

shared memory in any time. This approach of using a shared memory enables a visualization system to display the results at anytime during calculation. The visualization system can freely peep into a calculation process, leave from it, and return again to it, but does not interrupt the calculation by doing so. However, aside from such benefits, it requires an additional programming in both of calculation and visualization systems in order to use a shared memory function. Furthermore, the solver and visualization processes must prepare the same data structure. This harms the condition that requires both processes are independent, in a programming point of view. This is resolved by introducing the framework. Since the framework defines the field object to be used by all of the processes in CFD, if the object is made to be realized on a shared memory, the data structures in both processes become naturally identical. Moreover, the detail programming to use a shared memory can be hidden in the Field class. Therefore, the usage of the field object can be the same with the field object of standard memory allocation. We prepare a class for using shared memory. This design avoids an additional programming from the implementation of the calculation process. Figure 3.15 displays the example of using such system. The window at upper-right place on the computer-screen shows that a calculation process



```
[1] 1153
[takashi@atlantis ver0.1] >> grid size = 80 x 60
detached from shared memories.
grid size = 80 x 60
detached from shared memories.

[1]+  Done                    visucfd2 -v 1
[takashi@atlantis ver0.1] >>
[takashi@atlantis ver0.1] >>
[takashi@atlantis ver0.1] >> import -window root solver.bmp
[takashi@atlantis ver0.1] >> ls
Makefile       alloc.c        alloc.o        asyncvisu.c    asyncvisu.h
asyncvisu.o    decodearg.c    decodearg.o    event.c        event.o
exglobals.h    fileio.c       fileio.o       globals.h      initgl.c
initgl.o       ipcuse.c       ipcuse.h       ipcuse.o       proc_asmode.c
proc_sfmode.c  proc_sfmode.o  solver.bmp     visu_common.c  visu_common.o
visu_struct.c  visu_struct.o  visucfd2*      work_sfmode.c  work_sfmode.o
[takashi@atlantis ver0.1] >> display solver.bmp
[takashi@atlantis ver0.1] >> visucfd2 -v 1 &
[1] 1166
[takashi@atlantis ver0.1] >> grid size = 80 x 60

[takashi@atlantis ver0.1] >> import -window root visusolve.bmp
```

```
   @ time =  1.5100  residual = 7.457222e-02
   @ time =  1.5120  residual = 7.405849e-02
   @ time =  1.5140  residual = 7.370611e-02
   @ time =  1.5160  residual = 7.350466e-02
   @ time =  1.5180  residual = 7.344277e-02
   @ time =  1.5200  residual = 7.350832e-02
   @ time =  1.5220  residual = 7.368880e-02
   @ time =  1.5240  residual = 7.397185e-02
   @ time =  1.5260  residual = 7.426679e-02
   @ time =  1.5280  residual = 7.455115e-02
   @ time =  1.5300  residual = 7.491192e-02
   @ time =  1.5320  residual = 7.534050e-02
   @ time =  1.5340  residual = 7.582677e-02
   @ time =  1.5360  residual = 7.631457e-02
   @ time =  1.5380  residual = 7.683246e-02
   @ time =  1.5400  residual = 7.738316e-02
   @ time =  1.5420  residual = 7.792092e-02
   @ time =  1.5440  residual = 7.842050e-02
   @ time =  1.5460  residual = 7.893796e-02
   @ time =  1.5480  residual = 7.946767e-02
   @ time =  1.5500  residual = 7.991939e-02
   @ time =  1.5520  residual = 8.031622e-02
   @ time =  1.5540  residual = 8.072112e-02
   @ time =  1.5560  residual = 8.113018e-02
   @ time =  1.5580  residual = 8.154179e-02
   @ time =  1.5600  residual = 8.193737e-02
   @ time =  1.5620  residual = 8.229792e-02
   @ time =  1.5640  residual = 8.265662e-02
   @ time =  1.5660  residual = 8.301010e-02
   @ time =  1.5680  residual = 8.335512e-02
   @ time =  1.5700  residual = 8.368784e-02
   @ time =  1.5720  residual = 8.400841e-02
   @ time =  1.5740  residual = 8.431758e-02
   @ time =  1.5760  residual = 8.454212e-02
   @ time =  1.5780  residual = 8.463363e-02
   @ time =  1.5800  residual = 8.470807e-02
   @ time =  1.5820  residual = 8.475850e-02
   @ time =  1.5840  residual = 8.479104e-02
   @ time =  1.5860  residual = 8.482571e-02
   @ time =  1.5880  residual = 8.486239e-02
   @ time =  1.5900  residual = 8.489914e-02
   @ time =  1.5920  residual = 8.493066e-02
   @ time =  1.5940  residual = 8.496195e-02
   @ time =  1.5960  residual = 8.499229e-02
   @ time =  1.5980  residual = 8.502085e-02
```

Figure 3.15: Real-time visualization : solver and visualization process

is performed at that moment. The window at the left-lower place shows that another process for the visualization is proceeding at the same moment. As it monitors the change of the data in the field object, the visualized image is renewed appropriately. These processes are independent, and the visualization system can run and exit without requiring any particular treatments at the calculation process. These processes in this example are running on an SMP (symmetric multi processing) system, which has two processors. Therefore, the visualization system can run on one CPU when it is evoked, while the calculation process is running on the other CPU. With such an environment and the system, the visualization process does not harm the calculation performance.

A visualization system for a parallel computing can be designed similarly. Figure 3.16 shows one of the possible designs for real-time visualization system with the parallel computation. It

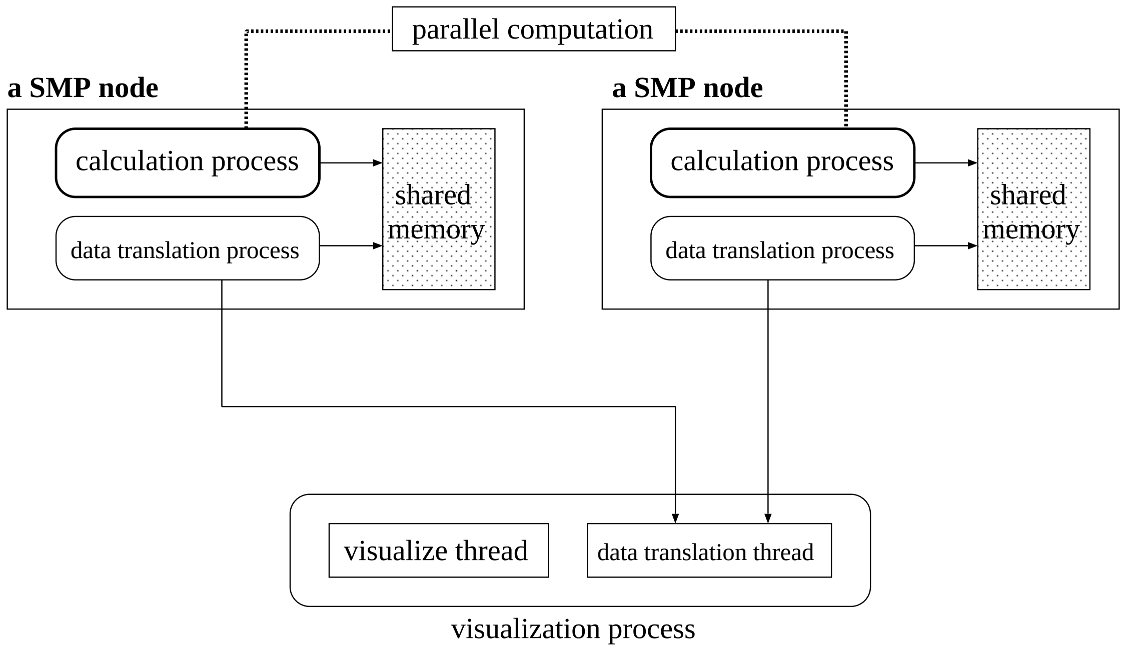


Figure 3.16: Real-time visualization for parallel computing

assumes that the calculation processes are running in parallel on an SMP-cluster system. On each SMP node, a calculation process is running only on one of two CPUs. When the visualization system is evoked to see the progress of the calculation, the processes responsible in transferring the data are generated corresponding to the calculation processes on every node;

each of these runs on the other CPU of each SMP node. A visualization system that is responsible to display the results resides on a remote machine other than the ones used in the parallel calculation. The data translation and the calculation processes share the data by using the shared memory function.

As these examples are indicating, various systems can be realized by combining the modules flexibly with an external component like the shared memory function, while the core architecture remains the same in any designs. The philosophy taken here is that there may not be one ideal system that suffices all kind of usage. Therefore, the framework is designed to offer a different system to a different requirement, by combining the modules flexibly. Thus, the role of the framework here is to offer such modules and to assure their flexible composition.

3.3.8 Persistency

In this framework, grid data and calculation results exist as the objects of Field class. With such a design, it is convenient if the object can be stored and restored directly as an object, instead of the traditional manner to do these actions by interpreting each data fragment in a certain format.

In order to give such facility to the field object, an abstract class, Serializable, is introduced (Fig. 3.17). The class defines the interfaces for writing and reading an object's contents. Any class that needs the function can be built by inheriting this abstract class. Each derived class should implement concrete processes for the defined abstract interfaces. This means each of these classes should define a method for storing itself as a file, and a method for restoring itself by reading data from the file. The stored information should be enough to rebuild the object. However, what and how the data is stored can be chosen freely if the above condition is satisfied. Therefore, either of ASCII or binary sequential data can be used for the purpose. Even HTML (Hyper Text Markup Language), or XML (eXtensible Markup Language) can be used. The difference of the format is a matter of the implementation that is hidden under the interface. Here, another classes, ObjectReader and ObjectWriter, are introduced to support the storing and the restoring of objects. The reason for installing these additional classes is to separate a

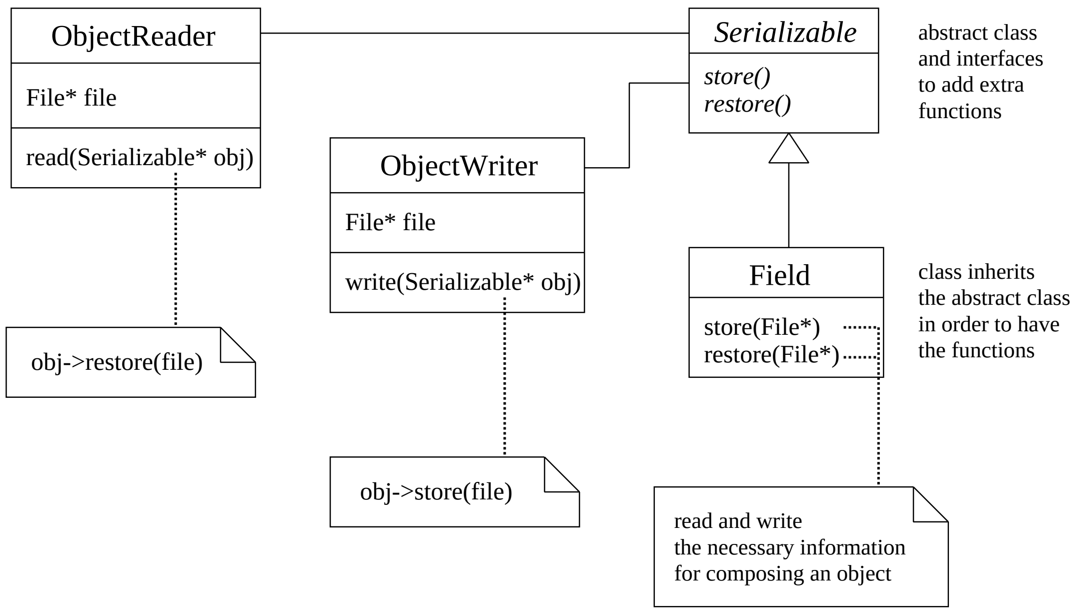


Figure 3.17: A Framework for Store/Restore Objects

file system from the field object, since these are conceptually different.

3.3.9 Extensions

In the extension of the framework, it is important to assure that the addition of new function will not cause an overall reconstruction of the original set. The means to adding newer function to the core architecture of the framework is to extract the abstraction of the common functionality. With the object-oriented approach, the above method is realized by setting an abstract class. The classes derived from the same abstract class become compatible. Therefore, replacing these classes does not affect the program where that functionality is used.

In this framework, the mechanism for using shared memory is realized in this approach. The common functionality concerning to the allocation of data is abstracted and prepared as allocator class. Various ways of allocating memory are realized as classes by inheriting that abstract allocator class. The data container classes like Array employ std_allocator by default, which offers a standard way of memory allocation. Using the shared memory is achieved by assigning shar_allocator as the allocator class. Here, shar_allocator is the class that is responsible to the use of shared memory. Since these allocator classes have the identical interface, the replacing them is done by doing so literally. With C++, this replacement is done smartly by using the template function. The same approach is taken to introduce a new set of flow variables.

The other way to add a new function other than replacing an existing module is to prepare a class that offers a new function. The other classes that want the function can acquire it by inheriting that class. In the framework, a thread-process and serialization of the classes are realized by this approach. For example, a calculation class can be performed as a thread process by making it as a derived class of Thread class, which provides the function.

These means for the extension are common aspect in the object-oriented approach. The important point here is to give a right design at the beginning, to allow such extension flexibly afterwards.

Chapter 4

CALCULATION WITH THE FRAMEWORK

4.1 Programming

4.1.1 Grid Generation

The framework offers FieldGenerator class for implementing a concrete procedures of the pre-process. A grid generation algorithm for a particular geometry is implemented in a method of FieldGenerator's derived class. The following is the sample generator class that is for the calculation of two-dimensional cavity flow.

```
class Cavity2d : public FieldGenerator {
  Field* getField();

  void defineGeometry();
  void prepareParameter();
  void prepareField();
  void setBounds();
  void setConditions();
  void setInitialCondition();

 private:
  Field  *field;
  Grid2d *grid;
  Vars2d *vars;
};
```

The methods seen in the class are defined in FieldGenerator class as the methods to be implemented in every derived class. A grid generation program is written in ”defineGeometry” method of this class. The method is written as follows.

```
void Cavity2d::defineGeometry() {
  float width  = this->getFloatParameter(0)->value();
  float height = this->getFloatParameter(1)->value();

   for (int i=grid->begin1(); i<=grid->end1(); i++)
    for (int j=grid->begin2(); j<=grid->end2(); j++) {
      grid->x(i,j) = width/(grid->size1()-1)*(i-grid->begin1());
      grid->y(i,j) = height/(grid->size2()-1)*(j-grid->begin2());
    }
}
```

For the completion of the pre-process, definition of boundaries and assignment of boundary conditions must be finished. In addition, the initial state of the variables should be set. These procedures are implemented in ”setBounds”, ”setConditions”, and ”setInitialCondition” respectively. These methods are written as followings.

```
void Cavity2d::setBounds() {
  grid->generateBoundary(4);

  Bound *bnd = grid->getBoundary(0); // bottom edge
  bnd->id() = 0;
  bnd->direction() = Bound::DIM1;
  bnd->position()  = grid->begin2();
  bnd->range()->begin() = grid->begin1();
  bnd->range()->end()   = grid->end1();

  bnd = grid->getBoundary(1); // top edge
  bnd->id() = 1;
  bnd->direction() = Bound::DIM1;
  bnd->position()  = grid->end2();
  bnd->range()->begin() = grid->begin1();
  bnd->range()->end()   = grid->end1();

  bnd = grid->getBoundary(2); // left edge
  bnd->id() = 2;
  bnd->direction() = Bound::DIM2;
  bnd->position()  = grid->begin1();
  bnd->range()->begin() = grid->begin2();
  bnd->range()->end()   = grid->end2();

  bnd = grid->getBoundary(3); // right edge
  bnd->id() = 3;
  bnd->direction() = Bound::DIM2;
  bnd->position()  = grid->end1();
  bnd->range()->begin() = grid->begin2();
  bnd->range()->end()   = grid->end2();
}

void Cavity2d::setConditions() {
  FieldType* fld = static_cast<FieldType*>(field);

  IncNonslip2d<VarType> *bc1 = new IncNonslip2d<VarType>;
  bc1->getRegion()->field = fld;
  bc1->getRegion()->bound = grid->getBoundary(0);
  bc1->getRegion()->place = BoundRegion2d::BOTTOM;
  bc1->initialize();

  UniformFlow2d<VarsType> *bc2 = new UniformFlow2d<VarsType>;
  VarsType uniform;
  uniform.u() = 1.0;
  uniform.v() = 0.0;
  uniform.p() = 0.0;
  bc2->setFlow(uniform);
  bc2->getRegion()->field = fld;
  bc2->getRegion()->bound = grid->getBoundary(1);
  bc2->getRegion()->place = BoundRegion2d::TOP;
  bc2->initialize();

  IncNonslip2d<VarType> *bc3 = new IncNonslip2d<VarType>;
  bc3->getRegion()->field = fld;
  bc3->getRegion()->bound = grid->getBoundary(2);
  bc3->getRegion()->place = BoundRegion2d::BOTTOM;
  bc3->initialize();

  IncNonslip2d<VarType> *bc4 = new IncNonslip2d<VarType>;
  bc4->getRegion()->field = fld;
  bc4->getRegion()->bound = grid->getBoundary(3);
  bc4->getRegion()->place = BoundRegion2d::TOP;
  bc4->initialize();

  fld->setCondition(static_cast<BoundaryProcess*>(bc1));
  fld->setCondition(static_cast<BoundaryProcess*>(bc2));
  fld->setCondition(static_cast<BoundaryProcess*>(bc3));
  fld->setCondition(static_cast<BoundaryProcess*>(bc4));
}

void Cavity2d::setInitialCondition() {
  VarsType initflow;
  initflow.u() = 0.0;
  initflow.v() = 0.0;
  initflow.p() = 0.0;

  for (int i=grid->begin1(); i<=grid->end1(); i++)
    for (int j=grid->begin2(); j<=grid->end2(); j++) {
      vars->at(i,j).u() = initflow.u();
      vars->at(i,j).v() = initflow.v();
      vars->at(i,j).p() = initflow.p();
    }
}
```

As it can be seen in these implementations, a boundary and a condition is also realized as objects. They are generated in these methods and appropriate parameters are set to the objects. Initial values are assigned to the object for the flow variables.

These are the way of realizing a pre-process of a particular geometry and conditions. For a different geometry and conditions, another class is prepared, and specific procedures are implemented at the methods of that class. The result of a pre-process class is obtained as an object of Field class.

4.1.2 Boundary Condition

For implementing a procedure to apply a boundary condition, an abstract class, BoundaryProcess, is offerred. A particular procedure is realized as a class that inherits that abstract class. Objects of the boundary condition classes are prepared in a field object, and are called in a solver class. For being fit into that framework, BoundaryProcess offers the interface as followings. Each derived class must have concrete implementations for the interface.

```
class BoundaryProcess {
 public:
  virtual void initialize() = 0;
  virtual void apply() = 0;
  virtual BoundaryProcess* generateObjectCopy() = 0;
};
```

"apply" is the method for applying the particular procedure of each class. Therefore, the particular algorithm of applying a boundary condition is implemented into this method of each class. In "initialize" method, the pointers for accessing the data in a field object are prepared. In addition, the position indices are prepared at this method. The last method, "generateObjectCopy", is used by the field decomposition process. Boundaries that remain after the decomposition give the copy of themselves to assign them to a newer generated field object for a decomposed region. By having this method, Decomposer can build a field object without knowing what objects of boundary processes should be generated.

An example of the concrete class of BoundaryProcess is shown as follows.

```
template <class Variables>
class UniformFlow2d : public BoundaryProcess2d<Variables> {
 public:
  void setFlow(Variables var) {
    uniflow = var;
  }

  Variables* getFlow() {
    return &uniflow;
  }

  void initialize() {
    field = static_cast<Field2d*>(region->field);
    grid  = static_cast<Grid2d*>(field->getGrid());
    vars  = static_cast<Vars2d*>(field->getVarfield());
    bound = static_cast<StrBound2d<double>* >(region->bound);

    appidx = bound->position();
    switch (region->place) {
     case BoundRegion2d::BOTTOM:
       viridx = appidx - 1;
       break;
     case BoundRegion2d::TOP:
       viridx = appidx + 1;
```

```
      break;
    }
  }

  void apply() {
    switch (bound->direction()) {
     case StrBound2d<>::DIM1:
      for (int i=bound->range()->begin(); i<=bound->range()->end(); i++) {
        for (int n=0; n<Variables::size(); n++) {
          vars->at(i,appidx).set(n,uniflow.get(n));
          vars->at(i,viridx).set(n,uniflow.get(n));
        }
      }
      break;
     case StrBound2d<>::DIM2:
      for (int j=bound->range()->begin(); j<=bound->range()->end(); j++) {
        for (int n=0; n<Variables::size(); n++) {
          vars->at(appidx,j).set(n,uniflow.get(n));
          vars->at(viridx,j).set(n,uniflow.get(n));
        }
      }
      break;
    }
  }

  BoundaryProcess* generateObjectCopy() {
    UniformFlow2d<Variables> *ocp = new UniformFlow2d<Variables>;
    ocp->setFlow(uniflow);
    return ocp;
  }

 protected:
  Variables uniflow;
  StrBound2d<double> *bound;
  int appidx,viridx;
};
```

The class is the one for applying uniform flow condition to a boundary. The class has proper methods other than the methods defined in the abstract class, for setting the constant value for the condition.

Other procedures are similarly prepared, by implementing the particular algorithm for applying its condition in ”apply” method.

4.1.3 Calculation Scheme

A solver algorithm is installed into two classes. Two abstract classes, Solver and Scheme, are prepared for that purpose. The advantages to have two classes for the implementation are counted as follows.

- can separate the treatment of calculation regions and numerical algorithm
- can realize a flexible composition in designing a solver
- can achieve a high modularity by exchanging a scheme part

The assignment of each class is explained as follows. Scheme is the class for implementing the very core of the numerical scheme. Solver is realized as a template class that receives a Scheme class as its parameter. Solver binds a field object and a scheme object, and controls the calculation by calling the methods of the scheme. Solver also prepares the necessary parameters for calculation. Solver class is offered as followings.

```
template <class AlgScheme>
class Solver : public FieldUser {
 public:
  virtual void prepareParameter() = 0;
  int numberOfFloatParameter();
  int numberOfIntParameter();

  IParameter* getIntParameter(const int& key);
  FParameter* getFloatParameter(const int& key);
  void generateIntParameter();
  void generateFloatParameter();
  void setCommunicationInterface(Communicator *mpiface);

  void initialize() {
    Fields::iterator fitr;
    for (fitr=fields.begin(); fitr!=fields.end(); fitr++) {
      Field* fld = fitr->second;
      Scheme* scheme = new AlgScheme;
      schemes.insert(Schemes::value_type(fitr->first,scheme));
      fld->accept(scheme);
      scheme->initialize();
    }
  }

 protected:
  Schemes   schemes;
  IParams   *iparam;
  FParams   *fparam;
  Communicator *mp;
};
```

A specific scheme class is set to a solver class to form a particular solver process. The abstract class, Scheme, is offered merely for giving a unified treatment for various different scheme classes. Therefore, no method is defined in that class. A particular scheme is realized as a class that inherits the abstract class. Each scheme class defines the methods for each own's benefit, which are called by the corresponding solver class.

The examples of Solver and Scheme classes for a particular algorithm are presented below.

```
/*--------------------------
    SolverMAC
---------------------------*/
template <class AlgScheme>
class SolverMAC : public Solver<AlgScheme> {
 public:
  void  prepareParameter();
  void  setDT(const float& value);
  void  setReynoldsNumber(const float& value);
  float getDT();
  float getReynoldsNumber();

  void calculation(const int& nloop);
  void calculation() {
    Schemes::iterator sitr;
    float residual = 0.0;
    float threshold = 0.0;
```

```
int counter = 0;
do {
  residual = 0.0;
  for (sitr=schemes.begin(); sitr!=schemes.end(); sitr++) {
    AlgScheme *macscheme
      = static_cast<AlgScheme*>(sitr->second);
    macscheme->solvePoisson();
    ++counter;
    residual += macscheme->getResidualSOR();
    threshold = macscheme->getThresholdSOR();
    this->syncvalue(residual);
  }
  this->applyBoundaryProcess();
} while (residual > threshold && counter < looplimit1);


for (sitr=schemes.begin(); sitr!=schemes.end(); sitr++) {
  AlgScheme *macscheme
    = static_cast<AlgScheme*>(sitr->second);
  macscheme->preprocNS();
}

counter = 0;
do {
  residual = 0.0;
  for (sitr=schemes.begin(); sitr!=schemes.end(); sitr++) {
    AlgScheme *macscheme
      = static_cast<AlgScheme*>(sitr->second);
      macscheme->solveNS();
      ++counter;
      residual += macscheme->getResidualNS();
      threshold = macscheme->getThresholdNS();
      this->syncvalue(residual);
    }
    this->applyBoundaryProcess();
  } while (residual > threshold && counter < looplimit2);
 }

 void applyBoundaryProcess() {
  Conditions::iterator citr;
  for (citr=conditions.begin(); citr!=conditions.end(); citr++) {
    citr->second->apply();
  }
 }
};

/*--------------------------
    IncmpUpwind2d
---------------------------*/
template <class Variables>
class IncmpUpwind2d : public SchemeStr2d<Variables> {
 public:
  typedef IncMetrics2d<Type,Allocator>  Metrics;
  void initialize();
  void setReynoldsNumber(const float& value);
  void setDT(const float& value);
void solvePoisson() {
  residu_s = 0.0;
  int ip,im,jp,jm;
  for (int i=grid->begin1()+1; i<=grid->end1()-1; i++)
    for (int j=grid->begin2()+1; j<=grid->end2()-1; j++) {
      ip = i + 1, im = i - 1;
      jp = j + 1, jm = j - 1;

      x1g11 = grid->x(ip,j) - 2.0*grid->x(i,j) + grid->x(im,j);
      x1g12 = 0.25*(grid->x(ip,jp) - grid->x(ip,jm)
                   - grid->x(im,jp) + grid->x(im,jm));
      x1g22 = grid->x(i,jp) - 2.0*grid->x(i,j) + grid->x(i,jm);

      x2g11 = grid->y(ip,j) - 2.0*grid->y(i,j) + grid->y(im,j);
      x2g12 = 0.25*(grid->y(ip,jp) - grid->y(ip,jm)
                   - grid->y(im,jp) + grid->y(im,jm));
      x2g22 = grid->y(i,jp) - 2.0*grid->y(i,j) + grid->y(i,jm);

      alpha = metric->alpha(i,j);
      beta  = metric->beta(i,j);
      gamma = metric->gamma(i,j);

      term1 = alpha*x1g11 + 2.0*beta*x1g12 + gamma*x1g22;
      term2 = alpha*x2g11 + 2.0*beta*x2g12 + gamma*x2g22;

      term_i1 = metric->m11(i,j)*term1;
      term_i2 = metric->m12(i,j)*term2;
```

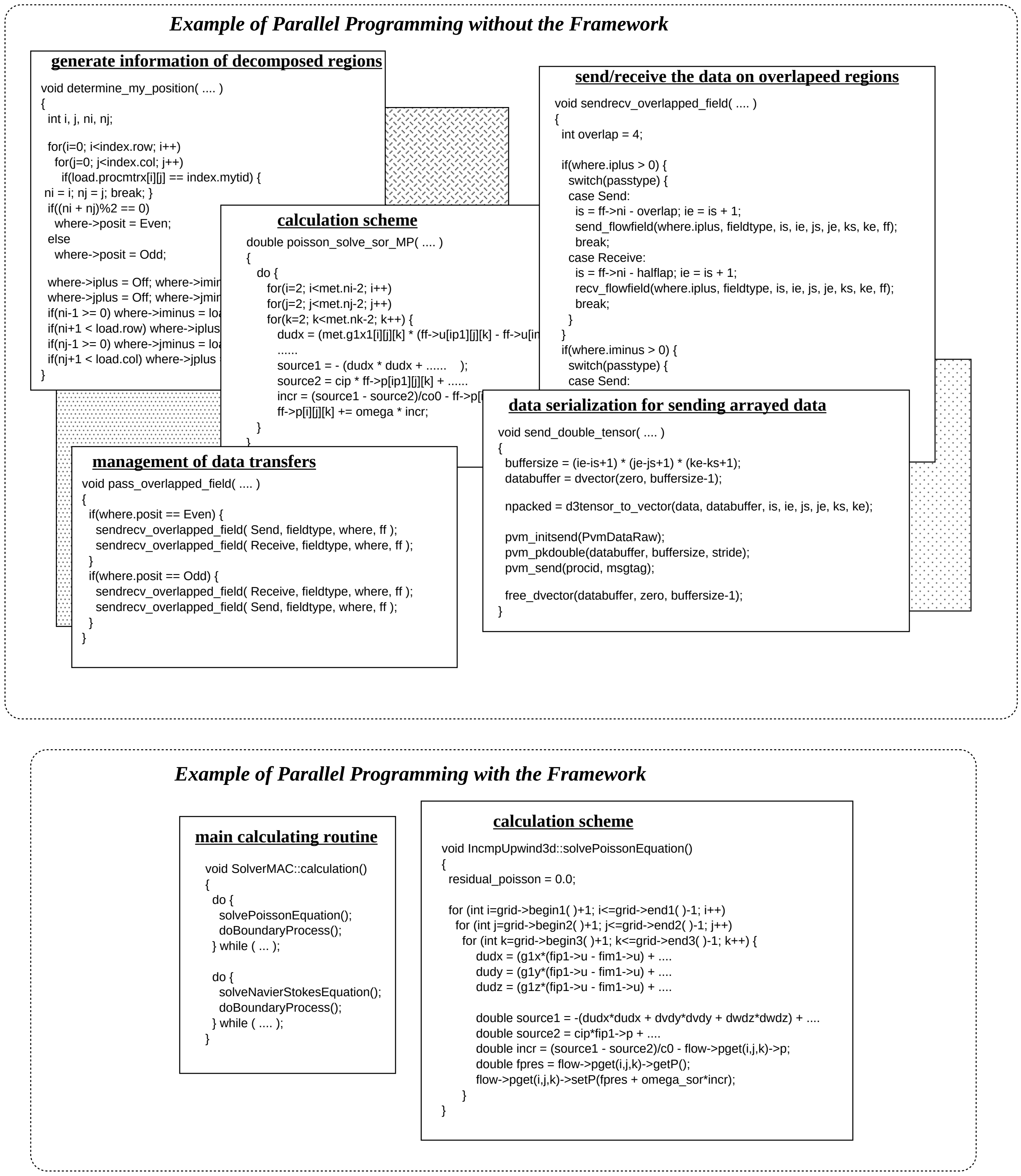
```
term_j1 = metric->m21(i,j)*term1;
term_j2 = metric->m22(i,j)*term2;

c0  = -2.0*(alpha + gamma);
cip = alpha - 0.5*(term_i1 + term_i2);
cim = alpha + 0.5*(term_i1 + term_i2);
cjp = gamma - 0.5*(term_j1 + term_j2);
cjm = gamma + 0.5*(term_j1 + term_j2);

udif1 = vars->at(ip,j).u() - vars->at(im,j).u();
udif2 = vars->at(i,jp).u() - vars->at(i,jm).u();
vdif1 = vars->at(ip,j).v() - vars->at(im,j).v();
vdif2 = vars->at(i,jp).v() - vars->at(i,jm).v();

dudx = 0.5*(metric->m11(i,j)*udif1 + metric->m21(i,j)*udif2);
dudy = 0.5*(metric->m12(i,j)*udif1 + metric->m22(i,j)*udif2);
dvdx = 0.5*(metric->m11(i,j)*vdif1 + metric->m21(i,j)*vdif2);
dvdy = 0.5*(metric->m12(i,j)*vdif1 + metric->m22(i,j)*vdif2);

src1 = -(dudx*dudx + dvdy*dvdy + 2.0*dudy*dvdx)
  + 1.0/dt*(dudx + dvdy);
src2 = cip*vars->at(ip,j).p() + cim*vars->at(im,j).p()
  + cjp*vars->at(i,jp).p() + cjm*vars->at(i,jm).p()
  + 2.0*0.25*beta*(vars->at(ip,jp).p() - vars->at(ip,jm).p()
                 - vars->at(im,jp).p() + vars->at(im,jm).p());

pprev = vars->at(i,j).p();
incr  = (src1 - src2)/c0 - pprev;
```

```
          vars->get(i,j)->p() = pprev + omega_s*incr;
          residu_s += incr*incr;
        }
    }

    void preprocNS();

    void solveNS() {
      ........
    }

    float getResidualSOR();
    float getResidualNS();
    float getThresholdSOR();
    float getThresholdNS();

   private:
    Metrics    *metric;
    float      Re;
    float      dt;
  };
```

These classes are for a solver of incompressible flow problem, by using the formulation of MAC (Marker and Cell) method. The interfaces are presented but the detail of the implementation is being omitted in most of parts. In this case, the scheme class implements two methods for the calculation. One is for solving the poisson equation for the pressure, and the other is for solving the Navier-Stokes equations. At ”calculation” method of the solver class, SolverMAC, these methods are called to complete the actual calculation procedure. The implementation for these classes are not effected when it is used for parallel computation.

For implementing another calculation algorithm, another set of Solver and Scheme classes is prepared. The numerical algorithm is implemented into the Scheme class, being installed into one or several methods. At Solver class, a method calling these methods in Scheme class must be prepared. Since Scheme class inherits FieldUser, the program is written by using the variables and data that are provided at a field object.

As for comparing the coding efforts for developing a parallel program with and without the framework, the amounts of the required programming are illustrated as Fig. 4.1. The upper part of the figure shows the coding for a CFD code using a message-passing library, but without the framework. The chart does not include all of the required coding. However, it is enough to reveal the complicated structure and the amount of the efforts for completing a code. The procedures for the data management of the domain decomposition and for the data transfers are explicitly written, in addition to the main calculation routine. The lower part of the figure shows the amount of the programming effort that is required for completing a parallel code, when the coding is supported by the framework. It can be easily observed that the required effort is significantly reduced. Another important aspect of using the framework other than the amount of the codings is understood here, It is the fact that one can concentrate only to the calculation algorithm even in developing a parallel code. This example shows how the framework helps the programming for parallel computation. The program lists for the both cases that appear in Fig. 4.1 are attached in Appendix A. These are not the complete codes, but are the fragments enough to give the hint on the kind and amount of the coding needed in both cases.



Example of Parallel Programming without the Framework

generate information of decomposed regions

```
void determine_my_position( .... )
{
  int i, j, ni, nj;

  for(i=0; i<index.row; i++)
    for(j=0; j<index.col; j++)
      if(load.procmtrx[i][j] == index.mytid) {
 ni = i; nj = j; break; }
  if((ni + nj)%2 == 0)
    where->posit = Even;
  else
    where->posit = Odd;

  where->iplus = Off; where->imir
  where->jplus = Off; where->jmir
  if(ni-1 >= 0) where->iminus = loa
  if(ni+1 < load.row) where->iplus
  if(nj-1 >= 0) where->jminus = loa
  if(nj+1 < load.col) where->jplus
}
```

calculation scheme

```
double poisson_solve_sor_MP( .... )
{
   do {
     for(i=2; i<met.ni-2; i++)
     for(j=2; j<met.nj-2; j++)
     for(k=2; k<met.nk-2; k++) {
        dudx = (met.g1x1[i][j][k] * (ff->u[ip1][j][k] - ff->u[im
        ......
        source1 = - (dudx * dudx + ......    );
        source2 = cip * ff->p[ip1][j][k] + ......
        incr = (source1 - source2)/co0 - ff->p[
        ff->p[i][j][k] += omega * incr;
     }
  }
```

send/receive the data on overlapeed regions

```
void sendrecv_overlapped_field( .... )
{
  int overlap = 4;

  if(where.iplus > 0) {
    switch(passtype) {
    case Send:
      is = ff->ni - overlap; ie = is + 1;
      send_flowfield(where.iplus, fieldtype, is, ie, js, je, ks, ke, ff);
      break;
    case Receive:
      is = ff->ni - halflap; ie = is + 1;
      recv_flowfield(where.iplus, fieldtype, is, ie, js, je, ks, ke, ff);
      break;
    }
  }
  if(where.iminus > 0) {
    switch(passtype) {
    case Send:
```

management of data transfers

```
void pass_overlapped_field( .... )
{
  if(where.posit == Even) {
    sendrecv_overlapped_field( Send, fieldtype, where, ff );
    sendrecv_overlapped_field( Receive, fieldtype, where, ff );
  }
  if(where.posit == Odd) {
    sendrecv_overlapped_field( Receive, fieldtype, where, ff );
    sendrecv_overlapped_field( Send, fieldtype, where, ff );
  }
}
```

data serialization for sending arrayed data

```
void send_double_tensor( .... )
{
  buffersize = (ie-is+1) * (je-js+1) * (ke-ks+1);
  databuffer = dvector(zero, buffersize-1);

  npacked = d3tensor_to_vector(data, databuffer, is, ie, js, je, ks, ke);

  pvm_initsend(PvmDataRaw);
  pvm_pkdouble(databuffer, buffersize, stride);
  pvm_send(procid, msgtag);

  free_dvector(databuffer, zero, buffersize-1);
}
```

Example of Parallel Programming with the Framework

main calculating routine

```
void SolverMAC::calculation()
{
  do {
    solvePoissonEquation();
    doBoundaryProcess();
  } while ( ... );

  do {
    solveNavierStokesEquation();
    doBoundaryProcess();
  } while ( .... );
}
```

calculation scheme

```
void IncmpUpwind3d::solvePoissonEquation()
{
  residual_poisson = 0.0;

  for (int i=grid->begin1( )+1; i<=grid->end1( )-1; i++)
    for (int j=grid->begin2( )+1; j<=grid->end2( )-1; j++)
      for (int k=grid->begin3( )+1; k<=grid->end3( )-1; k++) {
        dudx = (g1x*(fip1->u - fim1->u) + ....
        dudy = (g1y*(fip1->u - fim1->u) + ....
        dudz = (g1z*(fip1->u - fim1->u) + ....

        double source1 = -(dudx*dudx + dvdy*dvdy + dwdz*dwdz) + ....
        double source2 = cip*fip1->p + ....
        double incr = (source1 - source2)/c0 - flow->pget(i,j,k)->p;
        double fpres = flow->pget(i,j,k)->getP();
        flow->pget(i,j,k)->setP(fpres + omega_sor*incr);
      }
}
```

Figure 4.1: Comparison of the Coding Efforts with and without the Framework

4.2 Calculation Examples

In this section, several calculation results are presented. All the results are obtained by the programs built by using the framework. The first example shows that the same field, conditions, and the identical implementation of numerical scheme, can be applicable both to sequential and parallel calculations without a modification.

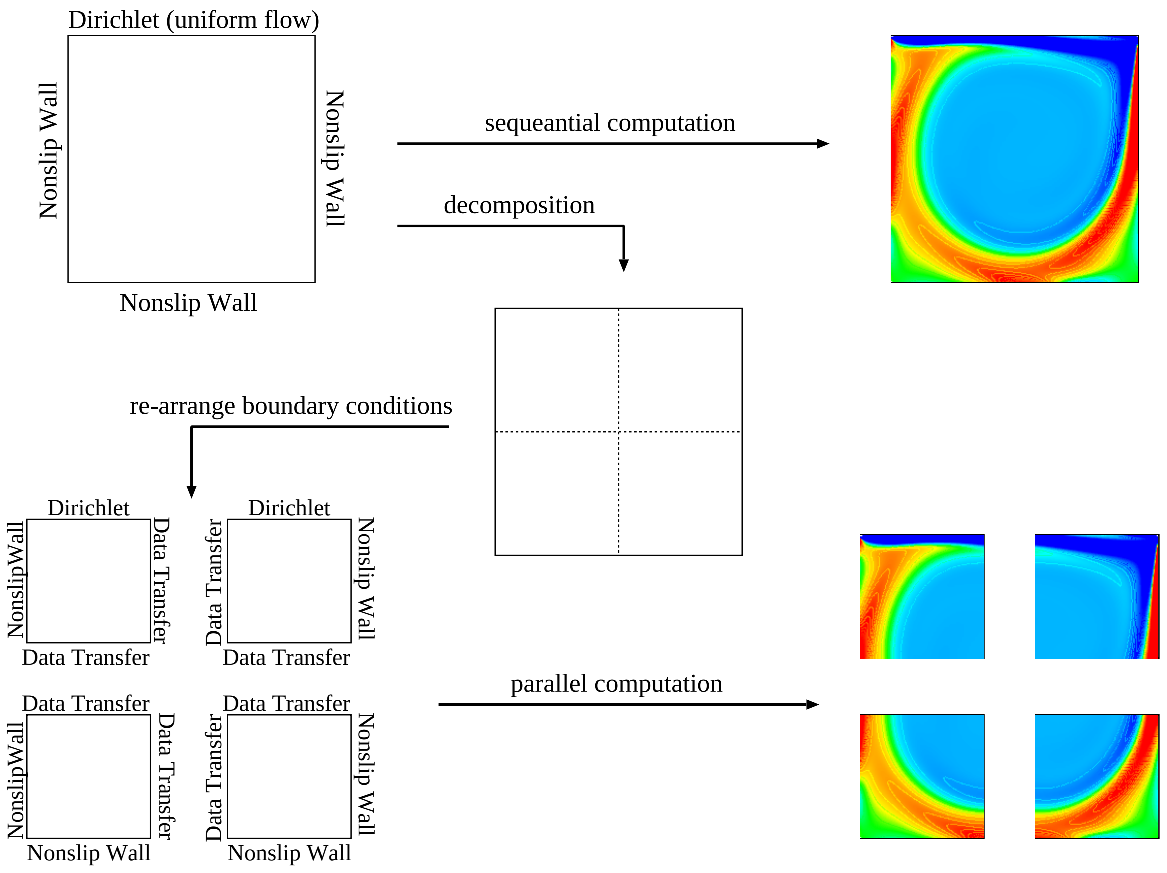


Figure 4.2: Sequential and Parallel Calculation

Figure 4.2 shows the process of the calculation by taking a two-dimensional cavity flow problem as an example. The rectangle at the left-top depicts the calculating region. The boundary conditions are defined at the four edges respectively. The sequential computation is applied to that field object as it is. For parallel computation, the field is decomposed and the boundary conditions are reconstructed for each of the decomposed region. Figure at the center shows that the field is decomposed into four regions in this case. The decomposed regions with the reconstructed boundaries are shown at left-bottom. The solver program applied to each of these fields and are run in parallel. The result of parallel calculation is shown as Fig. 4.3.

The programs for these sequential and parallel computings are written as followings.

```
// ------- sequential version -----------//
int main( int argc, char** argv )
{
  typedef IncmpUpwind2d<Incompress2d<> >   UpwindScheme;
  typedef SolverMAC<UpwindScheme>         MacSolver;

  Cavity2d generator;
  generator.initialize();
  generator.generate();
  generator.finalize();

  Field* field = generator.getField();

  MacSolver *solver = new MacSolver;
  field->accept(solver);
  solver->initialize();
  solver->calculation(nloop);

  delete solver;
  return 0;
}
```

```
// ------- parallel version -----------//
int main( int argc, char** argv )
{
  typedef IncmpUpwind2d<Incompress2d<> > UpwindScheme;
  typedef SolverMAC<UpwindScheme>       MacSolver;
  int ndiv1 = 2, int ndiv2 = 2;
  int nproc = 0;

  Communicator *mp = new CommMPI;
  mp->initialize(argc,argv,nproc);

  Cavity2d generator;
  generator.initialize();
  generator.generate();
  generator.finalize();

  Field* field = generator.getField();

  StrDecomposer2d<Incompress2d<> > decomposer;
  field->accept(&decomposer);
  decomposer.initialize();
  decomposer.setCommunicationInterface(mp);
  decomposer.setOverlapsize(2);
  decomposer.setDecompsize(ndiv1,ndiv2);
  decomposer.generateAttribute();

  Field *dfield = decomposer.getField(mp->id());
  MacSolver *solver = new MacSolver;
  dfield->accept(solver);
  solver->initialize();
  solver->setCommunicationInterface(mp);
  solver->calculation(nloop);

  delete solver;
  delete mp;
  return 0;
}
```

As these two programs indicate, parallel computation is realized by adding several modules related with the parallel execution. Adding the modules is done at the programming of the uppermost-level. Therefore, no modification is required inside of the modules like Solver and Scheme. In the example program, MPI is used for the communication library. In order to make the program available to PVM environment, it will be achieved merely by exchanging the object of the communication library. The part of the program is currently written as follows.

Communicator *mp = new CommMPI;

PVM can be used as the communication library, by changing this line to the following.

Communicator *mp = new CommPVM;.

The next example shows that fields of different geometry and condition are used in the same programming composition. The field is for a flow around a two-dimensional circular cylinder. The rearrangement of the boundary conditions can be processed similarly, even if the region possesses a non-mathematical condition like a periodic boundary condition (Fig.4.4). The calculation results are presented at Fig. 4.5. Flows at different Reynolds number are shown. Each calculation well realizes the characteristic of the flow at a certain Reynolds number. Here, the incompressible Navier-Stokes equations are solved by a finite difference method, with the MAC scheme's formulation. The program for the calculation is different only at an instance of Field class. The result shows that no unnatural discontinuation is introduced by the decomposition and parallel calculation.

The calculation with a three-dimensional field is also presented in Fig. 4.6, 4.7, and 4.8.

If different codes are using the same data structure, they can share many programming aspects as their common parts. Therefore, a compressible flow problem can be calculated in the same composition by replacing the variables and the scheme, as it is indicated in Fig. 4.9. This indicates that less programming effort is required with the framework.

Several benchmarking calculations are carried out to examine the parallel performance. The code is a finite-volume solver for compressible flow, that uses MUSCL and the Runge-Kutta scheme of fourth-order. The code includes a grid generating process, which is also performed in parallel. Calculations are carried out on several platforms such as RS6000/SP, SR2201, and Linux-running PC.

Figure 4.10 shows the speedup of the calculation's performance along with the number of nodes. The maximum number of the nodes is 48. The calculation is done on RS6000/SP, and MPI is used for the communication library. The grid size is not so large and is limited as 300 by 300, so that one node can handle it. The lines in the graphs represent the speedup of the main calculation, the grid generation, and the entire process, from the top respectively. All graphs show that the decent performances are obtained by parallel computation. It is observed by a calculation with a small grid size that the speedup ratio is declined at a larger number of nodes, since the ratio of calculation to the data transfer becomes smaller. Figure 4.11 is the performance of the code that is different only at the communication library; PVM is used. The graph shows the similar tendency of the performance, but the speedup of the total process is worse than the former case. The reason for this result resides neither in the framework nor in the algorithm, but it is caused by the computing environment with PVM. When a parallel code is executed on RS6000/SP's PVM system, pvmd, which is the PVM daemon, is launched and it again launches the local processes in every nodes. The periods consumed for this initialization are varied at different nodes. When the consumed time for the entire process is in a range of 100 seconds, the variety of the time for the initialization process as large as 10 seconds affects the speedup ratio badly. However, after the processes are once started, the real computing process of the grid generation and the calculation show a good curve of speedup. This suggests that the performance will be improved with a larger scale of calculation, since the ratio of the incoherent initialization process becomes ignorable. Figure 4.12 shows the CPU time of the calculations. Since the programs are identical except for the message passing modules, the amounts of the calculation time are similar. Therefore, the total calculation time is affected only by the difference of the message passing environment like the aforementioned aspect.

Figure 4.13 shows the results of the calculations that are carried out by RS6000/SP and SR2201. These results are taken by the code that is designed on the same basic concept, but every parts of the code are written from scratch. However, the very same basic concept is employed so that the separation of the parallelization procedures and the main calculation algorithm is realized. Two different size of grids, 1,200,000 (400 x 300) and 3,000,000 (600 x 500) points, are used for the calculation here. The quasi-linear speedup can be observed up to the use of 64 nodes.

A superior parallel performance is expected by a calculation of a much larger grid scale. For examining this, the calculations of a grid with one million (1000x1000) points are performed. The speedup is shown in Fig.4.14. Since it exceeds the ability of one local node, calculations are performed with more than 12 nodes. The speedup ratio is plotted by taking the performance of the calculation with 12 nodes as the unit. It would be different from the speedup based on the calculation on one node. Though the above condition must be noticed, a splendid speedup is obtained as it almost goes the ideal curve. This performance is resulted by the improved ratio of the computational loads of the main calculation and the data transfers. In addition, the program's design contributes to improve the performance, too. The framework forced a whole calculation process to run in parallel, by making it as a module. This achieves a large granularity of parallelism than the one obtained by parallelizing part by part.

The calculation is also carried out on a platform of different architecture. The system is PC based cluster system. Each node of the cluster system is SMP (Symmetric Multi-Processing) computer that holds two CPUs. The CPU is Intel's Pentium II (300Mhz). The cluster consists of eight of such nodes, and 100BaseT Ethernet is used for the network between these nodes. The performance is observed as Fig.4.15. The code is the identical one that are used for the results shown in Fig. 4.10 and Fig. 4.11. MPI is used for the communication library, but PVM gives a similar performance. The graph shows a quite good linear speedup, as seven and half times of speedup is achieved by eight nodes. Currently the calculation is performed by using a message passing library at both of inter-node communication and inner-node communication. It is observed that the performance of one SMP node is lower than that of two networked CPUs. It is expected that the former is faster since the data is transmitted far faster by the inner bus than by the Ethernet. However, the result is on the contrary to that expectation. One benchmark shows the former took 1135 seconds while the latter completes the same calculation by 1318 seconds. The memory contention is considered to be the main reason for the phenomenon, and it seems a common feature of SMP systems. For a solution to the phenomenon, using a threaded process can be considered for the calculation task performed within a SMP node. However, it generally requires a very subtle and difficult programming, for two different levels of parallelization techniques are intertwined. The framework can also help such situation by realizing a threaded solver object. The framework design for such programming is discussed more in later section.

The benchmarks are also carried out by an incompressible flow solver. The solver adopts the MAC (Marker and Cell) scheme for the formulation of the Navier-Stokes equations, and a 3rd-order upwind scheme for the nonlinear terms. SOR (Successive Over-Relaxation) method is used for solving the Poisson equation for the pressure field. The Euler implicit integration is used for the time-marching loop. The performance is shown in Fig. 4.16. It can be observed that the speedup is almost saturated at 16 nodes. However, the grid generating process keeps a good speedup until up to 32 nodes.

What makes the performance worse in this case is considered as the ratio of CPU time costs in the data transfer and the calculation at one loop. Figure 4.17 shows how that ratio affects the parallel performance. Consider two calculations that use same size of data, but each has a different amount of calculations in one calculation loop. The cost for the data transfer is the same since the data sizes are identical. When these calculations are performed in parallel in two processes by decomposing the data into two regions, the total calculation time for each calculation is reduced, as the figure illustrates. It is apparent that the speedup ratio is greater when the original calculation amount is larger, since the amount of the data transfer is identical. When comparing the solver for compressible flow and the one for incompressible flow that are used here, this explains the reason of the achievement of a better speedup by the former code, since it has a heavier calculation process than the latter. In order to prove this, we adjusted the SOR loop, so that it performs the data transfer at every five loops but not at every single loop. This enlarges the ratio of the computational load of the main calculation compared to the one of the data transfers. This must improve the parallel performance. The result is shown in Fig. 4.18. It is seen that the performance is greatly improved by this modification. A calculation with a larger grid size would further improve the performance. However, most of the codes that employ an implicit scheme would have a similar tendency. An innovative algorithm that is originally designed for parallel computation is expected in this area.

A calculation is also examined on the other platforms like Cray's T94. Cray T94 has a vector processor, and the C++ compiler supports a vectorization. However, the tests indicate that the code is not vectorized at all. The reason is suspected as followings. The compiler tries to vectorize a code where a loop repeats the same calculation with the primitive data types. The framework with an object-oriented approach writes such inner-loop calculation by arithmetic operations between objects. This manner of programming may harm the compiler to apply the vectorization. A vector processor shows a same or less performance than a standard CPU for PC, if the vectorization facility is not applied properly. One of the solution for using the machines of vector processor will be preparing a different version of the calculation module for the vector processors. This change does not affect the framework's design and the programs of the rest parts, since the difference is confined to the implementation within a method.

Another topic on the calculating performance is the overhead introduced by the use of the framework. Since the framework is designed and implemented by an object-oriented approach, there must be some overheads in accessing the data and the methods. We carried out a testing calculation to estimate the overhead. The result is shown in Fig. 4.19. One program is written entirely in C without using the framework, while the other is developed with the framework in C++. The programs are for solving the three-dimensional incompressible Navier-Stokes equations. The solver employs SOR for Poisson equation, and the implicit Euler scheme for the time integration. The number of loops for the implicit schemes is fixed so that both codes have the same calculation amounts. The result shows that the performance of the non-framework code is almost a double of the performance of the code written by using the framework. This result can be interpreted as fast enough, for the performance comes within the double of the manually elaborated code's performance. Or, on the contrary to that, it can be regarded worse enough when considering the use for the scientific computing purpose.

The followings are considered as the causes of the overhead.

- multiple tracing of pointers for accessing an object
- generation and deletion of temporary objects in object's operations
- compiler's ability for optimization

Making some methods inlined and setting a temporary reference to a frequently used object would improve some of the above problems. There is also a technique called as ”Expression Template” for the second problem, which is mentioned in later section. Some common techniques like ”unrolling” can be used for the program fragment for calculation scheme. If these efforts on the optimization is applied, it is expected that the performance will be improved. One great advantage of the framework in applying such optimization techniques is that these efforts can be confined to a class's implementation. Therefore, optimization efforts done to some classes will not cause a reconstruction of the rest of the program.

The last problem about the compiler is improved by using the other language at least to the part of coding dedicated to the main calculation. In order to appreciate the merits offered by the object-oriented approach, we recommend using multiple languages than rebuilding the entire framework by the other language. One example of using multiple languages is to implement the overall framework in C++, and to implement a computing-intensive part in FORTRAN. In order to estimate the effect, we carried a benchmark with a very simple calculation with an arrayed data. The result is shown in Fig. 4.20. The test calculation is simple enough that no framework's overhead is expected here. Therefore, the result mainly indicates the difference of the compiler's ability. The result suggests that the use of FORTRAN to the computing-intensive part will somewhat improve the entire performance.

As the summary, our opinion on this matter is denoted as follows. It is not the problem of choosing whether to use the framework or not, in order to achieve the good performance. Since we would like to appreciate the merits of the framework, what matters here is the performance of the code that is written by using the framework. As the probable candidates for causing a low-performance are listed, the performance can be improved by applying the appropriate modifications as suggested above. Thus, our decision is not to redesign or abandon our approach for the framework. The performance should be pursued within the use of the framework.

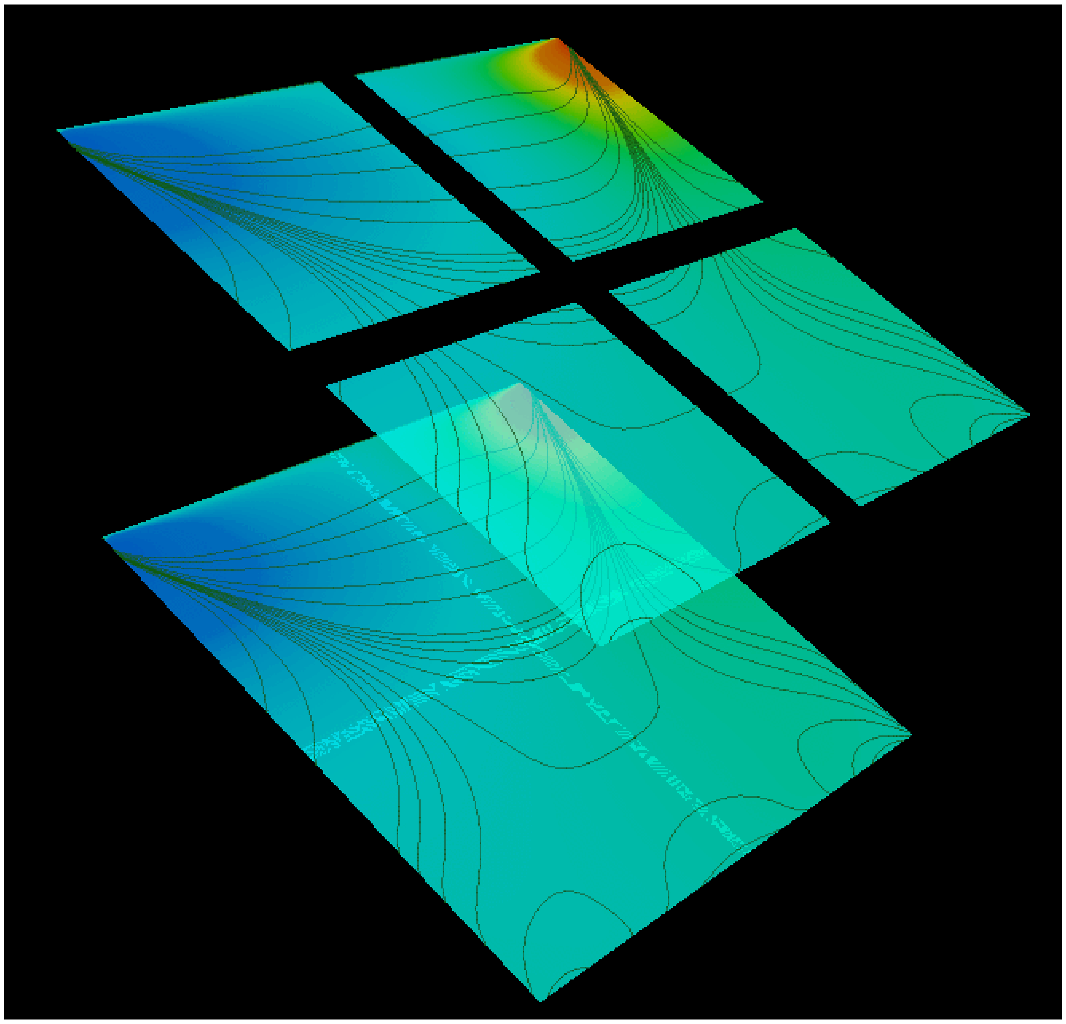


Figure 4.3: Parallel computation of two-dimensional cavity flow

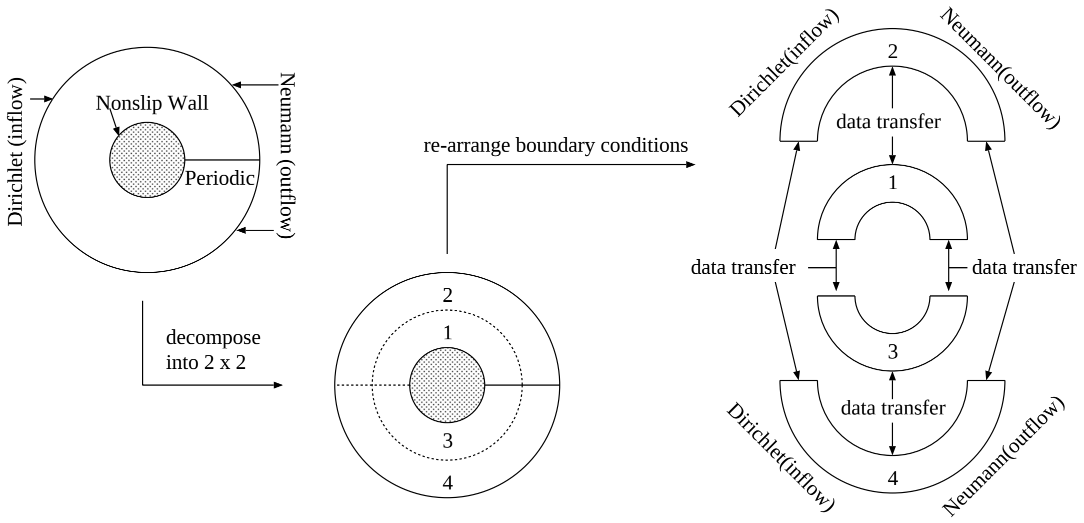


Figure 4.4: Rearrangement of Boundary Conditions

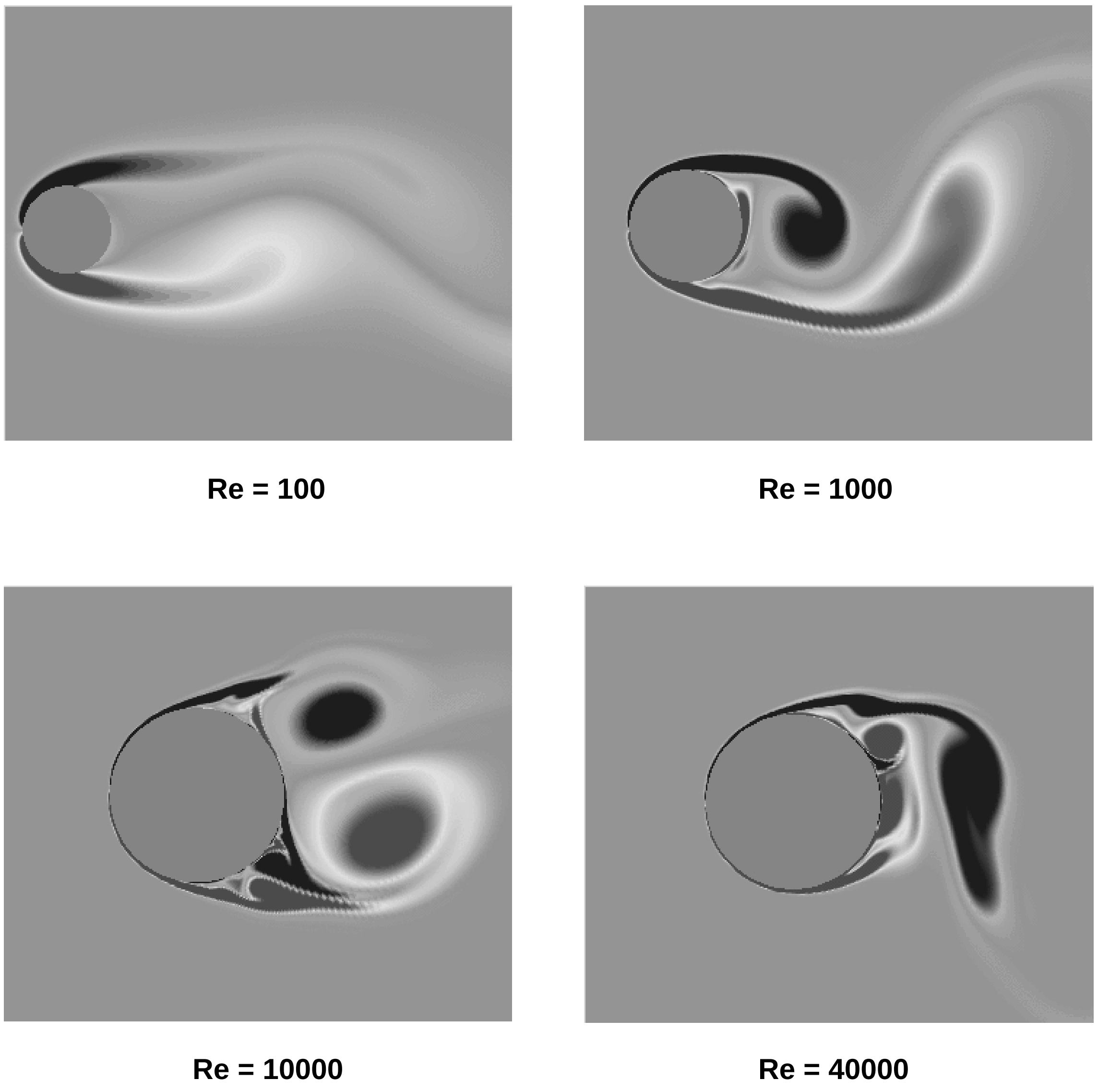


Figure 4.5: Incompressible Flow around Circular Cylinder

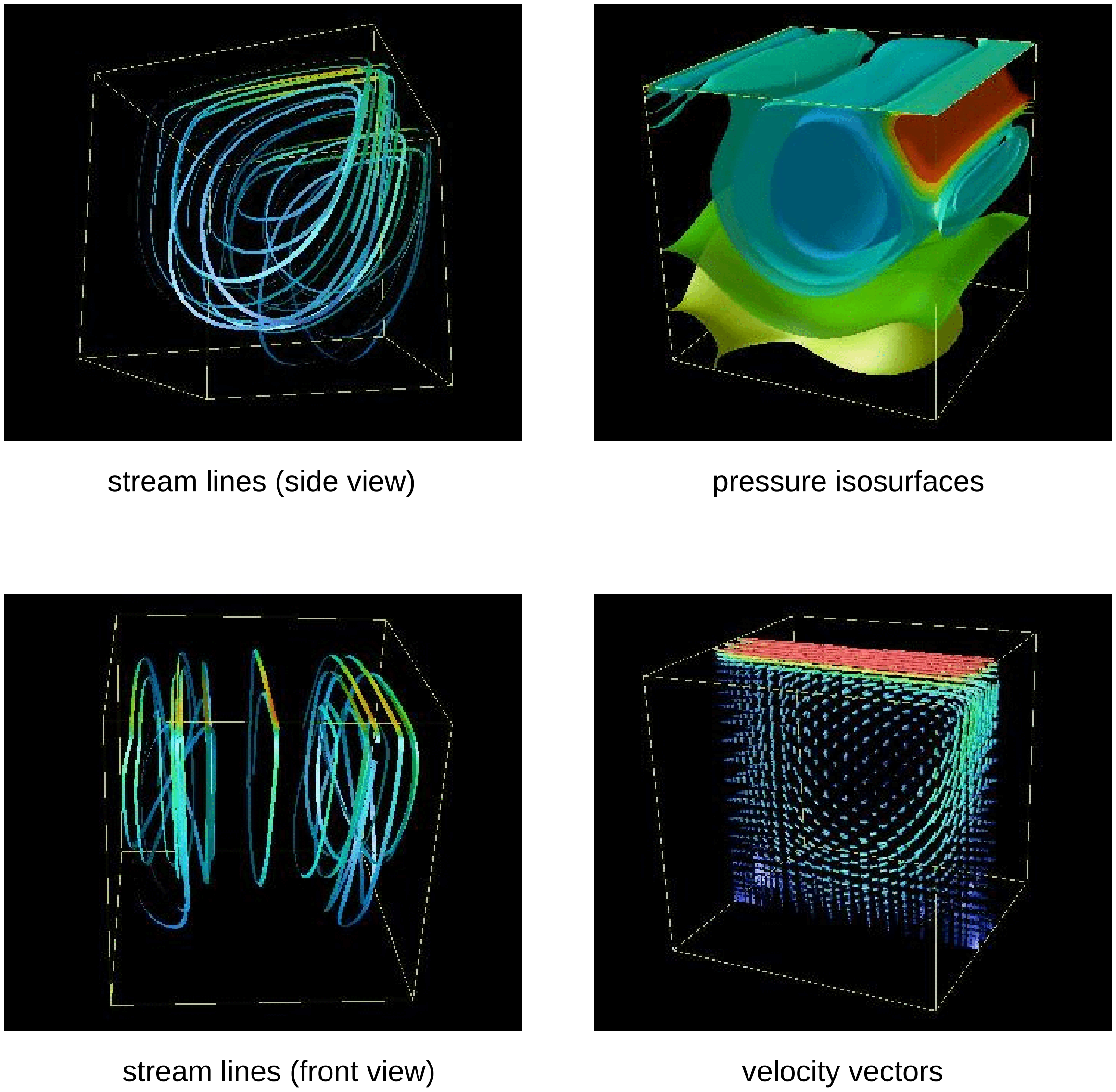


Figure 4.6: Three-dimensional cavity flow (Re = 1000)

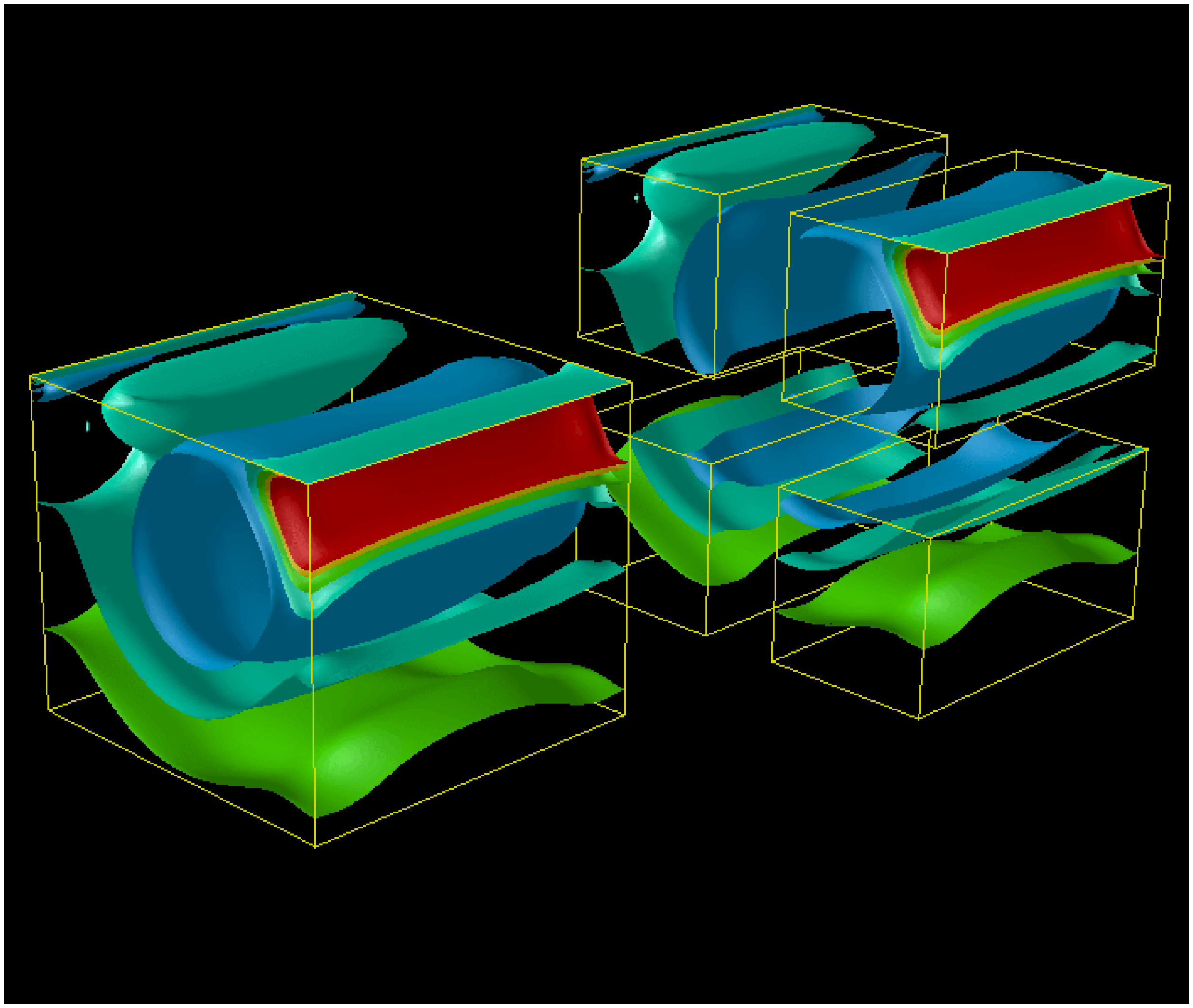
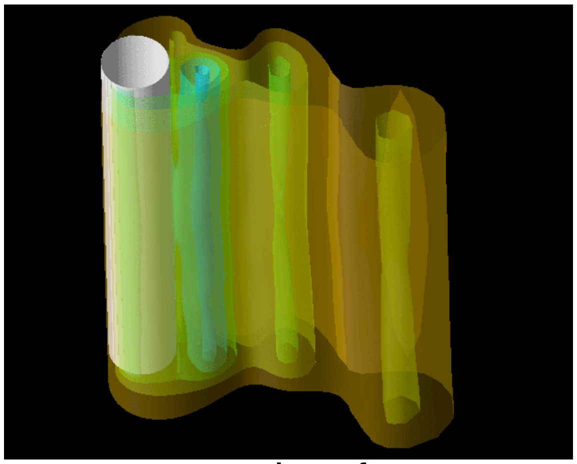
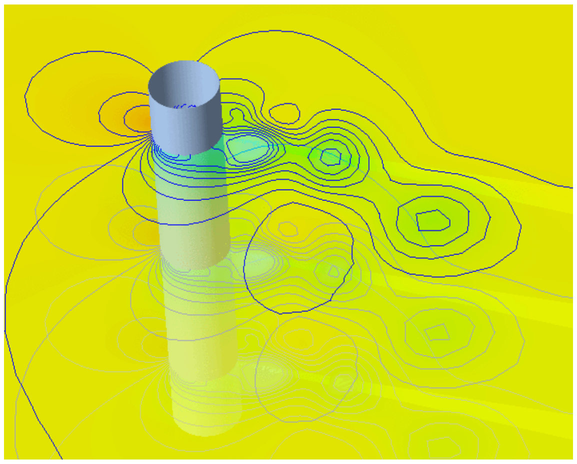


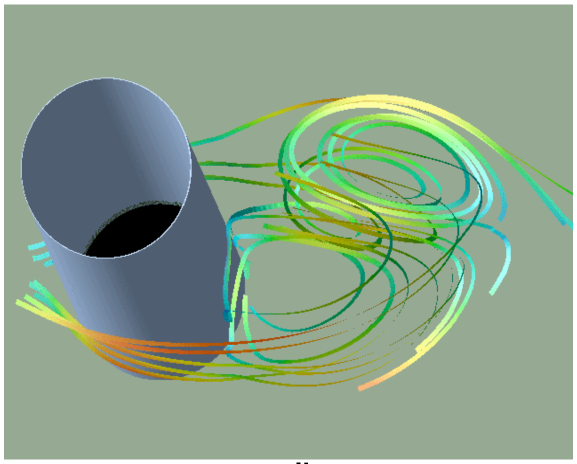
Figure 4.7: Parallel computation of three-dimensional cavity flow (Re = 1000)



pressure isosurface



pressure contours



stream lines

Figure 4.8: Tree-dimensional flow around circular cylinder (Re = 1000)

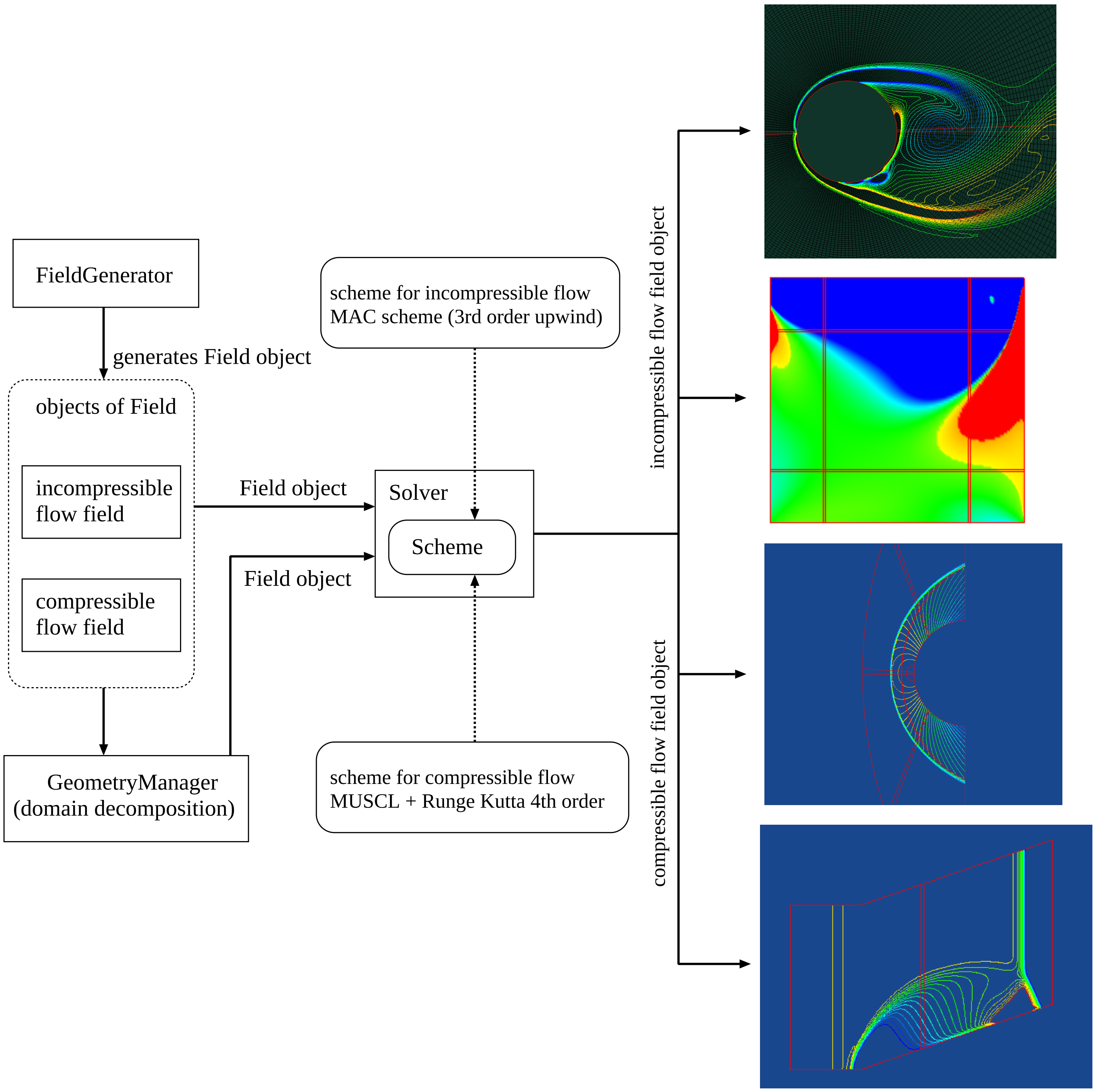
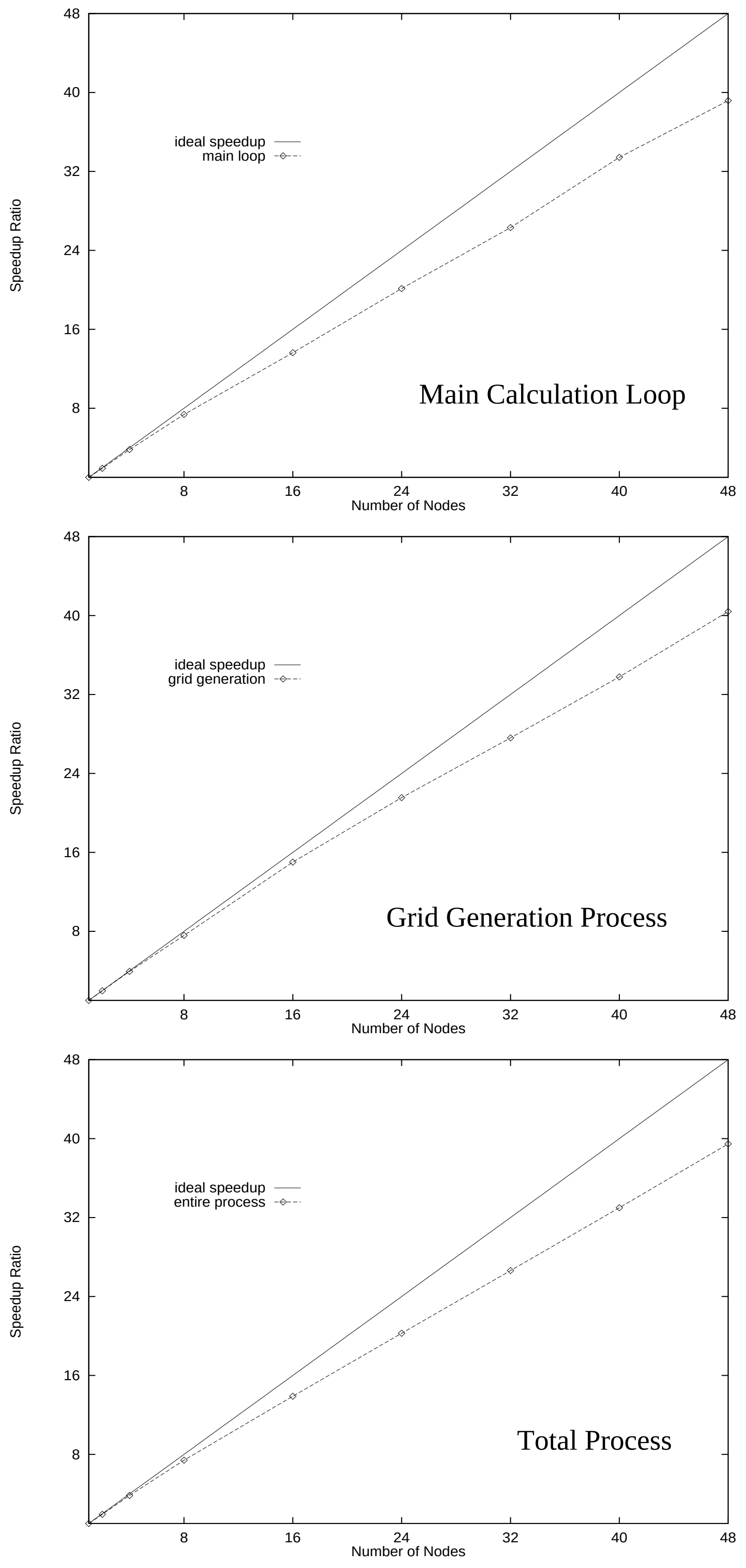

FieldGenerator
generates Field object
objects of Field
incompressible flow field
compressible flow field
GeometryManager (domain decomposition)
Field object
Field object
scheme for incompressible flow MAC scheme (3rd order upwind)
Solver
Scheme
scheme for compressible flow MUSCL + Runge Kutta 4th order
incompressible flow field object
compressible flow field object

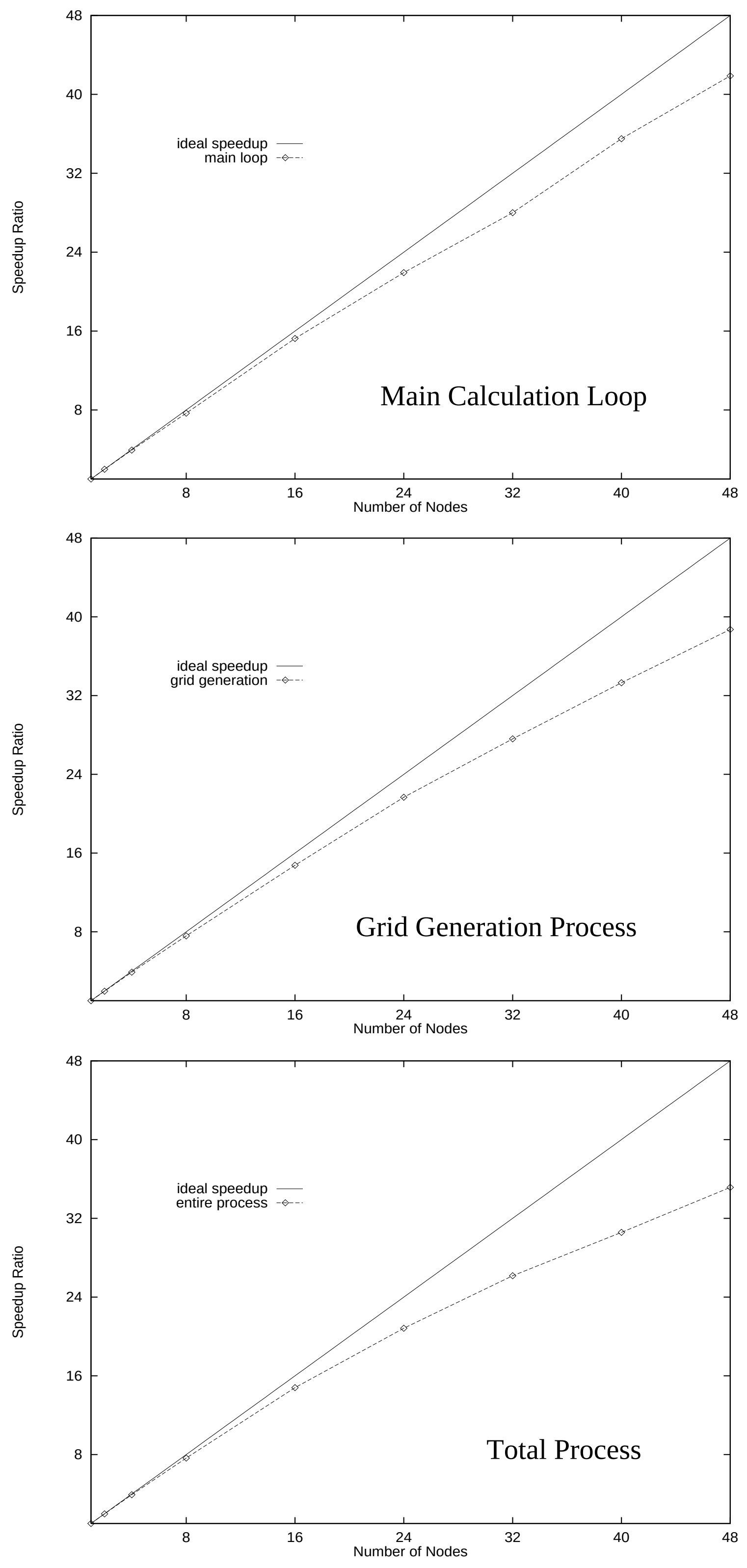
Figure 4.9: Calculations of Different Problems and Conditions



Numerical Scheme:
2D Euler equations
Runge-Kutta 4th
MUSCL-TVD

Condition:
grid size: 300x300

Figure 4.10: Speedup Ratio by RS6000/SP with MPI



Numerical Scheme:
  2D Euler equations
  Runge-Kutta 4th
  MUSCL-TVD

Condition:
  grid size: 300 x 300

Figure 4.11: Speedup Ratio by RS6000/SP with PVM

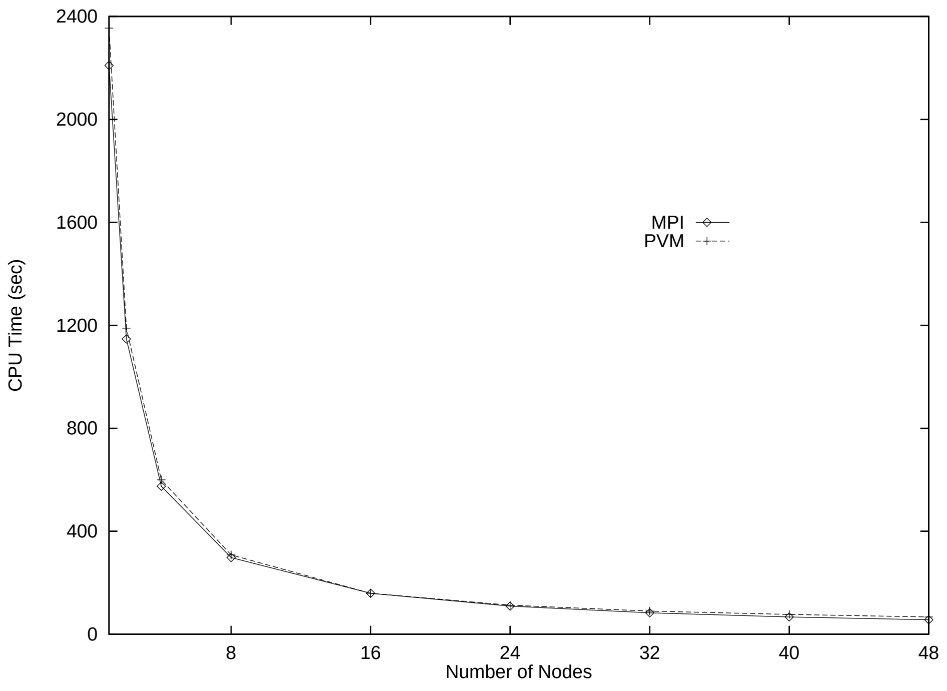


Figure 4.12: CPU Time

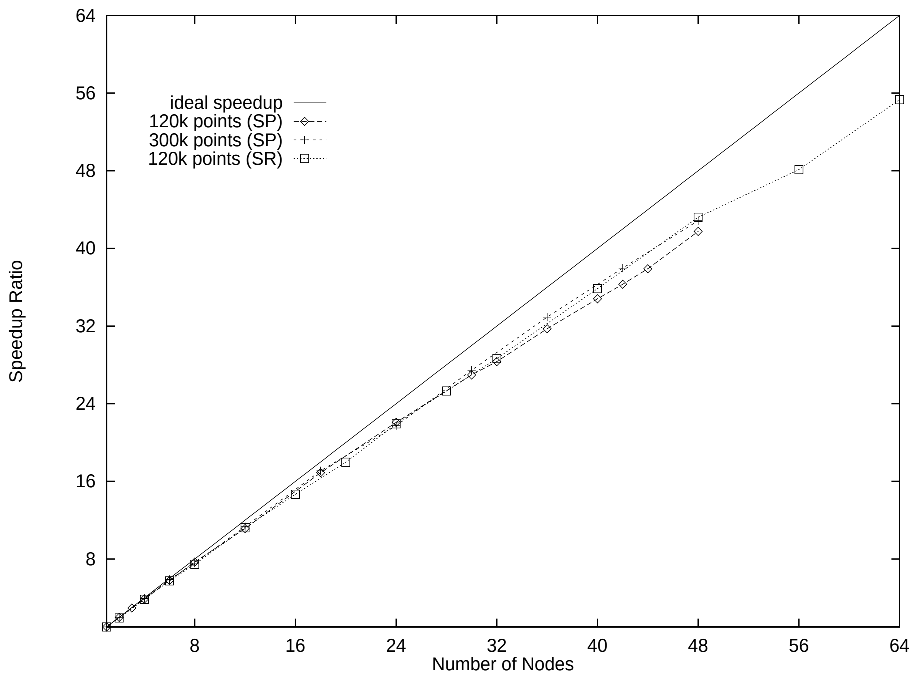


Figure 4.13: Speedup Ratio on RS6000/SP and SR2201

Numerical Scheme:
2D Euler equations
Runge-Kutta 4th
MUSCL-TVD

Conditions:
grid size: 1000x1000

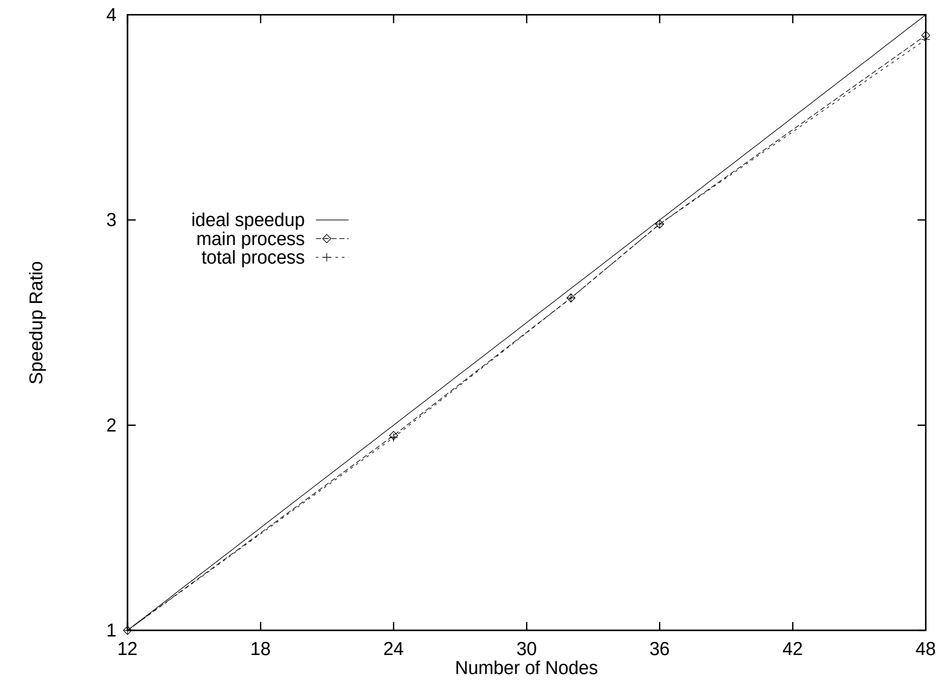


Figure 4.14: Speedup Ratio with Larger Grid Size (RS6000/SP)

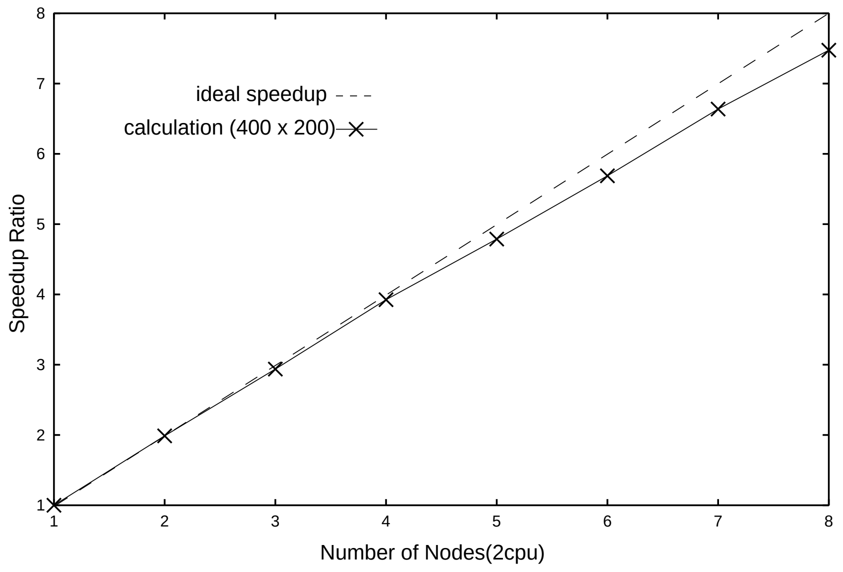
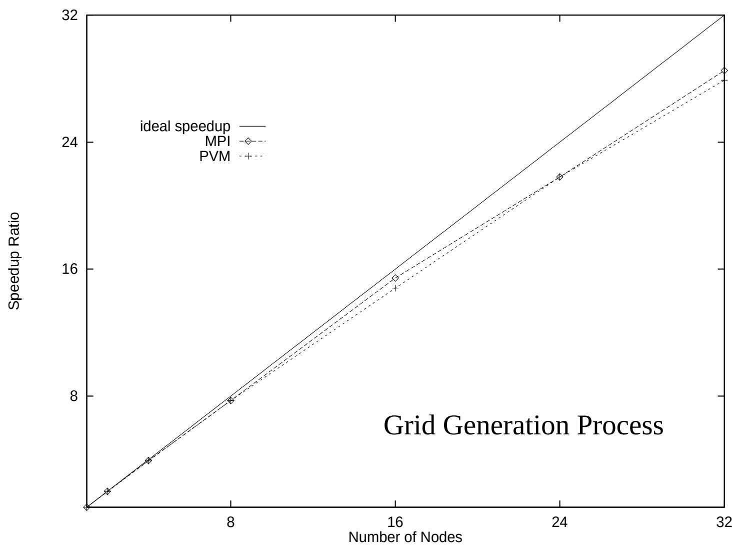
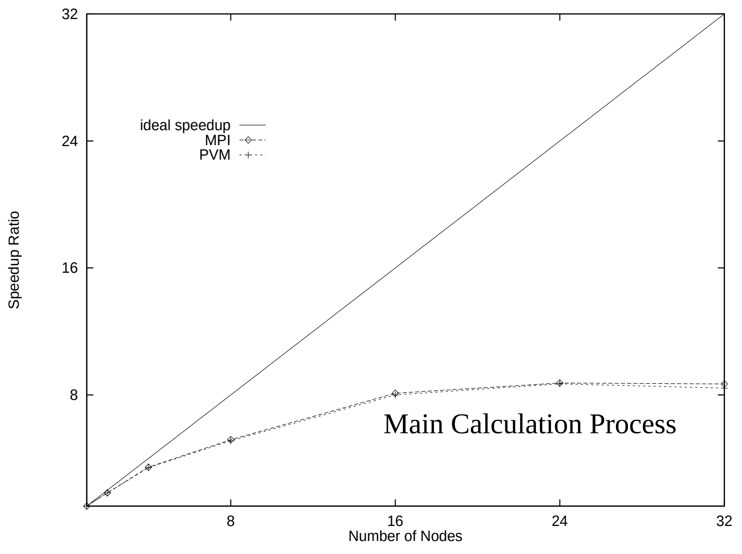


Figure 4.15: Speedup Ratio with PC Cluster





Numerical Scheme:
  2D Incompressible N.S.equations
  3rd order upwind for nonlinear term
  SOR for Poisson solver
  Euler Implicit for time marching

Condition:
  grid size: 300 x 300

Figure 4.16: Speedup Ratio with Incompressible flow solver (1)

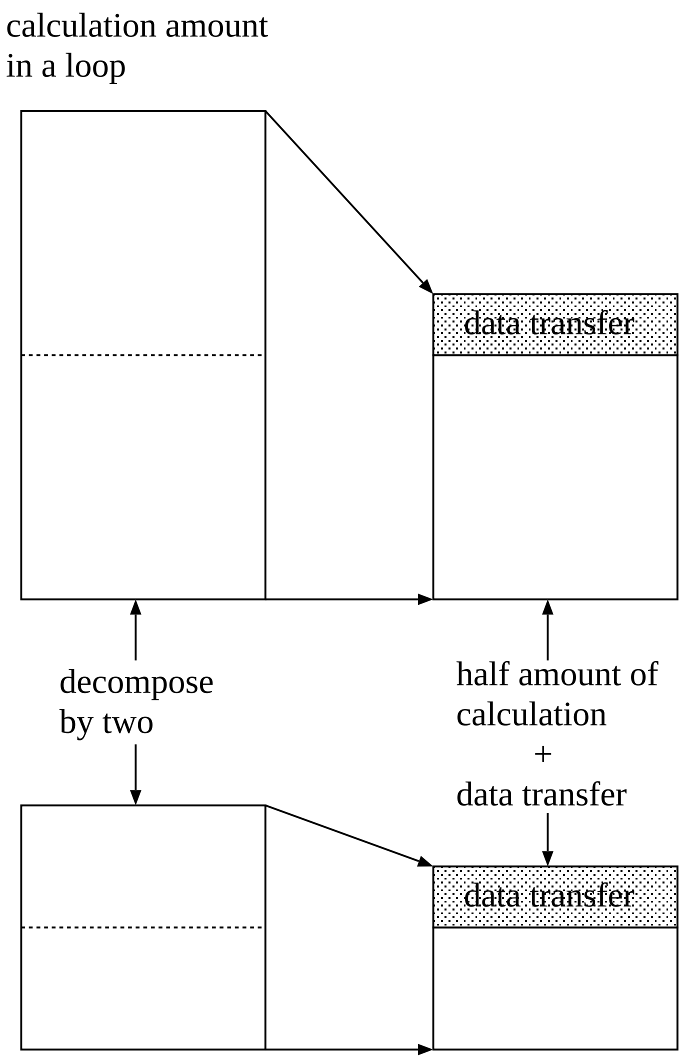


Figure 4.17: Effect of Data Transfer Ratio to Calculation Size

data transfer is occured
at every 5 relaxation loop

grid and condition
is the same with
the calculation
for Figure 4.16

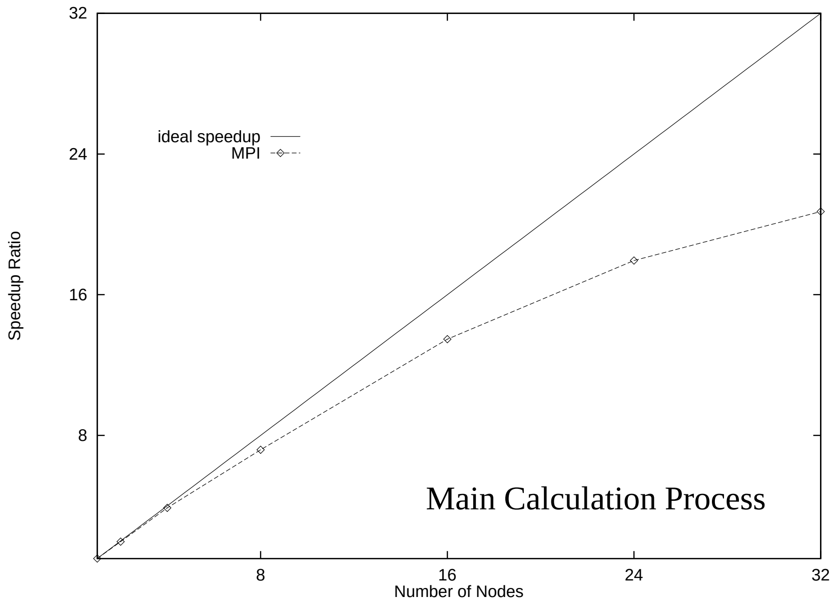
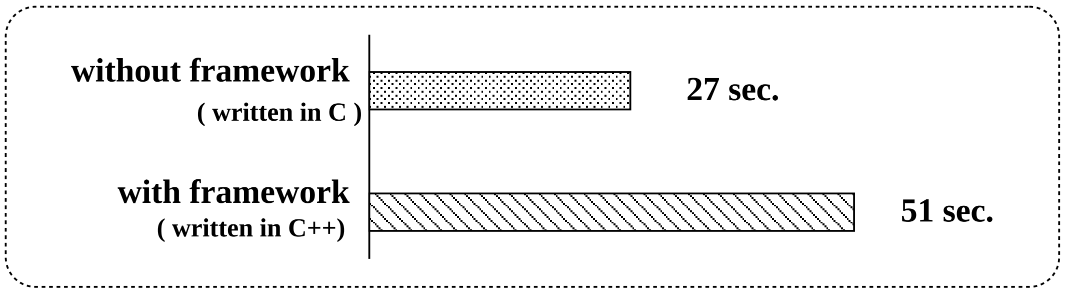


Figure 4.18: Speedup Ratio with Incompressible flow solver (2)



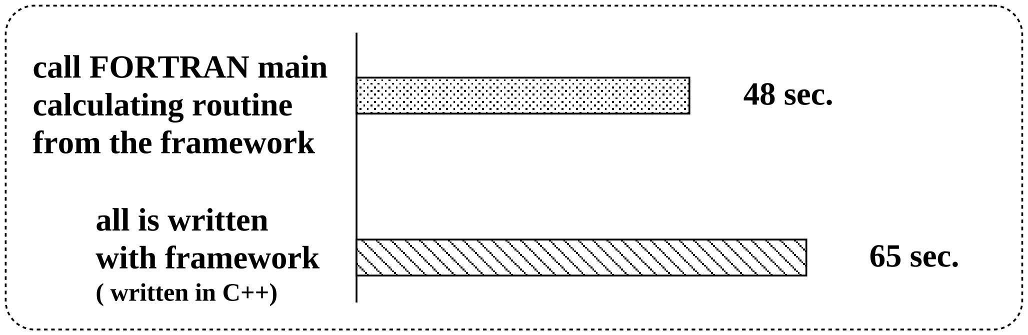
solver

three-dimensional incompressible Navier-Stokes equations
SOR for poisson equation
3rd-order upwind scheme for non-linear term
Euler implicit integration for time marching

conditions

sequential calculation (no parallel execution)
grid size is 27,000 points (30 x 30 x 30)
the number of converging loops for SOR and implicit time marching are both fixed to 1
results of calculations after 10 time step
performed on Linux 2.2.10 on Intel PentiumII 300 Mhz

Figure 4.19: The Approximate Overhead in Use of the Framework



conditions

calculations of repetitive substitutions with 3d arrayed data
array size: 100 x 100 x 100
the number of loops: 1000
performed on Linux 2.2.10 on Intel PentiumII 300 Mhz
compilers: g++ and g77

Figure 4.20: Perforamance Improvements by Use of Different Language(FORTRAN)

Chapter 5

DISCUSSIONS

5.1 Programming

The examples presented in the previous chapter show the various aspects that can be achieved by the framework. One of the original intention of this work is to eliminate a parallelization task from writing a program for numerical algorithm. This objective is achieved by the framework's design that makes the data module being independent from any of the procedures.

The most common programming approach taken for obtaining a parallel program is processed by the approach generally referred as parallelization. A sequential program is modified into a parallel program by rewriting every part where parallelization should be applied. The objective of this parallelization approach is to make an exiting program available for parallel computing. The demerits of this approach are as follows. First, it requires special knowledge and experience of parallel computing. Second, the rewriting task must be applied to every different code, since parallelization tends to be specific to each particular program. By the same reason, it is difficult to assure the program's reusability and portability. More fundamental problem by this approach is that the algorithm itself remains as it was originally devised for sequential computation, through this modification procedure. The parallelization will not alter the algorithm into one that is fundamentally designed for parallel computing. It only modifies a program to a code that can be performed in parallel. It even could be said that devising a genuine parallel algorithm is abandoned by this parallelization approach. For example, it is generally observed that an implicit time integration scheme does not show a good performance by parallel computation. This is because the ratio of the time consumed by the communication is high, compared with the time for the calculation. This will not be a serious problem with a large-scale calculation with a computer that installs a high-performance networking facility. However, it would be ideal to have an algorithm that is originally designed for parallel computing. The approach taken here suggests, or rather insists, to design a parallel algorithm from the beginning, but not to parallelize a sequential one. It will achieve a program with a larger granularity of parallelism, and would have a higher performance.

Many object-oriented approaches have been proposed for reducing the difficulty in parallel programming. Most of them try to do that by preparing an intelligent functionality by a language's grammar or an object. A typical example is an array object that behaves just like the array of HPF (High Performance FORTRAN) [30]. The parallelization procedure is encapsulated and hidden inside of the object. Therefore, such an object gives a convenient usability for users to compose a parallel program. However, in respect of program design, such objects change the means only, by which a parallelization programming is done. This is because their principle stays on the parallelization approach. Tools, grammars, or libraries differ only at the ways of realizing the parallelization and will not alter the programming paradigm, as long as they are devised by the same concept. Our approach does not adopt the parallelization as its principle for designing a program. The framework is designed to support writing a parallel program from the beginning. Some might say that the same benefits could be yielded even by the traditional programming approach, by setting proper subroutines. However, it is dubious if that will be achieved as far as the principle remains as the parallelization approach. What is significant is the approach or concept of how the programming is composed, but not the means to do it. Therefore, well-designed subroutines will yield the same advantages as our framework, if they are designed by the same basic concept as ours. However, such attempts would eventually require an object-oriented design, and will be achieved more conveniently by an object-oriented language.

5.2 Reusability, Portability, and Spcialty

As being an object-oriented approach, the framework is expected to improve the program's reusability and portability. Moreover, it has the benefits by being designed particularly for CFD use. Many object-oriented approaches prepare a class library of a low-level functionality of higher abstraction, such as data structures like array and list, and the interfacing objects to these modules. A program is composed by using the classes provided by the library as building parts. Therefore, it requires more consideration on the design of the program when such parts are of low-level functionality.

With this framework, the classes, which correspond to the conceptual units that appear in typical CFD programs, are provided. These classes are the ones like grid, flow variables, boundary condition, a scheme, and so forth. This correspondence will help the programming, for a program is composed by combining these modules straightforwardly. Writing one's own algorithm by the framework is done by using the class, which offeres a template for programming. The most of the classes that correspond to the CFD's conceptual units act as such template. By this arrangement, an achievement of a research effort like a new scheme can be made an interchangeable module. This aspect suggests that many researches can be assembled by a common infrastructure once a standard framework is established. Having program's portability offers us a great convenience in programming process. It enables us to develop a program on a personal workstation that has a greater interactivity of operation. The interactivity is a great advantage for the process since a program is frequently compiled and performed for debugging. Once a program is completed, that same program can be used for a large-scale calculation, with a supercomputer.

On the other hand, there are cases that having a special arrangement benefits more than portability. One such examples is a system with distributed SMP (Symmetric Multi-Processing) nodes. It is generally observed that the parallel performance within a SMP node is not good due to a memory contention. Adopting threaded processes within a SMP node would be a solution to this problem. However, this arrangement demands employing of two different parallelization methodologies in one program, for a message-passing approach is expected for the parallelization between the nodes. Thus, this example shows a case that needs a special design for a program. One other example is a cluster system of heterogeneous nodes. To make it work most effectively, an uneven load-balancing is required. This is different from the normal strategy of load-balancing that distributes the entire load evenly.

The requirements for portability and specialty are regarded as the opposing concepts. Therefore, it is difficult to manage them by one approach, and consequently they are tended one by one.This framework can handle such opposing requirements in a smart way. Every procedure is made into a module, which affects the calculation independently only through the changes in the instance of a field module. Therefore, adding a new procedure does not affect the other modules in view of the programming. In this arrangement, portability is supported by the core modules, while spcialty is achieved by add-on functions. For the case of a SMP-node cluster, a solver module that becomes a thread process by itself can be introduced. This is done by preparing a thread class like one offered by Java, and implementing a solver as a derived class of that. The functions for threading are implemented in the parent class only. Therefore, the solver class is not affected for supporting the thread functions. For the case of a cluster of heterogeneous nodes, the uneven load-balancing is achieved simply by replacing the module for pre-process. A decomposition module can have information on each node's performance in order to achieve an appropriate load balancing, by preparing an evaluating module.

In these cases, it is recognized that introducing a special treatment will not affect the programming of the other modules. The special treatment for a particular environment is realized by preparing an add-on module that is responsible to the feature, while portability is brought by the portability of the solver and other core modules.

5.3 Performance Issue

The parallel performance of the program written with the framework is shown at the section 4.2. It shows a speedup by the number of CPU goes up linearly. The result is sufficient to demonstrate that the approach can yield a good performance on parallel computation. However, the parallel performance does not prove anything of the absolute performance. Actually, a manually elaborated parallelized program would show a superior performance in the absolute speed, especially when a fine-tuning specific to a particular computing environment is applied to the program. A framework like the one presented here defines a program's structure, and this aspect may harm the ultimate optimization. In addition, some object-oriented features like function overloading, polymorphism, and using of pointers introduce some overheads to the calculation. However, the main objective of this approach is to propose a new programming paradigm, but is not to pursue a peak performance. The framework is also designed to be an infrastructure for programming CFD code. The primary objective is to appreciate these benefits the framework offers. Therefore, what matters here is not the absolute performance of the code, but is its parallel speedup. On the other hand, the absolute performance can be pursued even with the use of the framework. This topic is discussed in the section 4.2.

There are aspects where this approach can contribute to the performance. Since it forces the main calculation program to have a parallelism as the whole, the program is expected to have a large granularity of parallelism. This aspect helps the code to produce a good parallel performance, though it ultimately depends upon how the algorithm is designed. With the domain decomposition, the treatment of data transfer contributes in improving the parallel speedup. In general, a program using domain decomposition technique has two separated procedures for applying boundary conditions and data transfers. Time is wasted when these procedures are performed sequentially, since the number of the boundaries bound to each procedure is different by each decomposed region. For example, some decomposed regions have two data transfer boundaries while some other regions have four of such boundaries. When a calculation process finishes its work for the data transfer, it must wait idly until others finish their duties, if it has fewer boundaries to perform data transfer. This will considerably harm the parallel performance. This framework treats both mathematical conditions and data transfers as the same concept, and performs them in the same procedure. Therefore, no incoherence is observed among the decomposed regions, regarding the number of boundaries performed in one procedure. This eliminates the idly waiting time, thus contributes to improve the performance.

There are also several approaches that might improve the calculation performance, staying to be benefited by the framework. The first proposition is to implement a computing intensive part in FORTRAN. The FORTRAN program is called from a method of a class. With this approach, all the other aspects like the program's structure remain to be supported by the object-oriented framework written in C++. By the framework's designing policy, implementing one part in a different language does not affect the implementation and the use of the other classes. Therefore, it can be said that this approach can offer simultaneously the benefits of the usability of the framework written in C++, and the performance that FORTRAN compiler will bring. This strategy brings another merit than the improvement of performance. It eliminates a programming difficulty with C++. The programming labor by the framework is almost identical to the programming of the main calculation algorithm, and that task can now be done with FORTRAN, which is the language familiar than C++ to the most of CFD researchers. The strategy even would make a legacy FORTRAN program do parallel computation, with the help of the framework.

The second suggestion is to implement the entire framework by different language that yields a superior performance. For scientific computation, FORTRAN would be the choice of the programming language. While FORTRAN77 has very poor ability for achieving the object-oriented concepts, Fortran90 and 95 have the better support in that respect. This alternative may achieve much higher performance. However, it should be noticed that even with FORTRAN, it cannot avoid being suffered by the overhead introduced by object-oriented approach.

There is a special topic about the performance issue especially for the use of C++, which also benefits this approach. One of the major features of object-oriented programming is the encapsulation. The programming details of class's behavior are hidden behind the interfaces. This feature enables applying a fine-tuning without affecting the usage of the class. One good example can be seen in a use of *Expression Template* [31, 32]. It is a technique for avoiding the generation of temporal objects in the operation between multiple objects. Here, we explain briefly the technique and its effect.

In FORTRAN77, the arithmetic operation like addition between two vectors is processed by performing the operation in every element of the vectors. For example, addition of two three-dimensional vectors can be written as follows.

```
X1 = X2 + X3
Y1 = Y2 + Y3
Z1 = Z2 + Z3
```

In C++, this can be written directly as follows.

```
vec1 = vec2 + vec3;
```

This way of writing is not only for the convenience of programming, but also contributes in reducing errors. However, at the same time, this notation becomes a cause of the worse performance. It is a common approach to prepare arithmetic operators that can be applied directly to the objects. In a normal implementation, C++ compiler generates the object to store the temporary result of the operation of two objects. With an arithmetic expression like A + B + C +..., temporary objects are going to be generated for every operation between two objects. This routine consumes a considerable time in generating, copying and deleting the temporal objects, and harms the performance badly. There is the technique called *Expression Template*, in order to prevent the generation of temporary objects. We prepare the classes using this technique, and compared its performance with the code by the standard implementation and ones

vector length	F77 (sec.)	F90 (sec.)	F90 array Op. (sec.)	C++ with ET (sec.)	C++ without ET (sec.)	ratio
3	0.01	0.03	0.28	0.05	0.66	13.20
5	0.02	0.03	0.30	0.05	0.67	13.40
10	0.05	0.06	0.35	0.08	0.73	9.13
20	0.08	0.10	0.43	0.12	0.84	7.00
50	0.18	0.23	0.66	0.23	1.11	4.83
100	0.35	0.43	1.08	0.42	1.65	3.93
300	1.04	1.26	2.68	1.21	6.09	5.03
1000	5.63	4.15	11.36	7.86	23.16	2.95
10000	28.27	20.66	46.24	39.92	194.98	4.88

Table 5.1: Performance by using Expression Template (1)

by FORTRAN. The results are shown in Table 5.1 and 5.2. The programs used here are listed in Appendix B.

Table 5.1 shows the result of addition of five vectors, for which the mathematical expression is written as $v1+v2+v3+v4+v5$. The programs are implemented in FORTRAN77, Fortran90, and C++. There are two versions of C++ code; one is adopting the aforementioned ”Expression Template” technique, while the other is not. There are also two versions of Fortran90 code, one is written by using the array operation, and the other is not. The calculation platform is Linux on Intel machine (Pentium II 300Mhz). The compilers are ”GNU g77” for FORTRAN77, ”VAST f90” for Fortran90, and ”KAI C++” for C++, respectively. In the table, the column of ”ratio” represents the performance ratio of two C++ codes. As the result shows, the *Expression Template* technique greatly contributes in improving the performance.

It can be observed that the performances of the code with the technique are constantly faster than the one by the code without the technique. The ratio is about three to ten averagely. This improvement brings the performance of C++ code closer to the one of FORTRAN code. The

vector length	F77 (sec.)	F90 (sec.)	F90 array Op. (sec.)	C++ with ET (sec.)	C++ without ET (sec.)	ratio
3	0.01	0.03	0.29	0.07	0.92	13.14
5	0.04	0.04	0.30	0.09	0.93	10.33
10	0.06	0.07	0.36	0.14	1.01	7.21
20	0.11	0.12	0.45	0.21	1.15	5.48
50	0.24	0.26	0.71	0.43	1.51	3.51
100	0.47	0.50	1.17	0.80	2.20	2.75
300	1.43	1.45	2.88	2.28	8.57	3.76
1000	6.80	4.79	11.84	10.91	34.71	3.18
10000	34.04	23.88	48.65	67.16	281.90	4.20

Table 5.2: Performance by using Expression Template (2)

difference is now within the range of double. It is even observed that C++ code can be faster than F90 code in some cases. Table 5.2 shows the results of different calculation; the operation performed is $2.0 * (v1 + v2 * v3 + v4 - v5)$. The results show the same tendency. These results must be depended on the environment like platform and compiler significantly. Other testing environments will show a different result. However, it can be safely said that using *Expression Template* improves the performance considerably. Unfortunately, since this technique only compiles with the newest compilers, we could not test it with a practical calculation on parallel computers available to us. Another techniques for the tuning can be applied to reduce the overheads imposed by the use of object-oriented approach. These are mentioned in section 4.2.

Having the apparent tasks for improving the performance, we expect that a considerable speedup will be achieved once these tunings are applied. We do not have an intention to compare the performance with a manually labored parallelized program. The priority lies in the use of this approach, and we think the performance must be pursued within the use of the framework.

The comparison presented at section 4.2 revealed that the performance of the code written by the framework is already within double of the performance of a manually elaborated code. Therefore, additional tunings would bring the performance much closer to the level where the advantages of this approach compensate the inferior performance. Then, the merits offered by the framework would be welcomed when they are compared to the efforts that must be paid for obtaining a superior performance.

5.4 System Expansion

Though the framework prepares the classes only for structured data system at the moment, the same approach and concept can be applied to the other data models. When considering the use for CFD, unstructured data is expected next to structured data. Though the detail of the implementation would be different, the structure of the program can be abstracted in the same manner. What particularly should be prepared for the use of unstructured data system is a proper domain decomposition method. While there are not many choices with structured data, various methods can be devised for unstructured one. A proper field module and its access module also should be devised for unstructured data system.

Similar architecture can be observed among many scientific computation programs. Therefore, it is possible to apply the same approach to designing a framework for the application other than CFD. If frameworks are established for the calculations of multiple different physics, it will bring more benefits than that supplied by the use of each individual framework. One of the benefits is a support for a coupled calculation of multiple physics. For a loosely coupled simulation, the calculation proceeds by exchanging the results of each calculation at appropriate intervals [1, 2]. Therefore, it is required to prepare the interface procedure that interprets the data structure of one system, so that the other calculation can deal with it. In addition, a coupled calculation needs a global load balance for parallel computation. These requirements lead to build a system specifically designed for a particular calculation. This makes a development for a coupled calculation difficult, and the program would become an inflexible one.

With the frameworks, the program for each physics can be developed independently within each framework, and functions required for a coupled simulation will be added as extension modules. An arrangement for a loosely coupled simulation of CFD and CSD (Computational Structure Dynamics) is depicted in Fig. 5.1. There are frameworks for each calculation, and each has its own data module (denoted as CfdField and CsdField). As both frameworks have ”FieldUser” to access their own field modules, an interface module can be prepared easily as the module that inherits both Users. This example shows that a coupled calculation is achieved by preparing additional classes that reside outside of both systems, and the calculation algorithm for each physics can be developed independently.

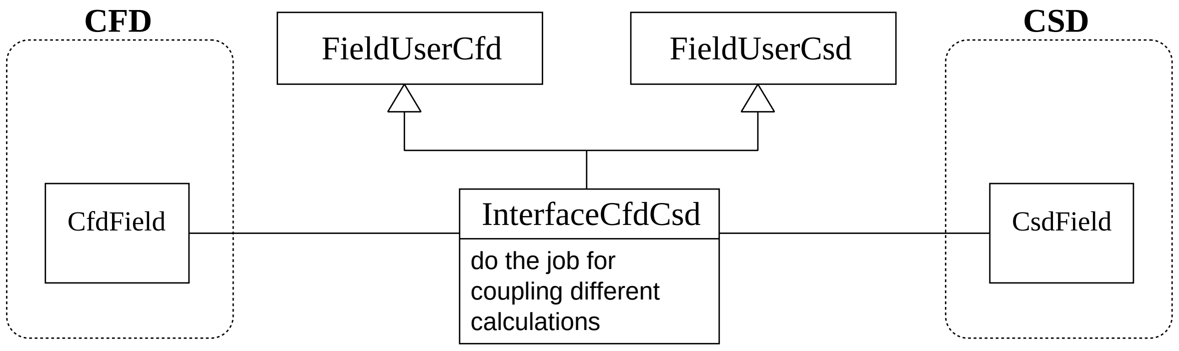


Figure 5.1: Coupled simulation system with the framework

It is convenient to have an integrated system that covers the entire process of CFD by one system. Generally, such system is made as one large system, by gathering all the functions and preparing the interfaces for all of these. That makes the system very large and complicated one. Therefore, it is difficult to develop such a system. The modularized procedures provided by the framework and dynamic generation of these modules will make it easier to build such integrated system. A conceptual design of integrated system is proposed as Fig. 5.2. The cylinders represent a kind of a database for storing objects. The objects will be migrated from these databases to the integrated system placed in the center, at request. The system dynamically changes its function and the user interface according to the object that is loaded as the process's target. For example, when a migrated object is FieldGenerator, the system dynamically generates the modules to function as grid generator. If the object is ”Field” object, a result of a calculation, the system will change itself as visualizer. We think the framework will help the construction of such system. For that, the framework is expected to provide the arrangements for pre- and post-processes. The support to this level is necessary for the reorganization of the entire analyzing process to meet the new paradigm brought by parallel computing.

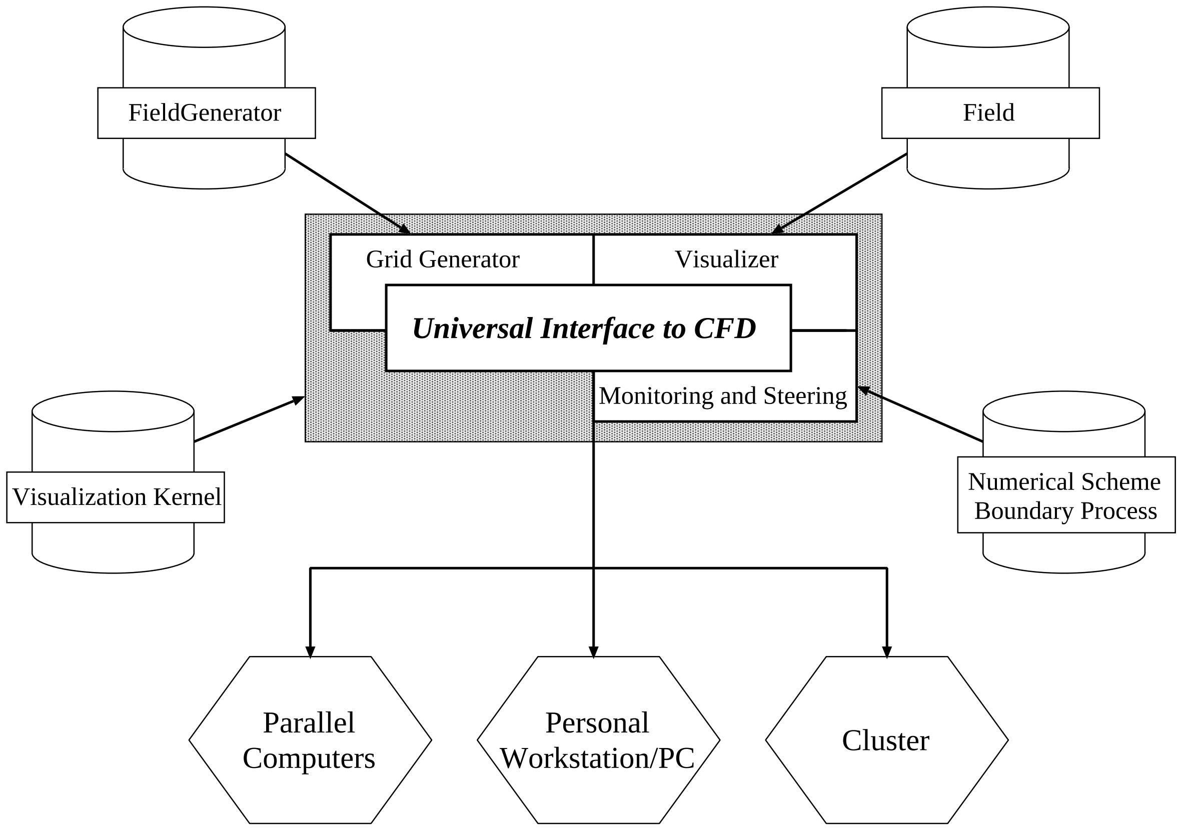


Figure 5.2: A conceptual design of integrated system

Chapter 6

CONCLUDING REMARK

We proposed a programming paradigm for parallel computation of CFD, and designed the framework that supports the programming compliant to the concept. This work is done for aiming the following achievements.

- to present a guiding paradigm
- to design a framework that provides a common infrastructure
- to achieve the separation of the procedures for main calculation and parallelization
- to assure portability of the programs for main calculation
- to absorb the special treatments for a particular environment within the same approach
- to allow addition of external features without affecting the existing modules
- to offer the framework for the pre- and post-processing

We built a framework, examined its utilization, and proved the validity of the framework regarding the above objectives. In this thesis, the arrangements for these features are explained and proved. The advantages offered by the framework are yielded by our programming paradigm. The main key of the approach is to find a proper abstraction of the algorithm and the data structure, so that they have same architecture regardless whether they are used for parallel or sequential computation.

The most of the advantages provided by the framework contribute in the ways of programming. The separation of the parallelization procedures benefits greatly in this aspect. It also enhances the portability of the programs; one same program can be used for both parallel and sequential calculations. Besides, the parallelization procedures are realized as reusable modules; this means the labors for parallelization are now reusable, and there is no need to parallelize every different program. These benefits considerably reduce the burden of parallel programming.

Another advantage of the framework is provided by the selection of the classes. Some classes correspond directly to the subjects appeared in typical CFD code, such as grid, flow variables, scheme and boundary conditions. These classes are prepared as abstract classes, and they act as the template for programming. Such selection of the classes makes the programming easier, because users can program by handling the familiar concepts directly. Preparing such classes also becomes an advantage when the framework acts as infrastructure. These advantages suggest the merits that would be achieved once a standard framework were established.

Other than the benefits to the programming tasks mentioned above, the important aspect of the framework is the programming paradigm. It does not recommend the parallelization of existing programs. It supports developing a fundamentally parallel program, by preparing a structure that suits to build the program. The attitude taken here is to insist designing a parallel algorithm, instead of parallelizing the algorithm that is originally designed for sequential calculation. There is no assurance on the validity of the current parallelization approach when considering the large volume of data and the complicatedness of the programs that will be handled by parallel computation in future. We think altering the programming paradigm is highly desired for making a computing process much more suitable to the new philosophy of parallel computing, in order to gain the benefits fully from it. It cannot be achieved merely by altering the programming of the calculation code. The recomposition of the total process is necessary for coping with a large volume and distributed data that is produced by parallel computation. Therefore, this framework intentionally includes the arrangements for pre- and post-processes.

We examined how the calculation process could be practiced with the approach like ours, and argued the importance and the advantages of establishing a methodology for CFD research.

Bibliography

[1] R. Onishi, T. Ohta, and T. Kimura. A conceptual design of multidisciplinary-integrated CFD simulation on parallel computers. JAERI-DATA/Code 96-031, Japan Atomic Energy Research Institute, 1996. (in Japanese).

[2] R. Onishi, T. Kimura, T. Ohta, and Z. Guo. Development of parallel computing environment for aircraft aero-structural coupled analysis. In D.R.Emerson et al., editors, *Parallel Computational Fluid Dynamics*, pages 667–673. Parallel CFD ’97, Elsevier, 1998.

[3] Al Geist et al. *PVM*. The MIT Press, 1994.

[4] William Gropp et al. *Using MPI*. The MIT Press, 1994.

[5] Takashi Ohta. An object-oriented programming paradigm for parallel computational fluid dynamics on memory distributed parallel computers. In D.R.Emerson et al., editors, *Parallel Computational Fluid Dynamics*, pages 561–568. Parallel CFD ’97, Elsevier, 1998.

[6] Takashi Ohta. Design of a data class for parallel scientific computing. In Yutaka Ishikawa et al., editors, *Scientific Computing in Object-Oriented Parallel Environments*, pages 211–217. ISCOPE ’97, Springer, December 1997.

[7] Takashi Ohta. An object-oriented programming paradigm for parallelization of computational fluid dynamics. JAERI-DATA/Code 97-012, Japan Atomic Energy Research Institute, 1997. (in Japanese).

[8] Takashi Ohta. An object-oriented design for parallel computational fluid dynamics. In *Special Publication of National Aerospace Laboratory*, volume SP-37, pages 307–312. National Aerospace Laboratory, 1998. (in Japanese).

[9] D. R. Emerson et al., editors. *Parallel Computational Fluid Dynamics, Recent Development and Advances Using Parallel Computers*. Parallel CFD '97, North Holland, 1997.

[10] Horst D. Simon, editor. *Parallel Computational Fluid Dynamics, Implemnetation and Results*. MIT Press, 1992.

[11] Takashi Nakamura et al. Simulation of the 3 dimensional cascade flow with numerical wind tunnel(NWT). In *Proc. IEEE Supercomuting '96*, 1996.

[12] A. Stoessel, M. Baum, and T. J. Poinsot. Towards 3D direct numerical simulation of turbulent reactive flow. In N. Satofuka et al., editors, *Parallel Computational Fluid Dynamics*, pages 85–92. Parallel CFD '94, North-Holland, 1995.

[13] K. Morgan, N. P. Weatherill, O. Hassan, and M. T. Manzari. Parallel processing for large scale aerospace engineering. In D. R. Emerson et al., editors, *Parallel Computational Fluid Dynamics*, pages 15–22. Parallel CFD '97, North Holland, 1998.

[14] Kengo Nakajima, Hisashi Nakamura, and Takahiko Tanahashi. Parallel iterative solvers with localized ILU preconditioning. In D. R. Emerson et al., editors, *Parallel Computational Fluid Dynamics*, pages 359–366. Parallel CFD '97, North-Holland, 1998.

[15] S. Hammond and T. Barth. On a masssively parallel Euler solver for unstructured grids. In Horst D. Simon, editor, *Parallel Computational Fluid Dynamics*, chapter 3, pages 55–68. The MIT Press, 1992.

[16] Rainald Löhner. Parallel unstructured grid generation. *Computer Methods in Applied Mechanics and Engineering*, pages 343–357, 1992.

[17] Horst D. Simon. Partitioning of unstructured problems for parallel processing. Technical Report RNR-91-08, NASA Ames Research Center, 1991.

[18] Gregory V. Wilson and Paul Lu, editors. *Parallel Programming Using C++*. MIT Press, 1996.

[19] Yutaka Ishikawa et al., editors. *Scientific Computing in Object-Oriented Parallel Environments*. ISCOPE '97, Springer, 1997.

[20] Jean-Pierre Briot et al., editors. *Object-Based Parallel and Distributed Computatioin*. OBPDC '95, Springer, 1996.

[21] Carl Kesselman. CC++. In *Parallel Programming Using C++*, chapter 3, pages 91–130. The MIT Press, 1996.

[22] Denis Caromel, Fabrice Belloncle, and Yves Roudier. C++//. In *Parallel Programming Using C++*, chapter 7, pages 257–296. The MIT Press, 1996.

[23] W. G. O'Farrell et al. ABC++. In *Parallel Programming Using C++*, chapter 1, pages 1–42. The MIT Press, 1996.

[24] Anthony Skjellum et al. MPI++. In *Parallel Programming Using C++*, chapter 12, pages 465–506. The MIT Press, 1996.

[25] J. V. W. Reynders et al. POOMA. In *Parallel Programming Using C++*, chapter 14, pages 547–587. The MIT Press, 1996.

[26] William Humphrey et al. Optimization of data-parallel field expressions in the POOMA framework. In Yutaka Ishikawa et al., editors, *Scientific Computing in Object-Oriented Parallel Environments*, pages 185–194. ISCOPE '97, Springer, 1997.

[27] Martin Flowler. *UML Distilled*. Addison-Wesley, 1997.

[28] Hans-Erik Eriksson and Magnus Penker. *UML Toolkit*. John Wiley & Sons, 1998.

[29] Jackie Neider et al. *OpenGL Programming Guide*. Addison-Wesley, 1993.

[30] Charles H. Koelbel et al. *The High Performance Fortran Handbook*. The MIT Press, 1994.

[31] T. L. Veldhuizen and M. E. Jernigan. Will C++ be faster than Fortran? In Yutaka Ishikawa et al., editors, *Scientific Computing in Object-Oriented Parallel Environments*, pages 49–56. ISCOPE '97, Springer, 1997.

[32] Todd L. Veldhuizen. Scientific computing: C++ versus Fortran: C++ has more than caught up. *Dr. Dobb's Journal of Software Tools*, 22(11):34, 36–38, 91, November 1997.

Acknowledgement

The author would like to thank Professor Hirotoshi Kubota of department of Aeronautics and Astronautics, the University of Tokyo, for his support and advices in completing the thesis. I would like to express my gratitude to Professor Susumu Shirayama, of department of Environmental and Ocean Engineering, the University of Tokyo, for his valuable advices and discussions on the research subject. I also would like to thank professor Kunio Kuwahara of the Institute of Space and Astronautical Science, for guiding me to the research on CFD. The author thank professor Toshio Kawai, the Dean of department of Photonics Science and Technology, Chitose Institute of Science and Technology, who is formerly Professor of Keio University, for leading me to computational science and parallel computing. Lastly, I would like to thank to the Center for Computational Science and Engineering, Japan Atomic Energy Research Institute, for allowing me to pursue this research. The most of calculation results appeared in the thesis were carried out by the computer facility there.

Appendix A

Comparison of Programming Efforts

Here, two parallel programs are listed to compare the programming efforts needed in programming with and without the aide of the framework. One is the coding needed to develop a parallel code when the rest part is provided by the framework. The other is the parallel code written by using a message passing library but without the aid of the framework. The latter is not a complete code but omitted some non-important subroutines. However, it is enough to show the amount of programming needed in that approach.

A Scheme Class for Use of the Framework

```
#include "schemestr3d.h"
#include "incmetrics3d.h"
#include "strvars3d.h"
#include <cmath>
using namespace std;

template <class Variables>
class IncmpUpwind3d : public SchemeStr3d<Variables> {
 public:
  typedef IncMetrics3d<Type,Allocator> Metrics;

  void solvePoisson() {
```

```
residu_s = 0.0;

double c0,cip,cim,cjp,cjm,ckp,ckm;
double dudx,dudy,dudz;
double dvdx,dvdy,dvdz;
double dwdx,dwdy,dwdz;
int ip,im,jp,jm,kp,km;

double g1x,g2x,g3x;
double g1y,g2y,g3y;
double g1z,g2z,g3z;

for (int i=grid->begin1()+1; i<=grid->end1()-1; i++)
  for (int j=grid->begin2()+1; j<=grid->end2()-1; j++)
    for (int k=grid->begin3()+1; k<=grid->end3()-1; k++) {
      ip = i + 1, im = i - 1;
      jp = j + 1, jm = j - 1;
      kp = k + 1, km = k - 1;

      double a1 = metric->a1(i,j,k);
      double a2 = metric->a2(i,j,k);
      double a3 = metric->a3(i,j,k);
      double b1 = metric->b1(i,j,k);
      double b2 = metric->b2(i,j,k);
      double b3 = metric->b3(i,j,k);
      double c1 = metric->c1(i,j,k);
      double c2 = metric->c2(i,j,k);
      double c3 = metric->c3(i,j,k);

      c0 = -2.0*(a1 + a2 + a3);
      cip = a1 + 0.5*b1;
      cim = a1 - 0.5*b1;
      cjp = a2 + 0.5*b2;
      cjm = a2 - 0.5*b2;
      ckp = a3 + 0.5*b3;
      ckm = a3 - 0.5*b3;

      g1x = metric->m11(i,j,k);
      g1y = metric->m12(i,j,k);
      g1z = metric->m13(i,j,k);
      g2x = metric->m21(i,j,k);
      g2y = metric->m22(i,j,k);
      g2z = metric->m23(i,j,k);
      g3x = metric->m31(i,j,k);
      g3y = metric->m32(i,j,k);
      g3z = metric->m33(i,j,k);
```

```
        double udif1 = vars->at(ip,j,k).u() - vars->at(im,j,k).u();
        double udif2 = vars->at(i,jp,k).u() - vars->at(i,jm,k).u();
        double udif3 = vars->at(i,j,kp).u() - vars->at(i,j,km).u();
        double vdif1 = vars->at(ip,j,k).v() - vars->at(im,j,k).v();
        double vdif2 = vars->at(i,jp,k).v() - vars->at(i,jm,k).v();
        double vdif3 = vars->at(i,j,kp).v() - vars->at(i,j,km).v();
        double wdif1 = vars->at(ip,j,k).w() - vars->at(im,j,k).w();
        double wdif2 = vars->at(i,jp,k).w() - vars->at(i,jm,k).w();
        double wdif3 = vars->at(i,j,kp).w() - vars->at(i,j,km).w();

        dudx = (g1x*udif1 + g2x*udif2 + g3x*udif3)*0.5;
        dudy = (g1y*udif1 + g2y*udif2 + g3y*udif3)*0.5;
        dudz = (g1z*udif1 + g2z*udif2 + g3z*udif3)*0.5;
        dvdx = (g1x*vdif1 + g2x*vdif2 + g3x*vdif3)*0.5;
        dvdy = (g1y*vdif1 + g2y*vdif2 + g3y*vdif3)*0.5;
        dvdz = (g1z*vdif1 + g2z*vdif2 + g3z*vdif3)*0.5;
        dwdx = (g1x*wdif1 + g2x*wdif2 + g3x*wdif3)*0.5;
        dwdy = (g1y*wdif1 + g2y*wdif2 + g3y*wdif3)*0.5;
        dwdz = (g1z*wdif1 + g2z*wdif2 + g3z*wdif3)*0.5;

        double src1 = -(dudx*dudx + dvdy*dvdy + dwdz*dwdz)
                    - 2.0*(dudy*dvdx + dudz*dwdx + dvdz*dwdy)
                    + 1.0/dt*(dudx + dvdy + dwdz);

        double src2 = cip*vars->at(ip,j,k).p() + cim*vars->at(im,j,k).p()
                    + cjp*vars->at(i,jp,k).p() + cjm*vars->at(i,jm,k).p()
                    + ckp*vars->at(i,j,kp).p() + ckm*vars->at(i,j,km).p()
                    + c1*(vars->at(ip,jp,k).p() - vars->at(ip,jm,k).p()
                      - vars->at(im,jp,k).p() + vars->at(im,jm,k).p())*0.25
                    + c2*(vars->at(i,jp,kp).p() - vars->at(i,jp,km).p()
                      - vars->at(i,jm,kp).p() + vars->at(i,jm,km).p())*0.25
                    + c3*(vars->at(ip,j,kp).p() - vars->at(im,j,kp).p()
                      - vars->at(ip,j,km).p() + vars->at(im,j,km).p())*0.25;

        double pprev = vars->at(i,j,k).p();
        double incr  = (src1 - src2)/c0 - pprev;

        vars->get(i,j,k)->p() = pprev + omega_s*incr;
        residu_s += incr*incr;
      }
}

void preprocNS() {
  for (int i=vars->begin1(); i<=vars->end1(); i++)
    for (int j=vars->begin2(); j<=vars->end2(); j++)
```

```
      for (int k=vars->begin3(); k<=vars->end3(); k++) {
        wrk1->at(i,j,k).u() = vars->at(i,j,k).u();
        wrk1->at(i,j,k).v() = vars->at(i,j,k).v();
        wrk1->at(i,j,k).w() = vars->at(i,j,k).w();
        wrk1->at(i,j,k).p() = vars->at(i,j,k).p();
      }
}

void solveNS() {
  residu_n = 0.0;
  double Inv4  = 1.0/4.0;
  double Inv12 = 1.0/12.0;

  int ip1,im1,jp1,jm1,kp1,km1;
  int ip2,im2,jp2,jm2,kp2,km2;
  double c0,cip,cim,cjp,cjm,ckp,ckm;
  double g1x,g2x,g3x;
  double g1y,g2y,g3y;
  double g1z,g2z,g3z;

  for (int i=grid->begin1()+1; i<=grid->end1()-1; i++)
    for (int j=grid->begin2()+1; j<=grid->end2()-1; j++)
      for (int k=grid->begin3()+1; k<=grid->end3()-1; k++) {
        ip1 = i + 1, im1 = i - 1, ip2 = i + 2, im2 = i - 2;
        jp1 = j + 1, jm1 = j - 1, jp2 = j + 2, jm2 = j - 2;
        kp1 = k + 1, km1 = k - 1, kp2 = k + 2, km2 = k - 2;

        double a1 = metric->a1(i,j,k);
        double a2 = metric->a2(i,j,k);
        double a3 = metric->a3(i,j,k);
        double b1 = metric->b1(i,j,k);
        double b2 = metric->b2(i,j,k);
        double b3 = metric->b3(i,j,k);
        double c1 = metric->c1(i,j,k);
        double c2 = metric->c2(i,j,k);
        double c3 = metric->c3(i,j,k);

        c0 = -2.0*(a1 + a2 + a3);
        cip = a1 + 0.5*b1;
        cim = a1 - 0.5*b1;
        cjp = a2 + 0.5*b2;
        cjm = a2 - 0.5*b2;
        ckp = a3 + 0.5*b3;
        ckm = a3 - 0.5*b3;

        g1x = metric->m11(i,j,k);
```

```
g1y = metric->m12(i,j,k);
g1z = metric->m13(i,j,k);
g2x = metric->m21(i,j,k);
g2y = metric->m22(i,j,k);
g2z = metric->m23(i,j,k);
g3x = metric->m31(i,j,k);
g3y = metric->m32(i,j,k);
g3z = metric->m33(i,j,k);

double uc = g1x*wrk1->at(i,j,k).u()
        + g1y*wrk1->at(i,j,k).v()
        + g1z*wrk1->at(i,j,k).w();
double vc = g2x*wrk1->at(i,j,k).u()
        + g2y*wrk1->at(i,j,k).v()
        + g2z*wrk1->at(i,j,k).w();
double wc = g3x*wrk1->at(i,j,k).u()
        + g3y*wrk1->at(i,j,k).v()
        + g3z*wrk1->at(i,j,k).w();

double dn
  = 1.0 - dt*(-(fabs(uc) + fabs(vc) + fabs(wc))*6.0*Inv4 + c0/Re);

double nlterm_u1
  = uc*(vars->at(im2,j,k).u() - vars->at(ip2,j,k).u()
  + 8.0*(vars->at(ip1,j,k).u() - vars->at(im1,j,k).u()))*Inv12
  + fabs(uc)*(vars->at(ip2,j,k).u() + vars->at(im2,j,k).u()
  - 4.0*(vars->at(ip1,j,k).u() + vars->at(im1,j,k).u()))*Inv4;
double nlterm_u2
  = vc*(vars->at(i,jm2,k).u() - vars->at(i,jp2,k).u()
  + 8.0*(vars->at(i,jp1,k).u() - vars->at(i,jm1,k).u()))*Inv12
  + fabs(vc)*(vars->at(i,jp2,k).u() + vars->at(i,jm2,k).u()
  - 4.0*(vars->at(i,jp1,k).u() + vars->at(i,jm1,k).u()))*Inv4;
double nlterm_u3
  = wc*(vars->at(i,j,km2).u() - vars->at(i,j,kp2).u()
  + 8.0*(vars->at(i,j,kp1).u() - vars->at(i,j,km1).u()))*Inv12
  + fabs(wc)*(vars->at(i,j,kp2).u() + vars->at(i,j,km2).u()
  - 4.0*(vars->at(i,j,kp1).u() + vars->at(i,j,km1).u()))*Inv4;

double nlterm_v1
  = uc*(vars->at(im2,j,k).v() - vars->at(ip2,j,k).v()
  + 8.0*(vars->at(ip1,j,k).v() - vars->at(im1,j,k).v()))*Inv12
  + fabs(uc)*(vars->at(ip2,j,k).v() + vars->at(im2,j,k).v()
  - 4.0*(vars->at(ip1,j,k).v() + vars->at(im1,j,k).v()))*Inv4;
double nlterm_v2
  = vc*(vars->at(i,jm2,k).v() - vars->at(i,jp2,k).v()
  + 8.0*(vars->at(i,jp1,k).v() - vars->at(i,jm1,k).v()))*Inv12
  + fabs(vc)*(vars->at(i,jp2,k).v() + vars->at(i,jm2,k).v()
  - 4.0*(vars->at(i,jp1,k).v() + vars->at(i,jm1,k).v()))*Inv4;
double nlterm_v3
  = wc*(vars->at(i,j,km2).v() - vars->at(i,j,kp2).v()
  + 8.0*(vars->at(i,j,kp1).v() - vars->at(i,j,km1).v()))*Inv12
  + fabs(wc)*(vars->at(i,j,kp2).v() + vars->at(i,j,km2).v()
  - 4.0*(vars->at(i,j,kp1).v() + vars->at(i,j,km1).v()))*Inv4;

double nlterm_w1
  = uc*(vars->at(im2,j,k).w() - vars->at(ip2,j,k).w()
  + 8.0*(vars->at(ip1,j,k).w() - vars->at(im1,j,k).w()))*Inv12
  + fabs(uc)*(vars->at(ip2,j,k).w() + vars->at(im2,j,k).w()
  - 4.0*(vars->at(ip1,j,k).w() + vars->at(im1,j,k).w()))*Inv4;
double nlterm_w2
  = vc*(vars->at(i,jm2,k).w() - vars->at(i,jp2,k).w()
  + 8.0*(vars->at(i,jp1,k).w() - vars->at(i,jm1,k).w()))*Inv12
  + fabs(vc)*(vars->at(i,jp2,k).w() + vars->at(i,jm2,k).w()
  - 4.0*(vars->at(i,jp1,k).w() + vars->at(i,jm1,k).w()))*Inv4;
double nlterm_w3
  = wc*(vars->at(i,j,km2).w() - vars->at(i,j,kp2).w()
  + 8.0*(vars->at(i,j,kp1).w() - vars->at(i,j,km1).w()))*Inv12
  + fabs(wc)*(vars->at(i,j,kp2).w() + vars->at(i,j,km2).w()
  - 4.0*(vars->at(i,j,kp1).w() + vars->at(i,j,km1).w()))*Inv4;

double pu = (g1x*(vars->at(ip1,j,k).p() - vars->at(im1,j,k).p())
          + g2x*(vars->at(i,jp1,k).p() - vars->at(i,jm1,k).p())
          + g3x*(vars->at(i,j,kp1).p() - vars->at(i,j,km1).p()))*0.5;
double pv = (g1y*(vars->at(ip1,j,k).p() - vars->at(im1,j,k).p())
          + g2y*(vars->at(i,jp1,k).p() - vars->at(i,jm1,k).p())
          + g3y*(vars->at(i,j,kp1).p() - vars->at(i,j,km1).p()))*0.5;
double pw = (g1z*(vars->at(ip1,j,k).p() - vars->at(im1,j,k).p())
          + g2z*(vars->at(i,jp1,k).p() - vars->at(i,jm1,k).p())
          + g3z*(vars->at(i,j,kp1).p() - vars->at(i,j,km1).p()))*0.5;

double diff_u
  = cip*vars->at(ip1,j,k).u() + cim*vars->at(im1,j,k).u()
  + cjp*vars->at(i,jp1,k).u() + cjm*vars->at(i,jm1,k).u()
  + ckp*vars->at(i,j,kp1).u() + ckm*vars->at(i,j,km1).u()
  + c1*(vars->at(ip1,jp1,k).u() - vars->at(ip1,jm1,k).u()
      - vars->at(im1,jp1,k).u() + vars->at(im1,jm1,k).u())*Inv4
  + c2*(vars->at(i,jp1,kp1).u() - vars->at(i,jp1,km1).u()
      - vars->at(i,jm1,kp1).u() + vars->at(i,jm1,km1).u())*Inv4
  + c3*(vars->at(ip1,j,kp1).u() - vars->at(im1,j,kp1).u()
      - vars->at(ip1,j,km1).u() + vars->at(im1,j,km1).u())*Inv4;

double diff_v
```

```
        = cip*vars->at(ip1,j,k).v() + cim*vars->at(im1,j,k).v()
        + cjp*vars->at(i,jp1,k).v() + cjm*vars->at(i,jm1,k).v()
        + ckp*vars->at(i,j,kp1).v() + ckm*vars->at(i,j,km1).v()
        + c1*(vars->at(ip1,jp1,k).v() - vars->at(ip1,jm1,k).v()
          - vars->at(im1,jp1,k).v() + vars->at(im1,jm1,k).v())*Inv4
        + c2*(vars->at(i,jp1,kp1).v() - vars->at(i,jp1,km1).v()
          - vars->at(i,jm1,kp1).v() + vars->at(i,jm1,km1).v())*Inv4
        + c3*(vars->at(ip1,j,kp1).v() - vars->at(im1,j,kp1).v()
          - vars->at(ip1,j,km1).v() + vars->at(im1,j,km1).v())*Inv4;

      double diff_w
        = cip*vars->at(ip1,j,k).w() + cim*vars->at(im1,j,k).w()
        + cjp*vars->at(i,jp1,k).w() + cjm*vars->at(i,jm1,k).w()
        + ckp*vars->at(i,j,kp1).w() + ckm*vars->at(i,j,km1).w()
        + c1*(vars->at(ip1,jp1,k).w() - vars->at(ip1,jm1,k).w()
          - vars->at(im1,jp1,k).w() + vars->at(im1,jm1,k).w())*Inv4
        + c2*(vars->at(i,jp1,kp1).w() - vars->at(i,jp1,km1).w()
          - vars->at(i,jm1,kp1).w() + vars->at(i,jm1,km1).w())*Inv4
        + c3*(vars->at(ip1,j,kp1).w() - vars->at(im1,j,kp1).w()
          - vars->at(ip1,j,km1).w() + vars->at(im1,j,km1).w())*Inv4;

      double uchange
        = -(nlterm_u1 + nlterm_u2 + nlterm_u3) - pu + 1.0/Re*diff_u;
      double vchange
        = -(nlterm_v1 + nlterm_v2 + nlterm_v3) - pv + 1.0/Re*diff_v;
      double wchange
        = -(nlterm_w1 + nlterm_w2 + nlterm_w3) - pw + 1.0/Re*diff_w;

      double urelax
        = (wrk1->at(i,j,k).u() + dt*uchange)/dn - vars->at(i,j,k).u();
      double vrelax
        = (wrk1->at(i,j,k).v() + dt*vchange)/dn - vars->at(i,j,k).v();
      double wrelax
        = (wrk1->at(i,j,k).w() + dt*wchange)/dn - vars->at(i,j,k).w();

      wrk2->at(i,j,k).u() = vars->at(i,j,k).u() + omega_n*urelax;
      wrk2->at(i,j,k).v() = vars->at(i,j,k).v() + omega_n*vrelax;
      wrk2->at(i,j,k).w() = vars->at(i,j,k).w() + omega_n*wrelax;

      residu_n += urelax*urelax + vrelax*vrelax + wrelax*wrelax;
    }

for (int i=grid->begin1()+1; i<=grid->end1()-1; i++)
  for (int j=grid->begin2()+1; j<=grid->end2()-1; j++)
    for (int k=grid->begin3()+1; k<=grid->end3()-1; k++) {
      vars->at(i,j,k).u() = wrk2->at(i,j,k).u();
```

```
        vars->at(i,j,k).v() = wrk2->at(i,j,k).v();
        vars->at(i,j,k).w() = wrk2->at(i,j,k).w();
      }
  }

 protected:
  Metrics    *metric;

  float     Re;
  float     dt;

  float     omega_s;
  float     omega_n;
  float     residu_s;
  float     residu_n;
  float     threshold_s;
  float     threshold_n;

  StrVars3d<Variables> *wrk1;
  StrVars3d<Variables> *wrk2;

};
```

Parallel Code without the Framework

```
/*
 *   incNS3D02   : <scheme>
 *                 Incompressible flow
 *                 Navier-Stokes equations
 *                 Three-dimensional
 *                 Finite Difference Method
 *                 Generalized coordinates system (structured, generalized but time)
 *                 MAC method
 *                 SOR for poisson solver
 *                 euler implicit for time integration
 *                 mono-region grid system
 *               <parallel>
 *                 SPMD
 *                 PVM3.3
 */

#include <stdio.h>
```

```
#include <stddef.h>
#include <stdlib.h>
#include <math.h>
#include "pvm3.h"
#include "nrutil.h"
#include "func_inc3dMP.h"

void main( int argc, char *argv[] )
{
  PVM_param index;
  Files files;
  CFD_param param;
  int errcode;

  index.mytid = pvm_mytid(); /* enroll in pvm */
  index.mygid = pvm_joingroup(Group); /* join a group */
  index.myparent = pvm_parent();

  decode_input_parameters(argc, argv, &files, &param, &index);
  if(index.myparent == PvmNoParent)
    if(index.myparent == PvmNoParent) { /* then I'm master */
      index.tids[0] = index.mytid;

      if(index.nprocess > 1 ) {
        errcode = pvm_spawn(SlaveName, argv, PvmTaskDefault, "",
                            index.nprocess-1, &index.tids[1]);
        if(errcode != index.nprocess-1) {
          printf("!!! slave tasks haven't spawned succesfully.");
          printf("  spawend tasks : %d\n",errcode);
          exit(1);
        }
      }
    }

  pvm_barrier(Group, index.nprocess);

  if(index.myparent == PvmNoParent)
    master_task(index, param, files);
  else
    slave_task(index, param);

  pvm_barrier(Group, index.nprocess);
  pvm_lvgroup(Group); /* leave the group */
  pvm_exit(); /* exit pvm */
}
```

```
void master_task( PVM_param index, CFD_param param, Files files )
{
  LoadDiv load;
  TaskPosit where;

  /* for CFD scheme */
  Grid gco, co;
  BCondition gbc, bc;
  FlowField gff, ff;
  Metrics met;
  Laplacian lap;
  double fdirect[3];
  WorkField wf1, wf2;
  int lni, lnj, lnk;
  double time, residual;
  int zero = 0;

  send_taskid(index);
  set_load_parameters(index, &load);

  /* task specific to the master */
  read_gridsystem_data(files.griddata, &gco, &gbc);
  allocate_flowfield(&gff, gco.ni, gco.nj, gco.nk);
  if(files.frstate == Done)
    read_succession_file(files.insuccess, &gff);
  else
    condition_initial_flowfield(&gff);
  condition_bound_inflow(fdirect);
  condition_bound_pressure(&gff, gbc);
  condition_bound_velocity(&gff, gbc, fdirect);

  make_loadbalance_index(gco, index, &load);
  deliver_data_to_slave(index, load, gco, gff, gbc);
  determine_my_position(index, load, &where);

  lni = load.iend[zero] - load.istart[zero] + 1;
  lnj = load.jend[zero] - load.jstart[zero] + 1;
  lnk = gco.nk;
  allocate_geomfield(&co, &bc, lni, lnj, lnk);
  allocate_flowfield(&ff, lni, lnj, lnk);
  allocate_structures(&met, &lap, &wf1, &wf2, lni, lnj, lnk);
  deliver_data_to_master(load, gco, gff, gbc, &co, &ff, &bc);

  /* task common to the master and the slaves */
  calculate_metrics_tensor(co, &met);
  calculate_laplacian_coeffs(met, &lap);
  for(time = 0.0; time < param.maxtime; time += param.dt) {
    residual = 0.0;
    residual = poisson_solve_sor_MP(param, met, lap, bc, &ff, index, where);
    residual = ns_solve_eulerimplicit_MP(param, met, lap, bc, fdirect, &ff,
                                     wf1, wf2, index, where);
  }

  gather_data_from_slave(index, load, &gff);
  restore_data_to_master(load, &gff, ff);

  if(files.fdstate == Done)
    write_result_DXgeneral(files.resultdata, gco, gff, param );
  if(files.fwstate == Done)
    write_succession_file(files.outsuccess, gff);

}

void slave_task( PVM_param index, CFD_param param )
{
  LoadDiv load;
  TaskPosit where;

  /* for CFD scheme */
  Grid co;
  BCondition bc;
  FlowField ff;
  Metrics met;
  Laplacian lap;
  double fdirect[3];
  WorkField wf1, wf2;
  int lni, lnj, lnk;
  double residual, time;

  recv_taskid(&index);
  set_load_parameters(index, &load);
  condition_bound_inflow(fdirect);

  /* task specific to the slave */
  receive_data_from_master(index, &co, &ff, &bc);
  determine_my_position(index, load, &where);

  lni = co.ni; lnj = co.nj; lnk = co.nk;
  allocate_structures(&met, &lap, &wf1, &wf2, lni, lnj, lnk);

  /* task common to the master and the slaves */
  calculate_metrics_tensor(co, &met);
  calculate_laplacian_coeffs(met, &lap);

  for(time = 0.0; time < param.maxtime; time += param.dt) {
    residual = 0.0;
    residual = poisson_solve_sor_MP(param, met, lap, bc, &ff, index, where);
    residual = ns_solve_eulerimplicit_MP(param, met, lap, bc, fdirect, &ff,
                                     wf1, wf2, index, where);
  }

  send_data_to_master(index, ff);
}

double poisson_solve_sor_MP( CFD_param param, Metrics met, Laplacian lap,
                             BCondition bc, FlowField* ff,
                             PVM_param index, TaskPosit where )
{
  double residual;
  double eps, omega, dt;
  double incr, source1, source2;
  double co0, cip, cim, cjp, cjm, ckp, ckm;
  double dudx, dudy, dudz;
  double dvdx, dvdy, dvdz;
  double dwdx, dwdy, dwdz;
  int n, i, j, k;
  int ip1, im1, jp1, jm1, kp1, km1;
  int msgtag = 10;
  int oneitem = 1;
  int stride = 1;

  eps = param.eps_p; omega = param.omega_p; dt = param.dt;

  n = 0;
  do {
    residual = 0.0;
    for(i=2; i<met.ni-2; i++)
      for(j=2; j<met.nj-2; j++)
        for(k=2; k<met.nk-2; k++) {
          ip1 = i + 1; jp1 = j + 1; kp1 = k + 1;
          im1 = i - 1; jm1 = j - 1; km1 = k - 1;

          co0 = - 2.0 * (lap.a1[i][j][k] + lap.a2[i][j][k] + lap.a3[i][j][k]);
          cip = lap.a1[i][j][k] + 0.5 * lap.b1[i][j][k];
          cim = lap.a1[i][j][k] - 0.5 * lap.b1[i][j][k];
          cjp = lap.a2[i][j][k] + 0.5 * lap.b2[i][j][k];
          cjm = lap.a2[i][j][k] - 0.5 * lap.b2[i][j][k];
```

```
ckp = lap.a3[i][j][k] + 0.5 * lap.b3[i][j][k];
ckm = lap.a3[i][j][k] - 0.5 * lap.b3[i][j][k];

dudx = (met.g1x1[i][j][k] * (ff->u[ip1][j][k] - ff->u[im1][j][k])
       + met.g2x1[i][j][k] * (ff->u[i][jp1][k] - ff->u[i][jm1][k])
       + met.g3x1[i][j][k] * (ff->u[i][j][kp1] - ff->u[i][j][km1])) * 0.5;
dudy = (met.g1x2[i][j][k] * (ff->u[ip1][j][k] - ff->u[im1][j][k])
       + met.g2x2[i][j][k] * (ff->u[i][jp1][k] - ff->u[i][jm1][k])
       + met.g3x2[i][j][k] * (ff->u[i][j][kp1] - ff->u[i][j][km1])) * 0.5;
dudz = (met.g1x3[i][j][k] * (ff->u[ip1][j][k] - ff->u[im1][j][k])
       + met.g2x3[i][j][k] * (ff->u[i][jp1][k] - ff->u[i][jm1][k])
       + met.g3x3[i][j][k] * (ff->u[i][j][kp1] - ff->u[i][j][km1])) * 0.5;
dvdx = (met.g1x1[i][j][k] * (ff->v[ip1][j][k] - ff->v[im1][j][k])
       + met.g2x1[i][j][k] * (ff->v[i][jp1][k] - ff->v[i][jm1][k])
       + met.g3x1[i][j][k] * (ff->v[i][j][kp1] - ff->v[i][j][km1])) * 0.5;
dvdy = (met.g1x2[i][j][k] * (ff->v[ip1][j][k] - ff->v[im1][j][k])
       + met.g2x2[i][j][k] * (ff->v[i][jp1][k] - ff->v[i][jm1][k])
       + met.g3x2[i][j][k] * (ff->v[i][j][kp1] - ff->v[i][j][km1])) * 0.5;
dvdz = (met.g1x3[i][j][k] * (ff->v[ip1][j][k] - ff->v[im1][j][k])
       + met.g2x3[i][j][k] * (ff->v[i][jp1][k] - ff->v[i][jm1][k])
       + met.g3x3[i][j][k] * (ff->v[i][j][kp1] - ff->v[i][j][km1])) * 0.5;
dwdx = (met.g1x1[i][j][k] * (ff->w[ip1][j][k] - ff->w[im1][j][k])
       + met.g2x1[i][j][k] * (ff->w[i][jp1][k] - ff->w[i][jm1][k])
       + met.g3x1[i][j][k] * (ff->w[i][j][kp1] - ff->w[i][j][km1])) * 0.5;
dwdy = (met.g1x2[i][j][k] * (ff->w[ip1][j][k] - ff->w[im1][j][k])
       + met.g2x2[i][j][k] * (ff->w[i][jp1][k] - ff->w[i][jm1][k])
       + met.g3x2[i][j][k] * (ff->w[i][j][kp1] - ff->w[i][j][km1])) * 0.5;
dwdz = (met.g1x3[i][j][k] * (ff->w[ip1][j][k] - ff->w[im1][j][k])
       + met.g2x3[i][j][k] * (ff->w[i][jp1][k] - ff->w[i][jm1][k])
       + met.g3x3[i][j][k] * (ff->w[i][j][kp1] - ff->w[i][j][km1])) * 0.5;

source1 = - (dudx * dudx + dvdy * dvdy + dwdz * dwdz)
  - 2.0 * (dudy * dvdx + dudz * dwdx + dvdz * dwdy)
  + 1.0/dt * (dudx + dvdy + dwdz);
source2 = cip * ff->p[ip1][j][k] + cim * ff->p[im1][j][k]
  + cjp * ff->p[i][jp1][k] + cjm * ff->p[i][jm1][k]
  + ckp * ff->p[i][j][kp1] + ckm * ff->p[i][j][km1]
  + lap.c1[i][j][k] * (ff->p[ip1][jp1][k] - ff->p[ip1][jm1][k]
                    - ff->p[im1][jp1][k] + ff->p[im1][jm1][k]) * 0.25
  + lap.c2[i][j][k] * (ff->p[i][jp1][kp1] - ff->p[i][jp1][km1]
                    - ff->p[i][jm1][kp1] + ff->p[i][jm1][km1]) * 0.25
  + lap.c3[i][j][k] * (ff->p[ip1][j][kp1] - ff->p[im1][j][kp1]
                    - ff->p[ip1][j][km1] + ff->p[im1][j][km1]) * 0.25;

incr = (source1 - source2)/co0 - ff->p[i][j][k];
residual += incr * incr;
```

```
          ff->p[i][j][k] += omega * incr;
        }

    condition_bound_pressure(ff, bc);
    pass_overlapped_field(Pressure, where, ff);
    ++n;
    residual /= met.ni * met.nj * met.nk;

    if(index.myparent == PvmNoParent) {
      pvm_reduce(PvmSum, &residual, 1, PVM_DOUBLE, msgtag, Group, index.mygid);
      pvm_barrier(Group, index.nprocess);
      residual /= index.nprocess;

      pvm_initsend(PvmDataRaw);
      pvm_pkdouble(&residual, oneitem, stride);
      pvm_bcast(Group, msgtag);
    }
    else {
      pvm_reduce(PvmSum, &residual, 1, PVM_DOUBLE, msgtag, Group, Master);
      pvm_barrier(Group, index.nprocess);

      pvm_recv(index.myparent, msgtag);
      pvm_upkdouble(&residual, oneitem, stride);
    }
  } while((residual > eps) && (n < 100));

  return residual;
}

#define I12     1.0/12.0
#define I4      1.0/4.0

double ns_solve_eulerimplicit_MP( CFD_param param, Metrics met, Laplacian lap,
                                 BCondition bc, double* fdirect, FlowField* ff,
                                 WorkField wf1, WorkField wf2, PVM_param index, TaskPosit where )
{
  double residual;
  double eps, omega, dt, Re;
  double co0, cip, cim, cjp, cjm, ckp, ckm;
  double dudx, dudy, dudz;
  double dvdx, dvdy, dvdz;
  double dwdx, dwdy, dwdz;
  double denom;
  double u, v, w, uc, vc, wc;
  double pu, pv, pw;
  double nlterm_ui, nlterm_vi, nlterm_wi;
  double nlterm_uj, nlterm_vj, nlterm_wj;
  double nlterm_uk, nlterm_vk, nlterm_wk;
  double diff_u, diff_v, diff_w;
  double change_u, change_v, change_w;
  double relax_u, relax_v, relax_w;
  int n, i, j, k;
  int ip1, im1, jp1, jm1, kp1, km1;
  int ip2, im2, jp2, jm2, kp2, km2;
  double g1x1, g1x2, g1x3;
  double g2x1, g2x2, g2x3;
  double g3x1, g3x2, g3x3;
  int msgtag = 11;
  int oneitem = 1;
  int stride = 1;

  eps = param.eps_v; omega = param.omega_v; dt = param.dt; Re = param.Re;

  for(i=0; i<met.ni; i++)
    for(j=0; j<met.nj; j++)
      for(k=0; k<met.nk; k++) {
        wf1.u[i][j][k] = ff->u[i][j][k];
        wf1.v[i][j][k] = ff->v[i][j][k];
        wf1.w[i][j][k] = ff->w[i][j][k];
      }

  n = 0;
  do {
    residual = 0.0;
    for(i=2; i<met.ni-2; i++)
      for(j=2; j<met.nj-2; j++)
        for(k=2; k<met.nk-2; k++) {
          ip1 = i + 1; jp1 = j + 1; kp1 = k + 1;
          im1 = i - 1; jm1 = j - 1; km1 = k - 1;
          ip2 = i + 2; jp2 = j + 2; kp2 = k + 2;
          im2 = i - 2; jm2 = j - 2; km2 = k - 2;

          g1x1 = met.g1x1[i][j][k];
          g1x2 = met.g1x2[i][j][k];
          g1x3 = met.g1x3[i][j][k];
          g2x1 = met.g2x1[i][j][k];
          g2x2 = met.g2x2[i][j][k];
          g2x3 = met.g2x3[i][j][k];
          g3x1 = met.g3x1[i][j][k];
          g3x2 = met.g3x2[i][j][k];
          g3x3 = met.g3x3[i][j][k];
```

```
co0 = - 2.0 * (lap.a1[i][j][k] + lap.a2[i][j][k] + lap.a3[i][j][k]);
cip = lap.a1[i][j][k] + 0.5 * lap.b1[i][j][k];
cim = lap.a1[i][j][k] - 0.5 * lap.b1[i][j][k];
cjp = lap.a2[i][j][k] + 0.5 * lap.b2[i][j][k];
cjm = lap.a2[i][j][k] - 0.5 * lap.b2[i][j][k];
ckp = lap.a3[i][j][k] + 0.5 * lap.b3[i][j][k];
ckm = lap.a3[i][j][k] - 0.5 * lap.b3[i][j][k];

u = wf1.u[i][j][k]; v = wf1.v[i][j][k]; w = wf1.w[i][j][k];
uc = g1x1 * u + g1x2 * v + g1x3 * w;
vc = g2x1 * u + g2x2 * v + g2x3 * w;
wc = g3x1 * u + g3x2 * v + g3x3 * w;

denom = 1.0 - dt * (- (fabs(uc) + fabs(vc) + fabs(wc)) * 6.0 * I4 + 1.0/Re * co0);

nlterm_ui = uc * (ff->u[im2][j][k] - ff->u[ip2][j][k]
                + 8.0 * (ff->u[ip1][j][k] - ff->u[im1][j][k])) * I12
  + fabs(uc) * (ff->u[ip2][j][k] + ff->u[im2][j][k]
              - 4.0 * (ff->u[ip1][j][k] + ff->u[im1][j][k])) * I4;
nlterm_uj = vc * (ff->u[i][jm2][k] - ff->u[i][jp2][k]
                + 8.0 * (ff->u[i][jp1][k] - ff->u[i][jm1][k])) * I12
  + fabs(vc) * (ff->u[i][jp2][k] + ff->u[i][jm2][k]
              - 4.0 * (ff->u[i][jp1][k] + ff->u[i][jm1][k])) * I4;
nlterm_uk = wc * (ff->u[i][j][km2] - ff->u[i][j][kp2]
                + 8.0 * (ff->u[i][j][kp1] - ff->u[i][j][km1])) * I12
  + fabs(wc) * (ff->u[i][j][kp2] + ff->u[i][j][km2]
              - 4.0 * (ff->u[i][j][kp1] + ff->u[i][j][km1])) * I4;

nlterm_vi = uc * (ff->v[im2][j][k] - ff->v[ip2][j][k]
                + 8.0 * (ff->v[ip1][j][k] - ff->v[im1][j][k])) * I12
  + fabs(uc) * (ff->v[ip2][j][k] + ff->v[im2][j][k]
              - 4.0 * (ff->v[ip1][j][k] + ff->v[im1][j][k])) * I4;
nlterm_vj = vc * (ff->v[i][jm2][k] - ff->v[i][jp2][k]
                + 8.0 * (ff->v[i][jp1][k] - ff->v[i][jm1][k])) * I12
  + fabs(vc) * (ff->v[i][jp2][k] + ff->v[i][jm2][k]
              - 4.0 * (ff->v[i][jp1][k] + ff->v[i][jm1][k])) * I4;
nlterm_vk = wc * (ff->v[i][j][km2] - ff->v[i][j][kp2]
                + 8.0 * (ff->v[i][j][kp1] - ff->v[i][j][km1])) * I12
  + fabs(wc) * (ff->v[i][j][kp2] + ff->v[i][j][km2]
              - 4.0 * (ff->v[i][j][kp1] + ff->v[i][j][km1])) * I4;

nlterm_wi = uc * (ff->w[im2][j][k] - ff->w[ip2][j][k]
                + 8.0 * (ff->w[ip1][j][k] - ff->w[im1][j][k])) * I12
  + fabs(uc) * (ff->w[ip2][j][k] + ff->w[im2][j][k]
              - 4.0 * (ff->w[ip1][j][k] + ff->w[im1][j][k])) * I4;
nlterm_wj = vc * (ff->w[i][jm2][k] - ff->w[i][jp2][k]
                + 8.0 * (ff->w[i][jp1][k] - ff->w[i][jm1][k])) * I12
  + fabs(vc) * (ff->w[i][jp2][k] + ff->w[i][jm2][k]
              - 4.0 * (ff->w[i][jp1][k] + ff->w[i][jm1][k])) * I4;
nlterm_wk = wc * (ff->w[i][j][km2] - ff->w[i][j][kp2]
                + 8.0 * (ff->w[i][j][kp1] - ff->w[i][j][km1])) * I12
  + fabs(wc) * (ff->w[i][j][kp2] + ff->w[i][j][km2]
              - 4.0 * (ff->w[i][j][kp1] + ff->w[i][j][km1])) * I4;

pu = (g1x1 * (ff->p[ip1][j][k] - ff->p[im1][j][k])
     + g2x1 * (ff->p[i][jp1][k] - ff->p[i][jm1][k])
     + g3x1 * (ff->p[i][j][kp1] - ff->p[i][j][km1])) * 0.5;
pv = (g1x2 * (ff->p[ip1][j][k] - ff->p[im1][j][k])
     + g2x2 * (ff->p[i][jp1][k] - ff->p[i][jm1][k])
     + g3x2 * (ff->p[i][j][kp1] - ff->p[i][j][km1])) * 0.5;
pw = (g1x3 * (ff->p[ip1][j][k] - ff->p[im1][j][k])
     + g2x3 * (ff->p[i][jp1][k] - ff->p[i][jm1][k])
     + g3x3 * (ff->p[i][j][kp1] - ff->p[i][j][km1])) * 0.5;

diff_u = cip * ff->u[ip1][j][k] + cim * ff->u[im1][j][k]
  + cjp * ff->u[i][jp1][k] + cjm * ff->u[i][jm1][k]
  + ckp * ff->u[i][j][kp1] + ckm * ff->u[i][j][km1]
  + lap.c1[i][j][k] * (ff->u[ip1][jp1][k] - ff->u[ip1][jm1][k]
                    - ff->u[im1][jp1][k] + ff->u[im1][jm1][k]) * 0.25
  + lap.c2[i][j][k] * (ff->u[i][jp1][kp1] - ff->u[i][jp1][km1]
                    - ff->u[i][jm1][kp1] + ff->u[i][jm1][km1]) * 0.25
  + lap.c3[i][j][k] * (ff->u[ip1][j][kp1] - ff->u[im1][j][kp1]
                    - ff->u[ip1][j][km1] + ff->u[im1][j][km1]) * 0.25;
diff_v = cip * ff->v[ip1][j][k] + cim * ff->v[im1][j][k]
  + cjp * ff->v[i][jp1][k] + cjm * ff->v[i][jm1][k]
  + ckp * ff->v[i][j][kp1] + ckm * ff->v[i][j][km1]
  + lap.c1[i][j][k] * (ff->v[ip1][jp1][k] - ff->v[ip1][jm1][k]
                    - ff->v[im1][jp1][k] + ff->v[im1][jm1][k]) * 0.25
  + lap.c2[i][j][k] * (ff->v[i][jp1][kp1] - ff->v[i][jp1][km1]
                    - ff->v[i][jm1][kp1] + ff->v[i][jm1][km1]) * 0.25
  + lap.c3[i][j][k] * (ff->v[ip1][j][kp1] - ff->v[im1][j][kp1]
                    - ff->v[ip1][j][km1] + ff->v[im1][j][km1]) * 0.25;
diff_w = cip * ff->w[ip1][j][k] + cim * ff->w[im1][j][k]
  + cjp * ff->w[i][jp1][k] + cjm * ff->w[i][jm1][k]
  + ckp * ff->w[i][j][kp1] + ckm * ff->w[i][j][km1]
  + lap.c1[i][j][k] * (ff->w[ip1][jp1][k] - ff->w[ip1][jm1][k]
                    - ff->w[im1][jp1][k] + ff->w[im1][jm1][k]) * 0.25
  + lap.c2[i][j][k] * (ff->w[i][jp1][kp1] - ff->w[i][jp1][km1]
                    - ff->w[i][jm1][kp1] + ff->w[i][jm1][km1]) * 0.25
  + lap.c3[i][j][k] * (ff->w[ip1][j][kp1] - ff->w[im1][j][kp1]
                    - ff->w[ip1][j][km1] + ff->w[im1][j][km1]) * 0.25;
```

```
          change_u = -(nlterm_ui + nlterm_uj + nlterm_uk) - pu + 1.0/Re * diff_u;
          change_v = -(nlterm_vi + nlterm_vj + nlterm_vk) - pv + 1.0/Re * diff_v;
          change_w = -(nlterm_wi + nlterm_wj + nlterm_wk) - pw + 1.0/Re * diff_w;
          relax_u = (wf1.u[i][j][k] + dt * change_u)/denom - ff->u[i][j][k];
          relax_v = (wf1.v[i][j][k] + dt * change_v)/denom - ff->v[i][j][k];
          relax_w = (wf1.w[i][j][k] + dt * change_w)/denom - ff->w[i][j][k];

          wf2.u[i][j][k] = ff->u[i][j][k] + omega * relax_u;
          wf2.v[i][j][k] = ff->v[i][j][k] + omega * relax_v;
          wf2.w[i][j][k] = ff->w[i][j][k] + omega * relax_w;

          residual += relax_u * relax_u + relax_v * relax_v + relax_w * relax_w;
        }
    for(i=2; i<met.ni-2; i++)
      for(j=2; j<met.nj-2; j++)
        for(k=2; k<met.nk-2; k++) {
          ff->u[i][j][k] = wf2.u[i][j][k];
          ff->v[i][j][k] = wf2.v[i][j][k];
          ff->w[i][j][k] = wf2.w[i][j][k];
        }

    condition_bound_velocity(ff, bc, fdirect);
    pass_overlapped_field(Velocity, where, ff);
    ++n;
    residual /= met.ni * met.nj * met.nk;

    if(index.myparent == PvmNoParent) {
      pvm_reduce(PvmSum, &residual, 1, PVM_DOUBLE, msgtag, Group, index.mygid);
      pvm_barrier(Group, index.nprocess);
      residual /= index.nprocess;

      pvm_initsend(PvmDataRaw);
      pvm_pkdouble(&residual, oneitem, stride);
      pvm_bcast(Group, msgtag);
    }
    else {
      pvm_reduce(PvmSum, &residual, 1, PVM_DOUBLE, msgtag, Group, Master);
      pvm_barrier(Group, index.nprocess);

      pvm_recv(index.myparent, msgtag);
      pvm_upkdouble(&residual, oneitem, stride);
    }
  } while((residual > eps) && (n < 100));

  return residual;
}
```

```
void condition_bound_inflow( double* fdirect )
{
  double alpha, beta;

  alpha = 0.0;
  beta = 0.0;

  fdirect[0] = cos(alpha);
  fdirect[1] = sin(alpha);
  fdirect[2] = sin(beta);
}

void condition_bound_pressure( FlowField* ff, BCondition bc )
{
  int i,j,k;
  int one, endi, endj, endk;
  int two, nexti, nextj, nextk;

  one = 1; endi = ff->ni-2; endj = ff->nj-2; endk = ff->nk-2;
  two = 2; nexti = endi-1; nextj = endj-1; nextk = endk-1;

  for(j=1; j<endj; j++)
    for(k=1; k<endk; k++) {
      switch(bc.p[one][j][k]) {
      case UnDisturbed:
        ff->p[one][j][k] = 0.0;
        break;
      case Neumann:
        ff->p[one][j][k] = ff->p[two][j][k];
        break;
      }
      switch(bc.p[endi][j][k]) {
      case UnDisturbed:
        ff->p[endi][j][k] = 0.0;
        break;
      case Neumann:
        ff->p[endi][j][k] = ff->p[nexti][j][k];
        break;
      }
    }
  for(i=1; i<endi; i++)
    for(k=1; k<endk; k++) {
      switch(bc.p[i][one][k]) {
      case UnDisturbed:
        ff->p[i][one][k] = 0.0;
        break;
      case Neumann:
        ff->p[i][one][k] = ff->p[i][two][k];
        break;
      }
      switch(bc.p[i][endj][k]) {
      case UnDisturbed:
        ff->p[i][endj][k] = 0.0;
        break;
      case Neumann:
        ff->p[i][endj][k] = ff->p[i][nextj][k];
        break;
      }
    }
  for(i=1; i<endi; i++)
    for(j=1; j<endj; j++) {
      switch(bc.p[i][j][one]) {
      case UnDisturbed:
        ff->p[i][j][one] = 0.0;
        break;
      case Neumann:
        ff->p[i][j][one] = ff->p[i][j][two];
        break;
      }
      switch(bc.p[i][j][endk]) {
      case UnDisturbed:
        ff->p[i][j][endk] = 0.0;
        break;
      case Neumann:
        ff->p[i][j][endk] = ff->p[i][j][nextk];
        break;
      }
    }
}

void condition_bound_velocity( FlowField* ff, BCondition bc, double* fdirect )
{
  int i,j,k;
  int zero, one, two;
  int nexti, nextj, nextk;
  int next2i, next2j, next2k;
  int endi, endj, endk;

  zero = 0; one = 1; two = 2;
  endi = ff->ni-1; endj = ff->nj-1; endk = ff->nk-1;
  nexti = endi-1; nextj = endj-1; nextk = endk-1;
```

```
next2i = endi-2; next2j = endj-2; next2k = endk-2;

for(j=1; j<endj; j++)
  for(k=1; k<endk; k++) {
    switch(bc.v[one][j][k]) {
    case UnDisturbed:
      ff->u[one][j][k] = fdirect[0];
      ff->v[one][j][k] = fdirect[1];
      ff->w[one][j][k] = fdirect[2];
      break;
    case NonSlip:
      ff->u[one][j][k] = 0.0;
      ff->v[one][j][k] = 0.0;
      ff->w[one][j][k] = 0.0;
      break;
    case Slip:
      break;
    case Outflow:
      ff->u[one][j][k] = ff->u[two][j][k];
      ff->v[one][j][k] = ff->v[two][j][k];
      ff->w[one][j][k] = ff->w[two][j][k];
      break;
    case Periodical:
      break;
    }
    switch(bc.v[nexti][j][k]) {
    case UnDisturbed:
      ff->u[nexti][j][k] = fdirect[0];
      ff->v[nexti][j][k] = fdirect[1];
      ff->w[nexti][j][k] = fdirect[2];
      break;
    case NonSlip:
      ff->u[nexti][j][k] = 0.0;
      ff->v[nexti][j][k] = 0.0;
      ff->w[nexti][j][k] = 0.0;
      break;
    case Slip:
      break;
    case Outflow:
      ff->u[nexti][j][k] = ff->u[next2i][j][k];
      ff->v[nexti][j][k] = ff->v[next2i][j][k];
      ff->w[nexti][j][k] = ff->w[next2i][j][k];
      break;
    case Periodical:
      break;
    }
  }
for(j=0; j<ff->nj; j++)
  for(k=0; k<ff->nk; k++) {
    switch(bc.v[zero][j][k]) {
    case UnDisturbed:
      ff->u[zero][j][k] = fdirect[0];
      ff->v[zero][j][k] = fdirect[1];
      ff->w[zero][j][k] = fdirect[2];
      break;
    case NonSlip:
      ff->u[zero][j][k] = -ff->u[two][j][k];
      ff->v[zero][j][k] = -ff->v[two][j][k];
      ff->w[zero][j][k] = -ff->w[two][j][k];
      break;
    case Slip:
      break;
    case Outflow:
      ff->u[zero][j][k] = ff->u[two][j][k];
      ff->v[zero][j][k] = ff->v[two][j][k];
      ff->w[zero][j][k] = ff->w[two][j][k];
      break;
    case Periodical:
      break;
    }
    switch(bc.v[endi][j][k]) {
    case UnDisturbed:
      ff->u[endi][j][k] = fdirect[0];
      ff->v[endi][j][k] = fdirect[1];
      ff->w[endi][j][k] = fdirect[2];
      break;
    case NonSlip:
      ff->u[endi][j][k] = -ff->u[next2i][j][k];
      ff->v[endi][j][k] = -ff->v[next2i][j][k];
      ff->w[endi][j][k] = -ff->w[next2i][j][k];
      break;
    case Slip:
      break;
    case Outflow:
      ff->u[endi][j][k] = ff->u[next2i][j][k];
      ff->v[endi][j][k] = ff->v[next2i][j][k];
      ff->w[endi][j][k] = ff->w[next2i][j][k];
      break;
    case Periodical:
      break;
    }
  }
```

```
for(i=1; i<endi; i++)
  for(k=1; k<endk; k++) {
    switch(bc.v[i][one][k]) {
    case UnDisturbed:
      ff->u[i][one][k] = fdirect[0];
      ff->v[i][one][k] = fdirect[1];
      ff->w[i][one][k] = fdirect[2];
      break;
    case NonSlip:
      ff->u[i][one][k] = 0.0;
      ff->v[i][one][k] = 0.0;
      ff->w[i][one][k] = 0.0;
      break;
    case Slip:
      break;
    case Outflow:
      ff->u[i][one][k] = ff->u[i][two][k];
      ff->v[i][one][k] = ff->v[i][two][k];
      ff->w[i][one][k] = ff->w[i][two][k];
      break;
    case Periodical:
      break;
    }
    switch(bc.v[i][nextj][k]) {
    case UnDisturbed:
      ff->u[i][nextj][k] = fdirect[0];
      ff->v[i][nextj][k] = fdirect[1];
      ff->w[i][nextj][k] = fdirect[2];
      break;
    case NonSlip:
      ff->u[i][nextj][k] = 0.0;
      ff->v[i][nextj][k] = 0.0;
      ff->w[i][nextj][k] = 0.0;
      break;
    case Slip:
      break;
    case Outflow:
      ff->u[i][nextj][k] = ff->u[i][next2j][k];
      ff->v[i][nextj][k] = ff->v[i][next2j][k];
      ff->w[i][nextj][k] = ff->w[i][next2j][k];
      break;
    case Periodical:
      break;
    }
  }
```

```
for(i=0; i<ff->ni; i++)
  for(k=0; k<ff->nk; k++) {
    switch(bc.v[i][zero][k]) {
    case UnDisturbed:
      ff->u[i][zero][k] = fdirect[0];
      ff->v[i][zero][k] = fdirect[1];
      ff->w[i][zero][k] = fdirect[2];
      break;
    case NonSlip:
      ff->u[i][zero][k] = -ff->u[i][two][k];
      ff->v[i][zero][k] = -ff->v[i][two][k];
      ff->w[i][zero][k] = -ff->w[i][two][k];
      break;
    case Slip:
      break;
    case Outflow:
      ff->u[i][zero][k] = ff->u[i][two][k];
      ff->v[i][zero][k] = ff->v[i][two][k];
      ff->w[i][zero][k] = ff->w[i][two][k];
      break;
    case Periodical:
      break;
    }
    switch(bc.v[i][endj][k]) {
    case UnDisturbed:
      ff->u[i][endj][k] = fdirect[0];
      ff->v[i][endj][k] = fdirect[1];
      ff->w[i][endj][k] = fdirect[2];
      break;
    case NonSlip:
      ff->u[i][endj][k] = -ff->u[i][next2j][k];
      ff->v[i][endj][k] = -ff->v[i][next2j][k];
      ff->w[i][endj][k] = -ff->w[i][next2j][k];
      break;
    case Slip:
      break;
    case Outflow:
      ff->u[i][endj][k] = ff->u[i][next2j][k];
      ff->v[i][endj][k] = ff->v[i][next2j][k];
      ff->w[i][endj][k] = ff->w[i][next2j][k];
      break;
    case Periodical:
      break;
    }
  }
```

```
for(i=1; i<endi; i++)
  for(j=1; j<endj; j++) {
    switch(bc.v[i][j][one]) {
    case UnDisturbed:
      ff->u[i][j][one] = fdirect[0];
      ff->v[i][j][one] = fdirect[1];
      ff->w[i][j][one] = fdirect[2];
      break;
    case NonSlip:
      ff->u[i][j][one] = 0.0;
      ff->v[i][j][one] = 0.0;
      ff->w[i][j][one] = 0.0;
      break;
    case Slip:
      break;
    case Outflow:
      ff->u[i][j][one] = ff->u[i][j][two];
      ff->v[i][j][one] = ff->v[i][j][two];
      ff->w[i][j][one] = ff->w[i][j][two];
      break;
    case Periodical:
      break;
    }
    switch(bc.v[i][j][nextk]) {
    case UnDisturbed:
      ff->u[i][j][nextk] = fdirect[0];
      ff->v[i][j][nextk] = fdirect[1];
      ff->w[i][j][nextk] = fdirect[2];
      break;
    case NonSlip:
      ff->u[i][j][nextk] = 0.0;
      ff->v[i][j][nextk] = 0.0;
      ff->w[i][j][nextk] = 0.0;
      break;
    case Slip:
      break;
    case Outflow:
      ff->u[i][j][nextk] = ff->u[i][j][next2k];
      ff->v[i][j][nextk] = ff->v[i][j][next2k];
      ff->w[i][j][nextk] = ff->w[i][j][next2k];
      break;
    case Periodical:
      break;
    }
  }
for(i=0; i<ff->ni; i++)
```

```
    for(j=0; j<ff->nj; j++) {
      switch(bc.v[i][j][zero]) {
      case UnDisturbed:
        ff->u[i][j][zero] = fdirect[0];
        ff->v[i][j][zero] = fdirect[1];
        ff->w[i][j][zero] = fdirect[2];
        break;
      case NonSlip:
        ff->u[i][j][zero] = -ff->u[i][j][two];
        ff->v[i][j][zero] = -ff->v[i][j][two];
        ff->w[i][j][zero] = -ff->w[i][j][two];
        break;
      case Slip:
        break;
      case Outflow:
        ff->u[i][j][zero] = ff->u[i][j][two];
        ff->v[i][j][zero] = ff->v[i][j][two];
        ff->w[i][j][zero] = ff->w[i][j][two];
        break;
      case Periodical:
        break;
      }
      switch(bc.v[i][j][endk]) {
      case UnDisturbed:
        ff->u[i][j][endk] = fdirect[0];
        ff->v[i][j][endk] = fdirect[1];
        ff->w[i][j][endk] = fdirect[2];
        break;
      case NonSlip:
        ff->u[i][j][endk] = -ff->u[i][j][next2k];
        ff->v[i][j][endk] = -ff->v[i][j][next2k];
        ff->w[i][j][endk] = -ff->w[i][j][next2k];
        break;
      case Slip:
        break;
      case Outflow:
        ff->u[i][j][endk] = ff->u[i][j][next2k];
        ff->v[i][j][endk] = ff->v[i][j][next2k];
        ff->w[i][j][endk] = ff->w[i][j][next2k];
        break;
      case Periodical:
        break;
      }
    }
}
```

```
void determine_my_position( PVM_param index, LoadDiv load, TaskPosit* where )
{
  int i, j, ni, nj;

  for(i=0; i<index.row; i++)
    for(j=0; j<index.col; j++)
      if(load.procmtrx[i][j] == index.mytid) {
        ni = i; nj = j; break; }
  if((ni + nj)%2 == 0)
    where->posit = Even;
  else
    where->posit = Odd;

  where->iplus = Off; where->iminus = Off;
  where->jplus = Off; where->jminus = Off;
  if(ni-1 >= 0) where->iminus = load.procmtrx[ni-1][nj];
  if(ni+1 < load.row) where->iplus = load.procmtrx[ni+1][nj];
  if(nj-1 >= 0) where->jminus = load.procmtrx[ni][nj-1];
  if(nj+1 < load.col) where->jplus = load.procmtrx[ni][nj+1];
}

void pass_overlapped_field( int fieldtype, TaskPosit where, FlowField* ff )
{
  if(where.posit == Even) {
    sendrecv_overlapped_field( Send, fieldtype, where, ff );
    sendrecv_overlapped_field( Receive, fieldtype, where, ff );
  }
  if(where.posit == Odd) {
    sendrecv_overlapped_field( Receive, fieldtype, where, ff );
    sendrecv_overlapped_field( Send, fieldtype, where, ff );
  }
}

void sendrecv_overlapped_field( int passtype, int fieldtype,
                                TaskPosit where, FlowField* ff )
{
  int is, ie, js, je, ks, ke;
  int overlap = 4;
  int halflap = 2;

  ks = 0; ke = ff->nk - 1;

  if(where.iplus > 0) {
    js = 0; je = ff->nj - 1;
    switch(passtype) {
    case Send:
```

```
    is = ff->ni - overlap; ie = is + 1;
    send_flowfield(where.iplus, fieldtype, is, ie, js, je, ks, ke, ff);
    break;
  case Receive:
    is = ff->ni - halflap; ie = is + 1;
    recv_flowfield(where.iplus, fieldtype, is, ie, js, je, ks, ke, ff);
    break;
  }
}
if(where.iminus > 0) {
  js = 0; je = ff->nj - 1;
  switch(passtype) {
  case Send:
    is = 2; ie = is + 1;
    send_flowfield(where.iminus, fieldtype, is, ie, js, je, ks, ke, ff);
    break;
  case Receive:
    is = 0; ie = is + 1;
    recv_flowfield(where.iminus, fieldtype, is, ie, js, je, ks, ke, ff);
    break;
  }
}
if(where.jplus > 0) {
  is = 0; ie = ff->ni - 1;
  switch(passtype) {
  case Send:
    js = ff->nj - overlap; je = js + 1;
    send_flowfield(where.jplus, fieldtype, is, ie, js, je, ks, ke, ff);
    break;
  case Receive:
    js = ff->nj - halflap; je = js + 1;
    recv_flowfield(where.jplus, fieldtype, is, ie, js, je, ks, ke, ff);
    break;
  }
}
if(where.jminus > 0) {
  is = 0; ie = ff->ni - 1;
  switch(passtype) {
  case Send:
    js = 2; je = js + 1;
    send_flowfield(where.jminus, fieldtype, is, ie, js, je, ks, ke, ff);
    break;
  case Receive:
    js = 0; je = js + 1;
    recv_flowfield(where.jminus, fieldtype, is, ie, js, je, ks, ke, ff);
    break;
    }
  }
}

void send_flowfield( int procid, int fieldtype,
                     int is, int ie, int js, int je, int ks, int ke, FlowField* ff )
{
  switch(fieldtype) {
  case Velocity:
    send_double_tensor( procid, is, ie, js, je, ks, ke, ff->u );
    send_double_tensor( procid, is, ie, js, je, ks, ke, ff->v );
    send_double_tensor( procid, is, ie, js, je, ks, ke, ff->w );
    break;
  case Pressure:
    send_double_tensor( procid, is, ie, js, je, ks, ke, ff->p );
    break;
  }
}

void recv_flowfield( int procid, int fieldtype,
                     int is, int ie, int js, int je, int ks, int ke, FlowField* ff )
{
  switch(fieldtype) {
  case Velocity:
    receive_double_tensor( procid, is, ie, js, je, ks, ke, ff->u );
    receive_double_tensor( procid, is, ie, js, je, ks, ke, ff->v );
    receive_double_tensor( procid, is, ie, js, je, ks, ke, ff->w );
    break;
  case Pressure:
    receive_double_tensor( procid, is, ie, js, je, ks, ke, ff->p );
    break;
  }
}

void send_double_tensor( int procid, int is, int ie, int js, int je, int ks, int ke,
                         double*** data )
{
  int buffersize;
  double *databuffer;
  int zero = 0;
  int npacked;
  int msgtag = 2;
  int stride = 1;

  buffersize = (ie-is+1) * (je-js+1) * (ke-ks+1);
  databuffer = dvector(zero, buffersize-1);
  npacked = d3tensor_to_vector(data, databuffer, is, ie, js, je, ks, ke);
  if(npacked != buffersize) {
    printf("!!! packed data size is not the one expected.\n");
    exit(1);
  }
  pvm_initsend(PvmDataRaw);
  pvm_pkdouble(databuffer, buffersize, stride);
  pvm_send(procid, msgtag);

  free_dvector(databuffer, zero, buffersize-1);
}

void send_int_tensor( int procid, int is, int ie, int js, int je, int ks, int ke,
                      int*** data )
{
  int buffersize;
  int *databuffer;
  int zero = 0;
  int npacked;
  int msgtag = 2;
  int stride = 1;

  buffersize = (ie-is+1) * (je-js+1) * (ke-ks+1);
  databuffer = ivector(zero, buffersize-1);

  npacked = i3tensor_to_vector(data, databuffer, is, ie, js, je, ks, ke);
  if(npacked != buffersize) {
    printf("!!! packed data size is not the one expected.\n");
    exit(1);
  }
  pvm_initsend(PvmDataRaw);
  pvm_pkint(databuffer, buffersize, stride);
  pvm_send(procid, msgtag);

  free_ivector(databuffer, zero, buffersize-1);
}

void receive_double_tensor( int donner, int is, int ie, int js, int je, int ks, int ke,
                            double*** data )
{
  int buffersize;
  double *databuffer;
  int zero = 0;
  int npacked;
  int msgtag = 2;
  int stride = 1;
  int ni, nj, nk;

  ni = ie - is + 1;
  nj = je - js + 1;
  nk = ke - ks + 1;
  buffersize = ni * nj * nk;
  databuffer = dvector(zero, buffersize-1);

  pvm_recv(donner, msgtag);
  pvm_upkdouble(databuffer, buffersize, stride);
  npacked = vector_to_d3tensor(data, databuffer, is, ie, js, je, ks, ke);
  if(npacked != buffersize) {
    printf("!!! unpacked data size is not the one expected.\n");
    exit(1);
  }

  free_dvector(databuffer, zero, buffersize-1);
}

void receive_int_tensor( int donner, int is, int ie, int js, int je, int ks, int ke,
                         int*** data )
{
  int buffersize;
  int *databuffer;
  int zero = 0;
  int npacked;
  int msgtag = 2;
  int stride = 1;
  int ni, nj, nk;

  ni = ie - is + 1;
  nj = je - js + 1;
  nk = ke - ks + 1;
  buffersize = ni * nj * nk;
  databuffer = ivector(zero, buffersize-1);

  pvm_recv(donner, msgtag);
  pvm_upkint(databuffer, buffersize, stride);
  npacked = vector_to_i3tensor(data, databuffer, is, ie, js, je, ks, ke);
  if(npacked != buffersize) {
    printf("!!! unpacked data size is not the one expected.\n");
    exit(1);
  }

  free_ivector(databuffer, zero, buffersize-1);
```

```
}

void make_loadbalance_index( Grid co, PVM_param index, LoadDiv* load )
{
  int overlap = 4;
  divide_overlapped_line(co.ni, overlap, index.row, load->istart, load->iend);
  divide_overlapped_line(co.nj, overlap, index.col, load->jstart, load->jend);
}

void divide_overlapped_line( int np, int overlap, int ndiv, int* startindex, int* endindex )
{
  int virtualsize;
  int dividedsize;
  int remainder;
  int n;

  virtualsize = np + overlap * (ndiv - 1);
  dividedsize = virtualsize/ndiv;
  remainder = virtualsize%ndiv;

  for(n=0; n<remainder; n++)
    startindex[n] = (dividedsize + 1) * n - overlap * n;
  for(n=remainder; n<ndiv; n++)
    startindex[n] = (dividedsize + 1) * remainder
      + dividedsize * (n - remainder) - overlap * n;

  for(n=0; n<remainder; n++)
    endindex[n] = startindex[n] + dividedsize;
  for(n=remainder; n<ndiv-1; n++)
    endindex[n] = startindex[n] + dividedsize - 1;
  endindex[ndiv-1] = np - 1;
}

void deliver_data_to_master( LoadDiv load, Grid gco, FlowField gff, BCondition gbc,
                             Grid* co, FlowField* ff, BCondition *bc )
{
  int i, j, k;
  int zero, ni, nj, nk;

  zero = 0;
  ni = load.iend[zero] - load.istart[zero] + 1;
  nj = load.jend[zero] - load.jstart[zero] + 1;
  nk = gco.nk;

  for(i=0; i<ni; i++)
    for(j=0; j<nj; j++)
```

```
      for(k=0; k<nk; k++) {
        co->x[i][j][k] = gco.x[i][j][k];
        co->y[i][j][k] = gco.y[i][j][k];
        co->z[i][j][k] = gco.z[i][j][k];
        ff->u[i][j][k] = gff.u[i][j][k];
        ff->v[i][j][k] = gff.v[i][j][k];
        ff->w[i][j][k] = gff.w[i][j][k];
        ff->p[i][j][k] = gff.p[i][j][k];
        bc->p[i][j][k] = gbc.p[i][j][k];
        bc->v[i][j][k] = gbc.v[i][j][k];
      }
}

void restore_data_to_master( LoadDiv load, FlowField* gff, FlowField ff )
{
  int i, j, k;
  int zero, ni, nj, nk;

  zero = 0;
  ni = load.iend[zero] - load.istart[zero] + 1;
  nj = load.jend[zero] - load.jstart[zero] + 1;
  nk = gff->nk;
  for(i=0; i<ni; i++)
    for(j=0; j<nj; j++)
      for(k=0; k<nk; k++) {
        gff->u[i][j][k] = ff.u[i][j][k];
        gff->v[i][j][k] = ff.v[i][j][k];
        gff->w[i][j][k] = ff.w[i][j][k];
        gff->p[i][j][k] = ff.p[i][j][k];
      }
}

void deliver_data_to_slave( PVM_param index, LoadDiv load,
                         Grid co, FlowField ff, BCondition bc )
{
  int n, i, j;
  int ni, nj;
  int procid, is, ie, js, je, ks, ke;

  for(n=1; n<index.nprocess; n++) {
    for(i=0; i<index.row; i++)
      for(j=0; j<index.col; j++)
        if(load.procmtrx[i][j] == index.tids[n]) {
          ni = i; nj = j; break; }
    procid = load.procmtrx[ni][nj];
    is = load.istart[ni]; ie = load.iend[ni];
    js = load.jstart[nj]; je = load.jend[nj];
    ks = 0, ke = co.nk - 1;

    pass_data_to_slave(procid, is, ie, js, je, ks, ke, co, ff, bc);
  }
}

void pass_data_to_slave( int procid, int is, int ie, int js, int je, int ks, int ke,
                         Grid co, FlowField ff, BCondition bc )
{
  int ni, nj, nk;
  int msgtag = 2;
  int nitem = 1;
  int stride = 1;

  ni = ie - is + 1;
  nj = je - js + 1;
  nk = co.nk;

  pvm_initsend(PvmDataRaw);
  pvm_pkint(&ni, nitem, stride);
  pvm_pkint(&nj, nitem, stride);
  pvm_pkint(&nk, nitem, stride);
  pvm_send(procid, msgtag);

  send_double_tensor(procid, is, ie, js, je, ks, ke, co.x);
  send_double_tensor(procid, is, ie, js, je, ks, ke, co.y);
  send_double_tensor(procid, is, ie, js, je, ks, ke, co.z);
  send_double_tensor(procid, is, ie, js, je, ks, ke, ff.u);
  send_double_tensor(procid, is, ie, js, je, ks, ke, ff.v);
  send_double_tensor(procid, is, ie, js, je, ks, ke, ff.w);
  send_double_tensor(procid, is, ie, js, je, ks, ke, ff.p);
  send_int_tensor(procid, is, ie, js, je, ks, ke, bc.p);
  send_int_tensor(procid, is, ie, js, je, ks, ke, bc.v);
}

void receive_data_from_master( PVM_param index, Grid* co, FlowField* ff, BCondition* bc )
{
  int ni, nj, nk;
  int msgtag = 2;
  int nitem = 1;
  int stride = 1;
  int zero = 0;
  int parent = index.myparent;
  int is, ie, js, je, ks, ke;
  pvm_recv(parent, msgtag);
  pvm_upkint(&ni, nitem, stride);
  pvm_upkint(&nj, nitem, stride);
  pvm_upkint(&nk, nitem, stride);

  allocate_geomfield(co, bc, ni, nj, nk);
  allocate_flowfield(ff, ni, nj, nk);

  is = zero, ie = ni - 1;
  js = zero, je = nj - 1;
  ks = zero, ke = nk - 1;
  receive_double_tensor(parent, is, ie, js, je, ks, ke, co->x);
  receive_double_tensor(parent, is, ie, js, je, ks, ke, co->y);
  receive_double_tensor(parent, is, ie, js, je, ks, ke, co->z);
  receive_double_tensor(parent, is, ie, js, je, ks, ke, ff->u);
  receive_double_tensor(parent, is, ie, js, je, ks, ke, ff->v);
  receive_double_tensor(parent, is, ie, js, je, ks, ke, ff->w);
  receive_double_tensor(parent, is, ie, js, je, ks, ke, ff->p);
  receive_int_tensor(parent, is, ie, js, je, ks, ke, bc->p);
  receive_int_tensor(parent, is, ie, js, je, ks, ke, bc->v);
}
```

Appendix B

Program Lists for Benchmark

The programs used for the benchmark shown in the section 4.2.2 are listed here. For C++, the very same code is used for both with-ET (Expression Template) and without-ET versions. The code in Fortran90 listed here is the version with the array operation.

Benchmark.cpp (C++)

```
#include <iostream>
#include <cstdlib>
#include "MathVector.h"

using namespace std;

int main(int argc, char** argv)
{
  int vsize = 10;
  int loop = 100000;
  if (argc > 1)
    vsize = atoi(argv[1]);

  MathVector<double> v1(vsize);
  MathVector<double> v2(vsize);
  MathVector<double> v3(vsize);
  MathVector<double> v4(vsize);
  MathVector<double> v5(vsize);
  MathVector<double> v6(vsize);

  for (int n=0; n<vsize; n++) {
    v2(n) = 1.0*(n+1);
    v3(n) = 0.1*(n+1);
    v4(n) = 0.01*(n+1);
    v5(n) = 0.001*(n+1);
    v6(n) = 0.0001*(n+1);
  }

  for (int n=0; n<loop; n++)
    v1 = v2 + v3 + v4 + v5 + v6;

  return 0;
}
```

MathVector.h (without Expression Template)

```
#ifndef __MathVector_H__
#define __MathVector_H__

template<class Type>
class MathVector
{
 public:
  //  typedef Type VectDataType;
  typedef Type TypeName;

  MathVector() : vlength(0) {}

  MathVector(const int& vectorsize)
    { allocate(vectorsize); }

  MathVector(const MathVector<Type>& other)
    { copy(other); }

  ~MathVector()
    { cleanup(); }
```

```
int   length() const
  { return vlength; }

void  create(const int& vectorsize)
  {
    cleanup();
    allocate(vectorsize);
  }

void  set(const int& index, const Type& value)
  { data[index] = value; }

Type  get(const int& index) const
  { return data[index]; }

MathVector<Type>&    multiply(const Type& multiplier)
  {
    for (int n=0; n<length(); n++)
      data[n] *= multiplier;
    return *this;
  }

MathVector<Type>&    multiply(const Type& multiplier, const MathVector<Type>& multiplicand)
  {
    for (int n=0; n<length(); n++)
      data[n] = multiplier*multiplicand(n);
    return *this;
  }

MathVector<Type>&    multiply(const MathVector<Type>& multiplicand, const Type& multiplier)
  {
    for (int n=0; n<length(); n++)
      data[n] = multiplier*multiplicand(n);
    return *this;
  }

MathVector<Type>&    divide(const Type& denominator)
  {
    for (int n=0; n<length(); n++)
      data[n] /= denominator;
    return *this;
  }

MathVector<Type>&    divide(const MathVector<Type>& numerator, const Type& denominator)
  {
    for (int n=0; n<length(); n++)
```

```
      data[n] = numerator(n)/denominator;
    return *this;
  }

Type& operator ()(const int& index)
  { return data[index]; }

Type operator ()(const int& index) const
  { return data[index]; }

MathVector<Type>& operator =(const MathVector<Type>& other)
{
  if (this == &other)
    return *this;

  cleanup();
  copy(other);

  return *this;
}

bool  operator ==(const MathVector<Type>& other);
bool  operator !=(const MathVector<Type>& other);

friend MathVector<Type> operator +(const MathVector<Type>& lhs,
                                   const MathVector<Type>& rhs)
{
  MathVector<Type> tmp(lhs.length());
  for (int n=0; n<lhs.length(); n++)
    tmp(n) = lhs(n) + rhs(n);
  return tmp;
}

friend MathVector<Type> operator -(const MathVector<Type>& lhs,
                                   const MathVector<Type>& rhs)
{
  MathVector<Type> tmp(lhs.length());
  for (int n=0; n<lhs.length(); n++)
    tmp(n) = lhs(n) + rhs(n);
  return tmp;
}

friend MathVector<Type> operator *(const MathVector<Type>& lhs,
                                   const MathVector<Type>& rhs)
{
  MathVector<Type> tmp(lhs.length());
```

```
   for (int n=0; n<lhs.length(); n++)
     tmp(n) = lhs(n)*rhs(n);
   return tmp;
 }

 friend MathVector<Type> operator *(const Type& lhs,
                                    const MathVector<Type>& rhs)
 {
   MathVector<Type> tmp(rhs.length());
   for (int n=0; n<rhs.length(); n++)
     tmp(n) = lhs*rhs(n);
   return tmp;
 }

 friend MathVector<Type> operator *(const MathVector<Type>& lhs,
                                    const Type& rhs)
 {
   MathVector<Type> tmp(lhs.length());
   for (int n=0; n<lhs.length(); n++)
     tmp(n) = lhs(n)*rhs;
   return tmp;
 }

 friend MathVector<Type> operator /(const MathVector<Type>& lhs,
                                    const Type& rhs)
 {
   MathVector<Type> tmp(lhs.length());
   for (int n=0; n<lhs.length(); n++)
     tmp(n) = lhs(n)/rhs;
   return tmp;
 }

 MathVector<Type> operator -()
 {
   MathVector<Type> tmp(length());
   for (int n=0; n<length(); n++)
     tmp(n) = -data[n];
   return tmp;
 }

protected:
 int   vlength;
 Type* data;

 void  allocate(const int& vectorsize);
 void  cleanup();
```

```
  void  copy(const MathVector<Type>& other);
};


template<class Type>
bool
MathVector<Type>::operator ==(const MathVector<Type>& other)
{
  if (this == &other)
    return true;

  if (length() != other.length())
    return false;

  for (int n=0; n<length(); n++)
    if (get(n) != other.get(n))
      return false;

  return true;
}

template<class Type>
bool
MathVector<Type>::operator !=(const MathVector<Type>& other)
{
  if (this == &other)
    return false;

  if (length() != other.length())
    return true;

  for (int n=0; n<length(); n++)
    if (get(n) != other.get(n))
      return true;

  return false;
}

template<class Type>
void
MathVector<Type>::allocate(const int& vectorsize)
{
  vlength = vectorsize;
  data = new Type[length()];
}
```

```
template<class Type>
void
MathVector<Type>::cleanup()
{
  if (length() != 0)
    delete[] data;
}

template<class Type>
void
MathVector<Type>::copy(const MathVector<Type>& other)
{
  allocate(other.length());
  for (int n=0; n<length(); n++)
    set(n,other.get(n));
}

#endif // __MathVector_H__
```

MathVector.h (with Expression Template)

```
#ifndef __MathVector_H__
#define __MathVector_H__

template<class Type>
class MathVector
{
 public:
  //  typedef Type VectDataType;
  typedef Type TypeName;

  MathVector() : vlength(0) {}

  MathVector(const int& vectorsize)
    { allocate(vectorsize); }

  MathVector(const MathVector<Type>& other)
    { copy(other); }

  ~MathVector()
    { cleanup(); }

  int   length() const
    { return vlength; }

  void  create(const int& vectorsize)
    {
      cleanup();
      allocate(vectorsize);
    }

  void  set(const int& index, const Type& value)
    { data[index] = value; }

  Type  get(const int& index) const
    { return data[index]; }

  MathVector<Type>&    multiply(const Type& multiplier)
    {
      for (int n=0; n<length(); n++)
        data[n] *= multiplier;
      return *this;
    }

  MathVector<Type>&    multiply(const Type& multiplier, const MathVector<Type>& multiplicand)
    {
      for (int n=0; n<length(); n++)
        data[n] = multiplier*multiplicand(n);
      return *this;
    }

  MathVector<Type>&    multiply(const MathVector<Type>& multiplicand, const Type& multiplier)
    {
      for (int n=0; n<length(); n++)
        data[n] = multiplier*multiplicand(n);
      return *this;
    }

  MathVector<Type>&    divide(const Type& denominator)
    {
      for (int n=0; n<length(); n++)
        data[n] /= denominator;
      return *this;
    }

  MathVector<Type>&    divide(const MathVector<Type>& numerator, const Type& denominator)
```

```
    {
      for (int n=0; n<length(); n++)
        data[n] = numerator(n)/denominator;
      return *this;
    }

  Type& operator ()(const int& index)
    { return data[index]; }

  Type operator ()(const int& index) const
    { return data[index]; }

  template<class Expr>
  MathVector<Type>& operator=(const Expr& expression)
    {
      for (int n=0; n<length(); n++)
        data[n] = expression.get(n);

      return *this;
    }

  bool  operator ==(const MathVector<Type>& other);
  bool  operator !=(const MathVector<Type>& other);

 protected:
  int   vlength;
  Type* data;

  void  allocate(const int& vectorsize);
  void  cleanup();
  void  copy(const MathVector<Type>& other);
};


template<class Type>
bool
MathVector<Type>::operator ==(const MathVector<Type>& other)
{
  if (this == &other)
    return true;

  if (length() != other.length())
    return false;

  for (int n=0; n<length(); n++)
    if (get(n) != other.get(n))
```

```
     return false;

  return true;
}

template<class Type>
bool
MathVector<Type>::operator !=(const MathVector<Type>& other)
{
  if (this == &other)
    return false;

  if (length() != other.length())
    return true;

  for (int n=0; n<length(); n++)
    if (get(n) != other.get(n))
      return true;

  return false;

}

template<class Type>
void
MathVector<Type>::allocate(const int& vectorsize)
{
  vlength = vectorsize;
  data = new Type[length()];
}

template<class Type>
void
MathVector<Type>::cleanup()
{
  if (length() != 0)
    delete[] data;
}

template<class Type>
void
MathVector<Type>::copy(const MathVector<Type>& other)
{
  allocate(other.length());
  for (int n=0; n<length(); n++)
    set(n,other.get(n));
```

```
}

#endif // __MathVector_H__
```

MathExpression.h

```
#ifndef __MathExpression_H__
#define __MathExpression_H__

/*
- - - - - - - - - - - - - - - - - - - -
 Expressions
- - - - - - - - - - - - - - - - - - - -
*/
template<class Op, class Expression1, class Expression2>
class BinaryExpression
{
 public:
  typedef typename Expression1::TypeName TypeName;

  BinaryExpression(const Expression1& e1, const Expression2& e2)
    : expr1(e1),expr2(e2)
    {}

  TypeName get(const int& index) const
  {
    return Op::apply(expr1.get(index),expr2.get(index));
  }

 protected:
  const Expression1& expr1;
  const Expression2& expr2;
};


template<class Op, class Type, class Expression>
class BinaryScalExpression
{
 public:
  typedef Type TypeName;
  BinaryScalExpression(const Type& s, const Expression& e)
    : scalvalue(s),expr(e)
    {}

  TypeName get(const int& index) const
  {
    return Op::apply(scalvalue,expr.get(index));
  }

 protected:
  const Type& scalvalue;
  const Expression& expr;
};


template<class Op, class Expression>
class UnaryExpression
{
 public:
  typedef typename Expression::TypeName TypeName;

  UnaryExpression(const Expression& e) : expr(e)
    {}

  TypeName get(const int& index) const
  {
    return Op::apply(expr.get(index));
  }

 protected:
  const Expression& expr;
};


/*
- - - - - - - - - - - - - - - - - - -
 Arithmetic Operator Classes
- - - - - - - - - - - - - - - - - - -
*/
template<class Type>
class OpAdd
{
 public:
  static inline Type apply(Type a, Type b)
  {
    return a + b;
  }
};

template<class Type>
class OpSubtract
{
 public:
  static inline Type apply(Type a, Type b)
  {
      return a - b;
  }
};

template<class Type>
class OpMultiply
{
 public:
  static inline Type apply(Type a, Type b)
  {
    return a*b;
  }
};

template<class Type>
class OpDivide
{
 public:
  static inline Type apply(Type a, Type b)
  {
      return b/a;
  }
};

template<class Type>
class OpMinus
{
 public:
  static inline Type apply(Type a)
  {
    return -a;
  }
};


/*
```

```
- - - - - - - - - - - - - - - - - - - -
 Operators ( addition )
- - - - - - - - - - - - - - - - - - - -
*/
template<class Expression1, class Expression2>
BinaryExpression<OpAdd<typename Expression1::TypeName>,
                Expression1,Expression2>
operator +(const Expression1& e1, const Expression2& e2)
{
  typedef BinaryExpression<OpAdd<typename Expression1::TypeName>,
                          Expression1,Expression2> ExprType;
  return ExprType(e1,e2);
}


/*
- - - - - - - - - - - - - - - - - - - -
 Operators ( subtraction )
- - - - - - - - - - - - - - - - - - - -
*/
template<class Expression1, class Expression2>
BinaryExpression<OpSubtract<typenameExpression1::TypeName>,
                Expression1,Expression2>
operator -(const Expression1& e1, const Expression2& e2)
{
  typedef BinaryExpression<OpSubtract<typenameExpression1::TypeName>,
                          Expression1,Expression2> ExprType;
  return ExprType(e1,e2);
}


/*
- - - - - - - - - - - - - - - - - - - -
 Operators ( multiplication )
- - - - - - - - - - - - - - - - - - - -
*/
template<class Expression>
BinaryScalExpression<OpMultiply<double>,double,Expression>
operator *(const double& s, const Expression& e)
{
  typedef BinaryScalExpression<OpMultiply<double>,
                              double,Expression> ExprType;
  return ExprType(s,e);
}

template<class Expression>
```

```
BinaryScalExpression<OpMultiply<float>,float,Expression>
operator *(const float& s, const Expression& e)
{
  typedef BinaryScalExpression<OpMultiply<float>,
                          float,Expression> ExprType;
  return ExprType(s,e);
}

template<class Expression>
BinaryScalExpression<OpMultiply<double>,double,Expression>
operator *(const Expression& e, const double& s)
{
  typedef BinaryScalExpression<OpMultiply<double>,
                          double,Expression> ExprType;
  return ExprType(s,e);
}

template<class Expression>
BinaryScalExpression<OpMultiply<float>,float,Expression>
operator *(const Expression& e, const float& s)
{
  typedef BinaryScalExpression<OpMultiply<float>,
                          float,Expression> ExprType;
  return ExprType(s,e);
}

template<class Expression1, class Expression2>
BinaryExpression<OpMultiply<typenameExpression1::TypeName>,
                       Expression1,Expression2>
operator *(const Expression1& e1, const Expression2& e2)
{
  typedef BinaryExpression<OpMultiply<typenameExpression1::TypeName>,
                               Expression1,Expression2> ExprType;
  return ExprType(e1,e2);
}


/*
- - - - - - - - - - - - - - - - - - -
 Operators ( division )
- - - - - - - - - - - - - - - - - - -
*/
template<class Expression>
BinaryScalExpression<OpDivide<double>,double,Expression>
operator /(const Expression& e, const double& s)
{
  typedef BinaryScalExpression<OpDivide<double>,
    double,Expression> ExprType;
  return ExprType(s,e);
}

template<class Expression>
BinaryScalExpression<OpDivide<float>,float,Expression>
operator /(const Expression& e, const float& s)
{
  typedef BinaryScalExpression<OpDivide<float>,
    float,Expression> ExprType;
  return ExprType(s,e);
}

template<class Expression1, class Expression2>
BinaryExpression<OpDivide<typename Expression1::TypeName>,
                         Expression1,Expression2>
operator /(const Expression1& e1, const Expression2& e2)
{
  typedef BinaryExpression<OpDivide<typename Expression1::TypeName>,
                                   Expression1,Expression2> ExprType;
  return ExprType(e1,e2);
}


/*
- - - - - - - - - - - - - - - - - - -
 Unary Operators
- - - - - - - - - - - - - - - - - - -
*/
template<class Expression>
UnaryExpression<OpMinus<typename Expression::TypeName>,Expression>
operator -(const Expression& e)
{
  typedef UnaryExpression<OpMinus<typename Expression::TypeName>,Expression>ExprType;
  return ExprType(e);
}

#endif // __MathExpression_H__
```

Benchmark.f (FORTRAN77)

```
      parameter(length=10)
      dimension v1(length)
      dimension v2(length)
      dimension v3(length)
      dimension v4(length)
      dimension v5(length)
      dimension v6(length)

      do i=1,length
        v2(i) = 1.0*i
        v3(i) = 0.1*i
        v4(i) = 0.01*i
        v5(i) = 0.001*i
        v6(i) = 0.0001*i
      end do

      do n=1,100000
        do i=1,length
          v1(i) = v2(i) + v3(i) + v4(i) + v5(i) + v6(i)
        end do
      end do

      stop
      end
```

Benchmark.f90 (Fortran90 with array operation)

```
module vect
  type vector
     integer vlength
     real, dimension(:), pointer :: data
  end type vector

contains
  subroutine create_vector(vec,vl)
    type(vector) vec
    integer, intent(in) :: vl
    allocate(vec%data(vl))
  end subroutine create_vector
end module vect

program addvec
  use vect
  implicit none

  integer n,i
  integer vlength
  type(vector) v1,v2,v3,v4,v5,v6

  vlength = 10
  call create_vector(v1,vlength)
  call create_vector(v2,vlength)
  call create_vector(v3,vlength)
  call create_vector(v4,vlength)
  call create_vector(v5,vlength)
  call create_vector(v6,vlength)

  do i=1,vlength
     v2%data(n) = 1.0*i
     v3%data(n) = 0.1*i
     v4%data(n) = 0.01*i
     v5%data(n) = 0.001*i
     v6%data(n) = 0.0001*i
  end do

  do n=1,100000
     v1%data = v2%data + v3%data + v4%data + v5%data + v6%data
  end do

end program addvec
```