卒業論文

Distributed Aggregate with Migration

平成17年2月9日提出

指導教員　近山　隆　教授
田浦 健次朗 助教授

電子情報工学科

30388 高橋 慧

Abstract

In a grid environment, it is required to support the number changes of processors which participating in the computation. However, current programming models are not sufficient in its performance or ease of description.

We propose the *Distributed Aggregate Model* as an extension of Distributed Object Model. It handles a data structure identified with indices, like distributed array and hash table. With this system, we can write high-performance programs supporting migration.

In our system, each object can be distributed among processors as fractions, but looks like one object for a programmer. So we can write a program without worrying about data distribution. A message reaches each fraction directly, so its performance is comparable to that of the message passing model.

Acknowledgements

It is a great pleasure for me to complete my first thesis. I would like to appreciate Professor Takashi Chikayama, for his guidance and advice. Associate Professor Kenjiro Taura gave me not only much technical advice but also opportunity to utilize my interest.

Finally, I want to thank members in the Chikayama-Taura lab. I cannot forget nights spent in the room 610/622. It is a priceless experience for me to write a program together, to talk about many interesting topics. Thank you very much.

Contents

Chapter 1

Introduction

1.1 Background

Nowadays, grid computing is becoming popular. Grid computing means connecting many computers that have various environments, and solving large-scale problems with them. While the performance of ordinary personal computers is developing rapidly, these computers are frequently idle. So we can obtain large computational resources with low cost by collecting these surplus resources.

Meanwhile, we cannot always use processors in a grid environment for our computation. We must leave the processors when the owner wants to occupy them, or wants to disconnect them. Thus for a large-scale programs, it is necessary to support changes in the number of processors. In addition, the performance of these processors may vary, so dynamic load balancing is also important. Load balancing is demanded not only in grid environment, but also in general parallel computation. Putting it all together, it is required for a large scale parallel program to support dynamic changes of distribution status.

It is not, however, easy to write a program supporting distribution changes. The main difficulty lies in the distribution of data and tasks. Most parallel programs need to access data handled by other processors. When a processor joins or leaves the computation, the data distribution must be changed. Some programming models have been proposed, but they are not sufficient in their performance and ease of description. A program is very complicated in some representation models, and its performance goes down in another model.

1.2 Proposal

We propose the *Distributed Aggregate Model*, extending distributed objects. It handles data that can be identified with indices; our system can present fractions distributed among processors as one large object. Distribution changes do not affect *the whole* appearance.

Existing distributed object systems enable us to call a method in a remote object, which was defined in the same way as a serial program. Based on this, our model provides the system to call a method of multiple fraction objects with one operation. A remote method is called with a range of indices as an argument: then a method is invoked in the objects whose indices intersect with the argument indices.

In the objects definition, a programmer needs to write `split()`, `merge()` and `serialize()` methods; he/she can migrate objects or change the distribution of elements.

1.3 Contribution

Existing distributed object models can represent a parallel program as a serial object-oriented program, but it is not easy to make it support the changes in the number of processors with high performance.

In our distributed aggregate system, each object can be distributed among processors as fractions, but looks like one object to programmers. So a programmer can write a program

without worrying about its distribution. When he/she calls a method, he/she specifies the range, not a certain fraction. This enables him to write distributed programs easily regardless of data position.

Just one function call is required for fraction migration; it simplifies procedures of processor number changes, or data distribution changes. Messages are reached directly, so it makes a performance comparable to message passing model.

1.4 Organization

The rest of this paper is organized as follows. In chapter 2, we describe the existing models for parallel programming. The definition of *Distributed Aggregates* and detail of our system are given in chapter 3. We show a programming example in chapter 4, and we conclude this paper in chapter 5.

Chapter 2

Parallel Programming Models

In this section, we explain the following existing methods to write parallel programs. The points are performance and ease of description, when we write a program supporting changes of the data distribution.

- Message Passing
- Distributed Shared Memory
- Distributed Object

2.1 Message Passing Model

2.1.1 Overview

Message passing model is the most basic and widely used parallel computing model. We can hold MPI [1] up as an example. It is similar to the basic socket communication, but recipient specification and confliction handling is devised.

Processors can access only their local memory, and can call only local functions. API for the communication is `send()` and `receive()` functions. The `send()` function sends messages for indicated peer; the sent message is delivered to *receive queue* in the peer processor; when the `receive()` function is called, it takes a message from the queue, and returns it. The receive queue ensures the message reception even if many processors send messages to one processor at once.

Remote data can be accessed by sending a message to a processor holding the data. Since a programmer can define the meaning of the messages, it has high-level freedom. As message passing system provides lower primitives, we can write a high-performance program.

At the same time, it is not easy to write complicated programs with it, since the programmer must write everything. Message passing is in itself procedural, so not fit to object-oriented programming well. Messages are not received until the `receive()` function is called. Since the order of data requests is unpredictable, we have to use multi-thread style. It complicates the program. And every message flow concentrates on the receiver thread, so it is not good for readability.

In the common message passing model, it is hard to support processor number changes. The problem lies in the way to specify the peer processor.

A message must include the specification of the recipient processor. In most message passing systems, one number or name is associated with processor. Then, if a programmer wants to write a program supporting distribution changes, he/she must maintain a conversion table of data and processors on his own. When the data distribution changes, the table should be updated by sending some messages.

2.1.2 Phoenix Model

To eliminate the difficulty of supporting distribution changes, *Phoenix Model*[4] is proposed. In this model, each processor has a set of numbers called *virtual nodes*, not a single number.

There should not be a wrap of numbers among virtual nodes held by each processor.

The mapping of processors and virtual nodes can be altered dynamically; if a processor leaves a computation, another processor takes over the virtual nodes owned by the original processor. If a processor joins, a processor which have already participated divides its virtual nodes, and gives it.

We send a message with specifying one virtual node (single number). The message arrives at a processor that includes the virtual node, and the system ensures the message reaches the specified number soon or late.

Owing to this mechanism, we can fix the number of virtual nodes; a program can go on without changing the recipient even if the number of processors changes. It enables us to write a program with data distribution changes easier. The readability problem is kept as is.

2.2 Distributed Shared Memory

2.2.1 Overview

Distributed shared memory makes it possible to access remote memory by ordinary read/write. It looks that there is a large address space shared among every processors, and each can read or write these space on their own. So we can write parallel programs in the same way as multi-thread programs for a single processor.

From the outside, there seems to be a large continuous address space, but in fact this address space consists of a collection of local memories. When the system catches an access to the remote memory, it sends a request message to the corresponding processor behind. Then the remote processor receives the message, performs the requested operation, and sends back the value if needed. We cache some remote memory data in the practical system, but it is the basic idea.

It simplifies representations of parallel programs much, since we do not need to care about the meaning of the messages. Each flow looks separated, so we can write a readable program.

On the other hand, a program written with this model tends to be slow because of useless messages. Since we do not know which memory is local and which memory is remote, sometimes remote memory access occurs repeatedly, which can be prevented in the other model. To overcome this fault, programmers must take care of these things, but it is bothering.

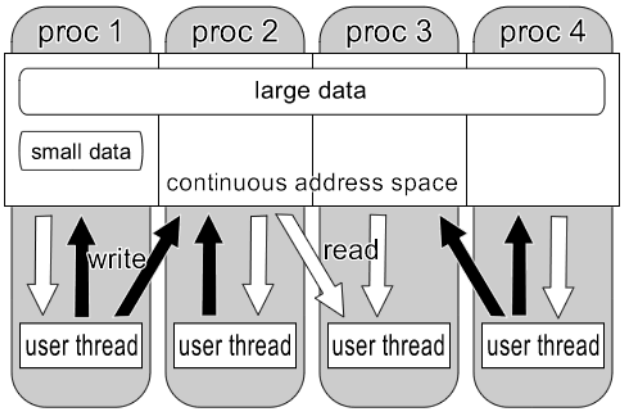

Figure 2.1. Distributed Shared Memory

Distributed shared memory is able to handle distribution changes easily. Since data are identified with virtual address space, we do not need to care about data distribution.

When a new processor joins, the existing processors divide their address space, and give them to the newcomer. We also need to divide tasks as well as data. When a processor

leaves, the rest processors take over its address space. The system perform these things automatically, so we can concentrate on the algorithm.

2.2.2 Remote Memory Access of MPI2 Model

MPI2[1] is an extended model of MPI, containing *Remote Memory Access (RMA)*. If you declare in advance, you can access remote memory with `mpi_put()` / `mpi_get()` functions.

Like distributed shared memory, remote data access can be written with ease. We can use message passing where the performance is important, so can achieve better performance. On the other hand, it is as difficult as normal MPI model to support distribution changes.

2.3 Distributed Objects

2.3.1 Overview

Distributed object model is a natural extension of object-oriented model of serial programs. In the object oriented programming, we first define some objects and write a main routine by using these objects. Data access of the object is written in methods, and access from the outside is performed by calling a method of the object. This limitation is called encapsulation, which is good for clearing up the algorithm.

In the distributed object model, a program can call a method of remote objects. It enables us to render them some tasks or to get some data. Since the method is executed in the remote processor, if each method contains a certain amount of tasks, task sharing is done automatically.

The program written with this framework tends to have a better performance than distributed shared memory, since the difference between outside and inside is obvious.

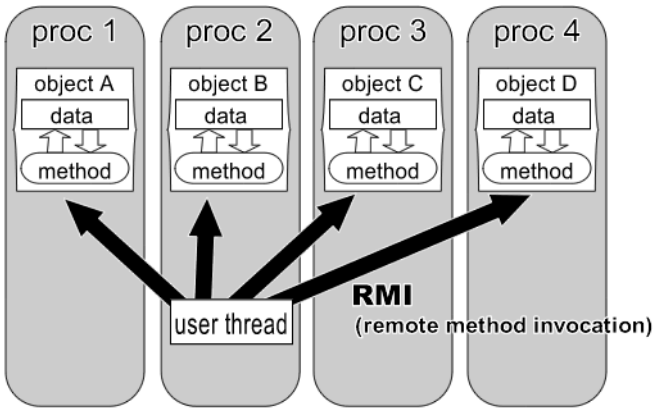

Figure 2.2. Distributed Object

Distributed object model can support distribution change in essence. When a processor joins a computation, all we have to do is to give objects to it; when some processors leave, we draw objects up and put them to another processor. If the method can be called as before, the program can go on without caring about the object movement. Since data are encapsulated and cannot be accessed from the outside, it does not matter much where the object is. Object move is called *migration* in general.

In the ordinary distributed object system, however, there are some problems with handling large data. Think of handling large data with distributed object; it is natural to divide data into many fraction objects, and put them to every processor one-by-one. If a new processor joins, we split a fraction, and migrate it to the new processor, but it is hard to write a program in this way. It is because ordinary migration is on one object basis, and it does

not support the object split. One object is treated as completely new when we want to split an object, and we cannot go on without concerning about this change. Another way is as follows: split the data into many small objects, and each processor holds multiple objects. If a new processor joins, some small fractions migrate to it. This way works well, but the performance goes down since it causes frequent remote method invocation.

2.3.2 Concurrent Aggregates

Concurrent aggregate[3] model is an extension of distributed object specialized in distribution of distributed data with migration. This system enables us to write a program supporting processor number changes without concerning about data distribution. When a programmer defines a fraction object, he/she can use a syntax that means "forward this request to the neighbor fraction." It makes the situation simple, because each fraction can conduct as a whole object.

Meanwhile, this system is not efficient for data having indices, because it does not utilize the index information. Take an example of writing distributed array with this system; the data access function may be written as follows: if a fraction contains the element corresponding the argument index, return the value; otherwise forward the request. A data request is relayed many times until reaching the fraction that has the element. It is a too heavy task to access one element of array. If we maintain a table of element-processor mappings, we can access the corresponding fraction directly almost every time.

Chapter 3

Design and Implementation

3.1 Overview

In the previous section, we analyzed existing programming models and showed problems in writing a program with distribution changes. Based on this analysis, we propose **Distributed Aggregate Framework** that supports migration, holds high performance and ease of description.

Our framework treats data that can be identified with indices. Each fraction has a set of indices which applies to holding elements, and an index set indicate the subject of remote method invocation. A programmer writes object definitions and other routines in which the instance are created. Just the fraction object definition is required, and in the main routine he/she can call a method for the whole aggregate. He/she specifies a range of indices as an argument, then the system calls a method of fraction objects whose indices intersect with the argument.

The system is built on *Phoenix Library*, which manages the routing and message passing. Owing to this, our system provides high fault tolerance since there are no single point of failure.

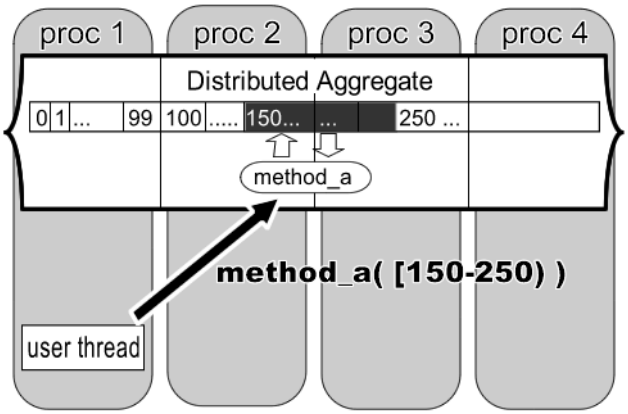

Figure 3.1. Distributed Aggregate Model

See the figure 3.1. In this example, the aggregate is distributed among four processors. Here, `method_a` is called with indices [150-250), and the method of objects in processor 2 and 3 is called.

3.2 Definition of Distributed Aggregates

In most distributed programs, data are distributed among processors. Above all, a data structure which has the elements identified with indices is frequently used. It includes distributed array and distributed hash table. We call it *Distributed Aggregates*.

In a serial program, these data structure are efficient because the address of a certain element can be calculated with ease. Likewise, if the processor number is fixed, it is easy to tell the position — or the processor — of a certain element. However, when the data distribution changes dynamically, the position is not obvious. Thus we must maintain a routing table — correspondences of elements and processors. Our system contains efficient implementation of this routing table.

3.3 APIs

This framework provides some classes for writing a distributed aggregate. With this library, a programmer can write a program with migration easier. These APIs are provided for C++ language, but the idea is indifferent to languages.

Now we show the details.

3.3.1 Index Set

Index Set is a set of integers, used to specify a data range. Each index maps one or multiple elements, and each fraction object has a set of indices corresponding to the handling data. We defined `IndexSet` interface consisting of set operation methods, and implemented it in two ways — linear list and rectangle array. You can choose one of them for each aggregate. Different aggregates can have different implementations of IndexSet.

See the figure 3.2. It shows the way to express the indices in the black square. This case, rectangular expression is better in its size.

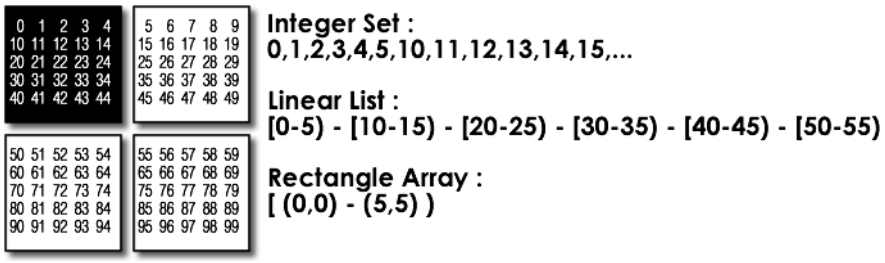

Figure 3.2. Expressions of Index Set

3.3.2 Library for Data Transfer

We provide `PackObject` and `UnpackObject` for data transfer among processors. These are classes for packing and unpacking variables. We also provide `Packable` interface, which contains `pack()` and `unpack()` methods. A programmer can pack any classes by implementing these methods on his own.

In this system, arguments of remote method are treated in the form of `PackObject` and `UnpackObject`. When a programmer call a remote method, he/she must specify `PackObject`. It has pack methods applying for most basic data type and for classes extending `Packable`. He/she packs certain arguments, and pass it to the system. The `PackObject` is transfered in the form of byte sequence, and reassembled into `UnpackObject` in the target object. The remote method receives the `UnpackObject` for an argument, and can use packed variables by calling unpack functions.

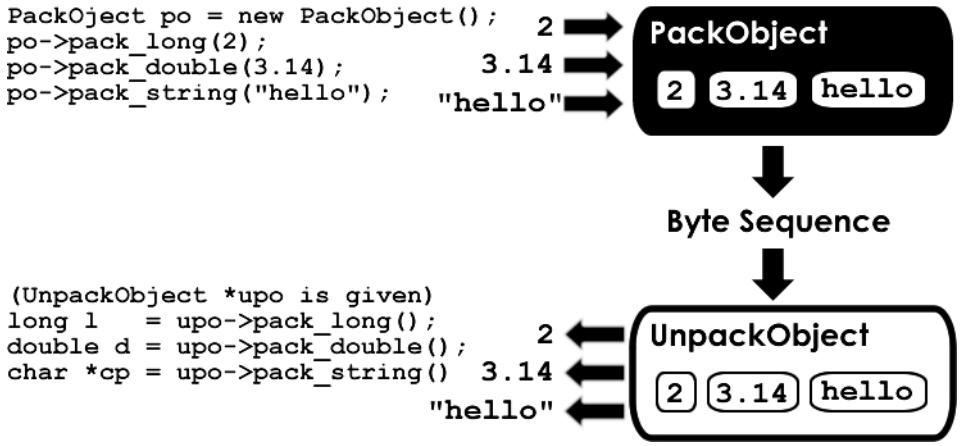

Figure 3.3. Pack / Unpack

3.3.3 Class definition

Fraction object class is defined as an extension of `DistributedAggregate`, which has `recv()`, `merge()`, `split()`, and `serialize()` as virtual methods. He/she implements these methods, and write methods he/she wants. The `send()` method is provided in order to access the whole aggregate with RMI.

Member Variables

Fraction object must have `object_id`, `own_is` and `whole_is`, which are variables of the super class `DistributedAggregate`. The `object_id` is a unique number for a whole aggregate. It must be common among fractions which consists of one aggregate. For now, a programmer determines the number. The `own_is` shows the handling index set applying the elements held by the fraction, and the programmer must keep consistency with them. When the data split, the `own_is` must also be split. The `whole_is` means the index set of whole aggregate.

Methods for RMI

A method for RMI must have `IndexSet` and `UnpackObject` as arguments. The former applies to intersection of original argument set and `own_is`; the latter contains other argument variables. A programmer should set up an order of arguments; arguments of an RMI are packed in that order, and they are unpacked in the method definition in the same way. Our system does not support return value for now, so it should have no return value. We explain the reason later.

A method for RMI must be registered in the `recv()` method and `Method::kind` enumeration. When the system receives an RMI request for the object, it just calls the `recv()` method with specifying method type. For now, the programmer needs to write a method sorting operation in the `recv()` manually.

See an instance of method definition below. The `method_a` needs one `int` and one `double` values for arguments. Then, we do some unpack operation in the beginning of the `method_a`. We also register this method in `recv()` and Method::kind enumeration.

```
class SampleAggregate {
   ...
  void method_a(IndexSet *is, UnpackObject *upo){
    int x    = upo->up_long();    // unpack argument variables
    double d = upo->up_double();  // unpack argument variables
  ...
  }
```

```
  void recv(IndexSet *is, Method::kind m, UnpackObject *upo){
    switch(m){
    case Method::_method_a:
      method_a(is, upo);
      break;
    }
  }
};
```

Methods for Migration

To enable migration, a programmer must implement these methods. The `split()` method divides the holding data, packs them into `PackObject` and returns it. The `merge()` method takes a `UnpackObject` as an argument, just merges the data with holding data. The `serialize()` is similar to `split()`, but it packs all the data it holds, and returns it. Calling this method makes the fraction empty.

The system must be informed of every changes of handling indices in order to deliver messages correctly. The changes are informed by calling the `assumeVps()` and `releaseVps()` methods. When a fraction releases some data, it must call `releaseVps()` with corresponding indices; when it receives some data, it must call `assumeVps()`. In the constructor and the destructor, they are called automatically,

Here is an example of `split()`. We split indices first, split data accordingly, pack the data and return the packed data.

```
PackObject *SampleAggregate::split(void){
  IndexSet *split_own_is = own_is->split();     // split indices
  releaseVps(split_own_ris);                    // release indices
  Data *split_data = data->separateDiff(own_is);// split data

  PackObject *po = new PackObject();
  po->p_(split_data);                           // pack the split data
  return po;
}
```

3.3.4 Instance Creation

A main routine must contain an instance of `DistributedAggregateManager`. It spawns a receiver thread automatically, which accepts remote requests. This receiver thread runs in the fraction objects, and the main routine does not access fractions direcitly.

After creating `DistributedAggregateManager`, a programmer can create a fraction objects in every processor. He/she creates a fraction object with specifying the `object_id`, handling indices, and whole indices.

We plan to write automatic fraction distribution aftertime. Then, a programmer may specify the object type and distribution pattern — for example equable distribution or concentrating distribution — then the fraction is created in every processor.

We show an example of instance creation. Here, this fraction holds indices from 0 to 100 while the whole indices is in the range from 0 to 10000. We do not access this fraction directly, so we do not need to keep a reference of the fraction in the main routine.

```
  DistributedAggregateManager *dam =
```

```
  new DistributedAggregateManager(...);

int object_id = 1;
IndexSet *whole_is  = new LinearIndexSet(0, 10000);
IndexSet *own_is = new LinearIndexSet(0, 100);
new SampleAggregate(dam, object_id, whole_is, own_is);
```

3.3.5 Remote Method Invocation

It is required to specify `object_id`, method kind, and index set for a remote method call. The index set shows the subject of operation, and method is called in every fraction object that has intersection in the indices.

Take an example: there are fraction objects with indices [0-10), [10-20) and [20-30). A method call for [5-15) is split two method call, and the first fraction receives method call request for [5-10), and the second fraction receives [10-15). This relay is done automatically.

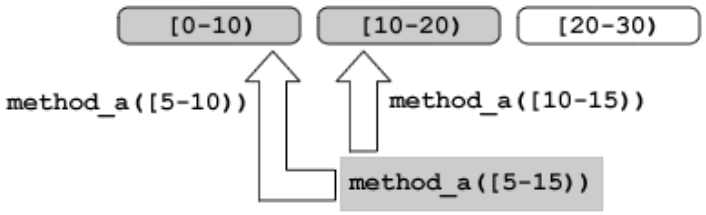

Figure 3.4. Remote Method Invocation

Arguments of a method must be stored in `PackObject`, and passed in a form of `UnpackObject`. As written before, a method for RMI has one `UnpackObject` as an argument, and it draws values afterward.

Here is an example code of RMI.

```
 // target indices
IndexSet *recipient = new LinearIndexSet(5, 15);
PackObject *po = new PackObject();
 // pack arguments
po->p_long(count);
...
 // call method_a
dam->send(object_id, recipient, Method::_method_a, po);
```

3.3.6 Migration

The fraction object migrates by calling a method of `DistributedAggregateManager`. When a processor joins a computation, it creates an empty instance of the fraction object. Then, it calls the `join()` method of `DistributedAggregateManager` with specifying object id. It sends a request message to a neighbor fraction; in the neighbor fraction, the `split()` method is called, and the data are sent back; the empty object is filled with this data, and join operation has completed. When a processor leaves, it calls `leave()` method; it calls `serialize()` method of the fraction, sends serialized data to a neighbor fraction; the

`merge()` method is called in the neighbor fraction, and leaving procedure has finished. The fraction is automatically deleted.

See the example code of migration below.

```
DistributedAggregateManager *dam = new DistributedAggregateManager(...);

int object_id = 1;
IndexSet *whole_is  = new LinearIndexSet(0, 10000);
IndexSet *own_is = new LinearIndexSet();
new SampleAggregate(dam, object_id, whole_is, own_is);

dam->joinRequest(object_id);
....

dam->leave(object_id);
```

3.4 Routing Tables

As shown before, we must maintain correspondences of elements and fractions; we call it routing table. Every method call and migration request is transfered in the form of messages; a destination of a message is determined by looking up this table. Here we explain two methods to implement the routing table: concentrating implementation and distributed implementation.

3.4.1 Concentrating Implementation

When we implement Distributed Aggregate with existing distributed objects, the most natural model is as follows.

One processor has a master object, which holds the complete routing table and manages data distribution. We can write this with normal distributed object framework with ease. Every processor can have a fraction object, which holds data. A data access must go through the master object, and we cannot access the data of fraction object directly. It keeps the consistency of routing table and the real mappings. This model is easy to implement, because the role of each object is clearly separated and the number of states are limited.

In this implementation, however, the performance is limited by message concentrations since every message runs through the master. Besides, it is not good from a standpoint of fault tolerance; if the master fails, the whole computation fails.

See the figure 3.5. Data are distributed among 4 processors, and processor 1 has the master object. The data is requested by threads in processor 1 and 3, and those requests first received by the master object, and it forwards the request to certain fraction objects.

3.4.2 Distributed Implementation

Thinking much of performance, we can conclude that the routing system must be distributed. Each processor has a routing table, and request can be sent directly to the processor holding the data.

Although this implementation is efficient, it is hard to implement it by existing distributed object framework out of nothing. The routing of our system is based on *Phoenix Library*, and a programmer can write a program with distributed routing table with our system.

Our routing tables are asynchronous inside — the routing table of the processor is sometimes different each other. The reason is as follows: if every processor always has correct routing table, it can always send message directly to the right processor. To that end, routing

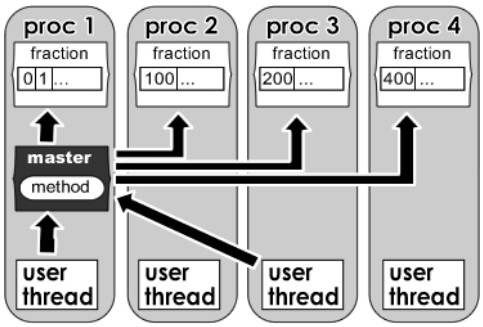

Figure 3.5. Concentrating Implementation

table of every processor must be synchronized. It is, however, not efficient. While synchronizing routing tables, all the remote access must be blocked; it makes a large overhead of distribution changes.

With asynchronous routing table, the overhead of distribution change is kept low. Sometimes the data request is received by a wrong processor, but it can be forwarded and finally reaches the right procsssor.

We mask these routing miss with the system — it has asynchronous distributed routing table inside, but the system masks the routing miss, and messages are always delivered to the correct processor. Thus, a programmer need not care about the message routing.

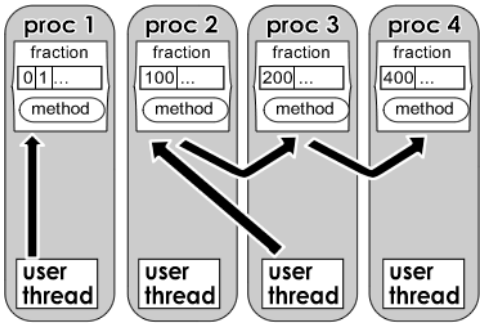

Figure 3.6. Distributed Implementation

3.5 Implementation

Our system is written in C++ language, using *Phoenix Library* for message passing. The length of the system is about 3000 lines.

3.5.1 Index Set

As refered before, we present `IndexSet` interface, and any implementations of the interface can be used for the system. We provide two implementations — linear list and array of rectangles.

Linear list expresses one continuous index set as two integers – first and last indices. Every index set can be divided into a set of continuous index set, so we can express it with set of two integers. It is suitable for the case that index set is not so fragmented.

Rectangle array is fit for data like 2 dimentional array. It expresses a rectangular indices with 4 integers – the upper-left and bottom-right coordinates.

Inadequate use of index-set structure increases the time of set operation, so we must choose suitable implementation according to the data kind. However it is trouble to write

an adequate implementation each time. We can consider binary decision diagram (BDD) for index set, which seems to express arbitral set small.

3.5.2 Routing and Relaying

Basic routing is treated by *Phoenix Library*. In the Phoenix Library, every processor has a set of numbers called *virtual nodes*, and a message is delivered to a specified virtual node. Mappings of processors and virtual nodes can change dynamically.

We map one index to one virtual node. When the distribution changes, the holding virtual nodes must be updated in the same way. The `assumeVps()` and `releaseVps()`, which are methods of `DistributedAggregate` changes the mappings of virtual nodes.

In this system, a remote method is called with specifying a set of indices. When our system receives the request, it takes one index from the indices, and sends the request to the virtual node applying to the index. A processor, which receives a request, takes the argument index set out, separate the argument set into two: the intersection with its own indices, and the rest. The specified method is called with the former indices and other arguments. The latter indices is forwarded in the same way — one index is chosen from the indices, and the request is sent to the index. Here, the other arguments are copied. The figure 3.7 shows this.

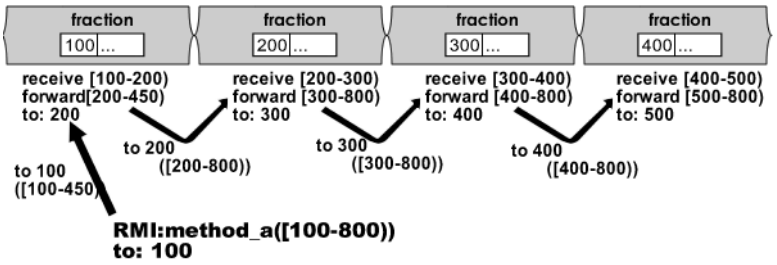

Figure 3.7. Routing and Relaying

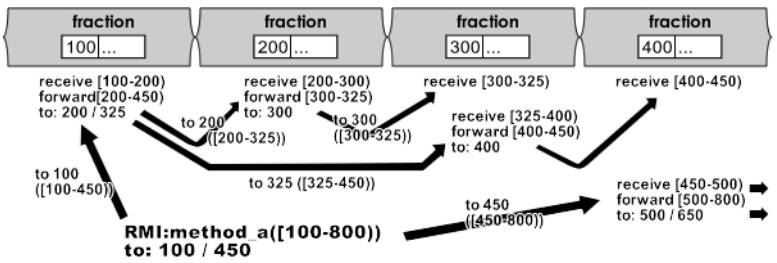

Figure 3.8. Smart Routing and Relaying

This relying method takes the time at a rate proportional to fraction number which consists indices, so the time increases when the argument indices are distributed among many processors. We also implemented another algorithm that uses tree to send a request. In this way, the time is at a rate proportional to $\log processornumber$ in the best case.

Algorithm is shown as below: we fix a threshold, and when the indices is larger than this threshold, it takes two indices from the argument indices, and sends the request to each. If the threshold value is adequate, the relaying time is shorter than the original algorithm. However, sometimes one fraction object receives the same RMI request for multiple times, and it becomes a problem in some applications.

The figure 3.8 shows this algorithm. In this figure, if the number of indices is larger than 100, the forward message is split.

3.6 Migration and Return Values

A fraction object can migrate between method invocationss. In our system, there is only one thread running on the objects, and it repeats the following operation: receives the request, executes method, and waits another request. If a method call blocks on the middle of the execution, the whole process is blocked; it may cause deadlock problems. Thus, each method should return in a short period without blocking. To fulfill this condition, we forbid stops on the middle of the method. These operations are divided, and written in multiple methods. In addition, every method invocation is asynchronous — a thread returns immediately from a remote method call. Owing to this mechanism, we can migrate an object between any method calls

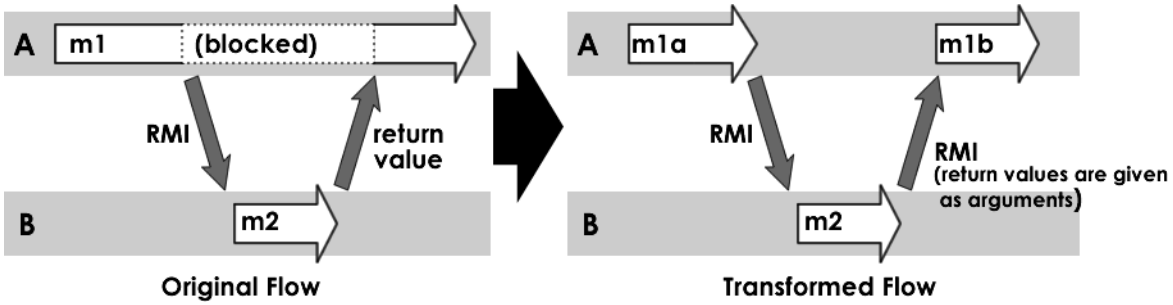

Figure 3.9. Transformation of Return Value

We show an example. See the figure 3.9: think of a state that method m_1 of object A calls another method m_2, and waiting for its return value. If we want to migrate A to another processor, it is natural to migrate not only the data of A but also the thread running on m_1. After the migration, the thread on m_1 can resume after receiving the return value of m_2.

It is, however, difficult to migrate thread to another processor; so if we want to migrate a thread, it is required that no running threads are in the object. Thus, we split the method m_1 into two – m_{1a} and m_{1b} as shown in 3.9. The m_{1a} calls m_2, and it terminates immediately. After m_2 finished, it calls m_{1b} with some arguments. Here, object A can migrate between the termiation of m_{1a} and the beginning of m_{1b} like figure 3.10.

We show a practical example of this transformation in the chapter 4.

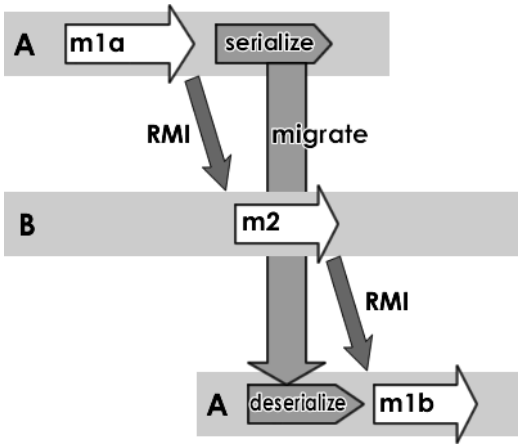

Figure 3.10. Migration

Chapter 4

Programming Examples

In the previous chapter, we explained the *Distributed Aggregate Framework*. Here we show a sample program of differential equation solution with our framework, and confirm its effectiveness.

4.1 Outline of the Problem

A large two-dimension array are given that representing some field. As the time passes, the field changes with following some difference equation. We simulate it by computations. Under some initial and border conditions, we iterate the update of every element.

When we update the values, we sum up the difference between the former value and the updated value; we call it *residual*. When the update of every element has done, we compare the residual and the threshold fixed in advance. If the residual is smaller than the threshold, we terminate the update and output the final values.

4.2 Basic Algorithm

When we solve this problem with multiple processors, it is natural to split the large array into small fractions. Each processor holds one fraction, and updates values of its buffer. As shown in figure 4.1, the left, right, top and bottom elements are required to update an element. Thus to update elements at the border, each fraction must exchange values with neighbor processors. After updating the elements, the residual values are collected. Here one processor collects all values, and compares it to the threshold. If the residual is larger, every processor repeats this operation again: if less, terminates.

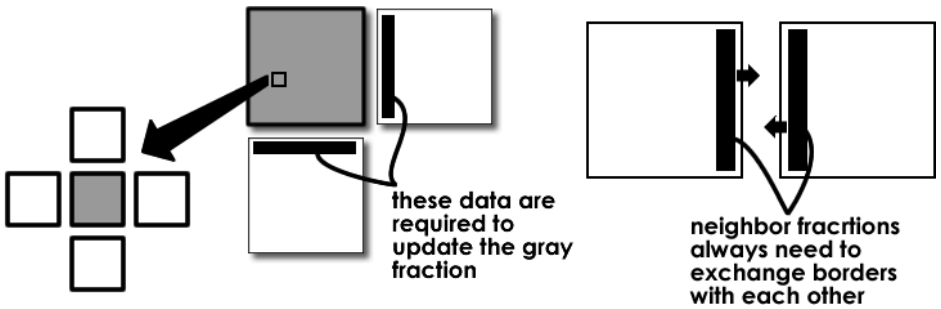

Figure 4.1. Border Exchanging

We write it in object-oriented style, and the code is written as follows. This code contains return values and multiple arguments instead of PackObject / UnpackObject, so we transform it later.

```
class SORMatrix {
  Data *own_data, border_data;

  void calc(IndexSet *recipient_is){
    // set initial conditions
    init(own_data);

    while(continue_loop){
      // get remote data.
      // send a request to the border, and it send the border data back
      border_data         = get_neighbor_data(border_index_set);

      // update own data
      my_residual         = update(own_data, border_data);

      // send residual to the index 0 by calling residual_check()
      // it returns whether residual is less than threshold.
      continue_loop       = sumup_residual(zero, own_is, my_residual);
    }

    finalize();
  }


  bool sumup_residual(IndexSet *recipient_is, IndexSet *rcvd_is,
                                              double rcvd_residual){
    // at first, ``not_arrived_is'' contains whole indices.
    not_arrived_is->remove(rcvd_is);
    residual += rcvd_residual;

    if(not_arrived_is->isEmpty(){
      // free the blocked thread
      notify();
    } else {
      while( ! not_arrived_is->isEmpty() ) {
        // wait for the arrival of residuals of other processes
        block();
      }
    }
    return (thredhold < residual);
  }
};
```

Now we transform it into methods without return value in the current system. Since the main loop in the `calc()` blocks in the `get_neighbor_data()` and `sumup_residual()` method, we split it. A variable named `state` records the status with values `border_exchanging` and `sent_residual` This status shows the "return place", and the behavior of `calc()` method changes with the status. The `s_exchanging` applies the state waiting for the return value of `get_neighbor_data()`, and `sent_residual` applys the state waiting for the return value of `sumup_residual()`.

We rewrite the border exchanging process into two methods — `send_border()` and `sendback_border()`. The `send_border()` calls `calc()` method, with right-bottom adjacent indices and right-bottom data as arguments. Figure 4.2 shows this.

When `calc()` is called in the `border_exchanging` state, the fraction forwards it to

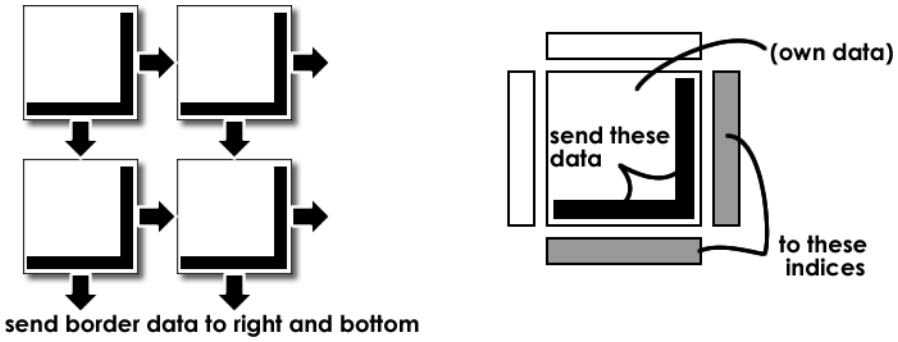

Figure 4.2. Border Exchange (send)

`sendback_border()` locally. The `sendback_border()` sends back its border to the sender of the received data; It prepares a border data applying for the recipient indices, and send it to indices of the recived data. to its right-bottom adjacent indices. This is shown in figure 4.3.

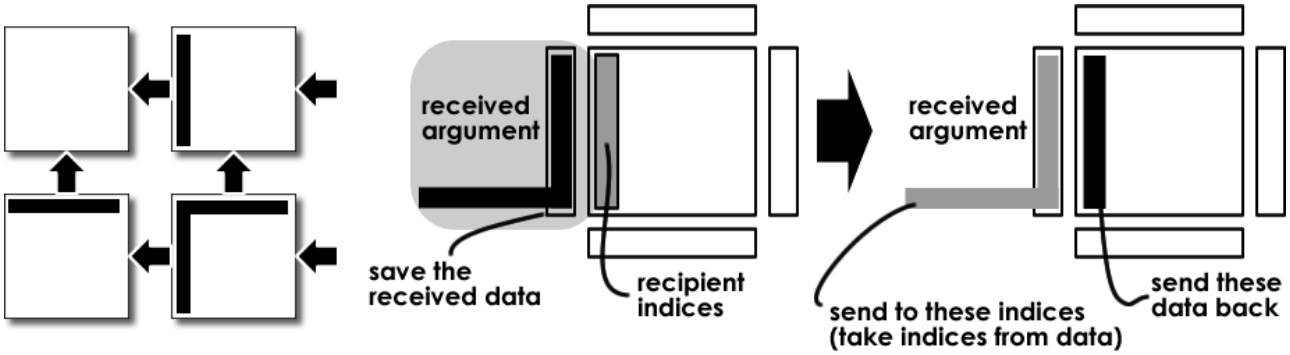

Figure 4.3. Border Exchange (sendback)

The `sumup_residual()` method also blocks unless the every residual has not arrived, so we split it in the same way. We show major methods below.

- update_buffer()
  update buffer and calculate residual value
- send_residual()
  send residual value and handling index set to index 0
- sumup_residual()
  accumulate received residual and index set. If received indices takes over whole indices, proceed to send_next()
- send_next()
  send the message "continue" to every fractions

With them, the program can be transformed as follows.

```
class SORMatrix {

  void calc(RectIndexSet *ris, UnpackObject *upo){
    switch(state){
    case sent_residual:
      send_border();
      state = border_exchanging;
```

```
      break;
    case border_exchanging:
      break;
    }

    bool ready = sendback_border(ris, upo);
    if(!ready) return;
    double my_residual = update_buffer();
    send_residual(my_residual);

    prepare_for_exchanging();
    state = sent_residual;
  }

  void sumup(RectIndexSet *ris, UnpackObject *upo){
    bool ready = sumup_residual(ris, upo);

    // if any fraction has not sent the residual, return.
    if(!ready) return;

    if(residual < threshold) send_exit();
    else                     send_next();
  }
};
```

The figure 4.4 shows the process flow of this program.

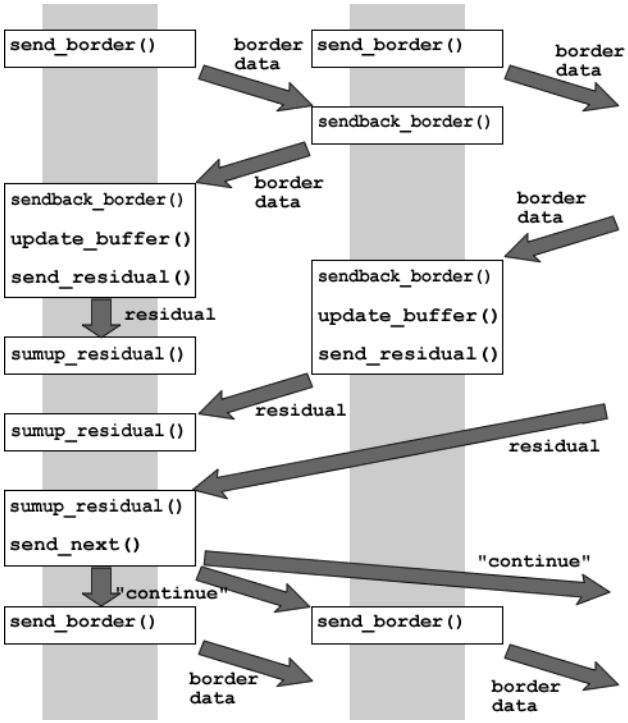

Figure 4.4. Process Flow

4.3 Migration

With our *Distributed Aggregate Framework*, an aggregate can support migration if the `split()`, `merge()`, `serialize()` methods are implemented. Basically, we implement these functions as is: The `split()` splits the own data into two; the argument data are merged with the `merge()`; In the `serialie()`, we pack whole data into `PackObject`, and returns it.

In addition to that, we must reconstruct the border data. As shown in figure 4.5, when the fraction splits the border indices change. Especially, the data on the split border line must be shared by both processors. Besides, when we merge two fractions we must take care of their iterations. If data of one fraction is newer, we cannot merge them. We must update the older one, and merge after that.

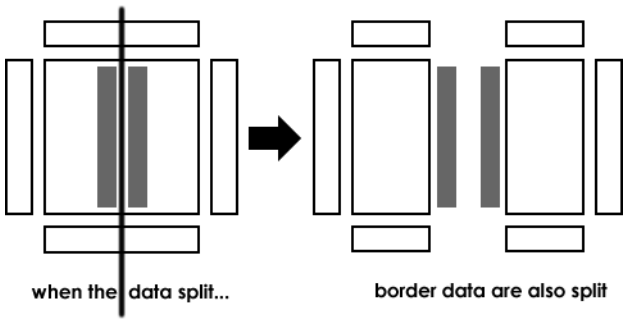

Figure 4.5. Split Operation

After writing these methods, this program can run even when the processor number changes dynaically. Since every method call is done with specifying a set of indices, the border data is delivered to right processor. We show an example of processor join in the figure 4.6. In this example the upper fraction split into left-fraction and right-farction, but the message is delivered to both processors and the computation can continue.

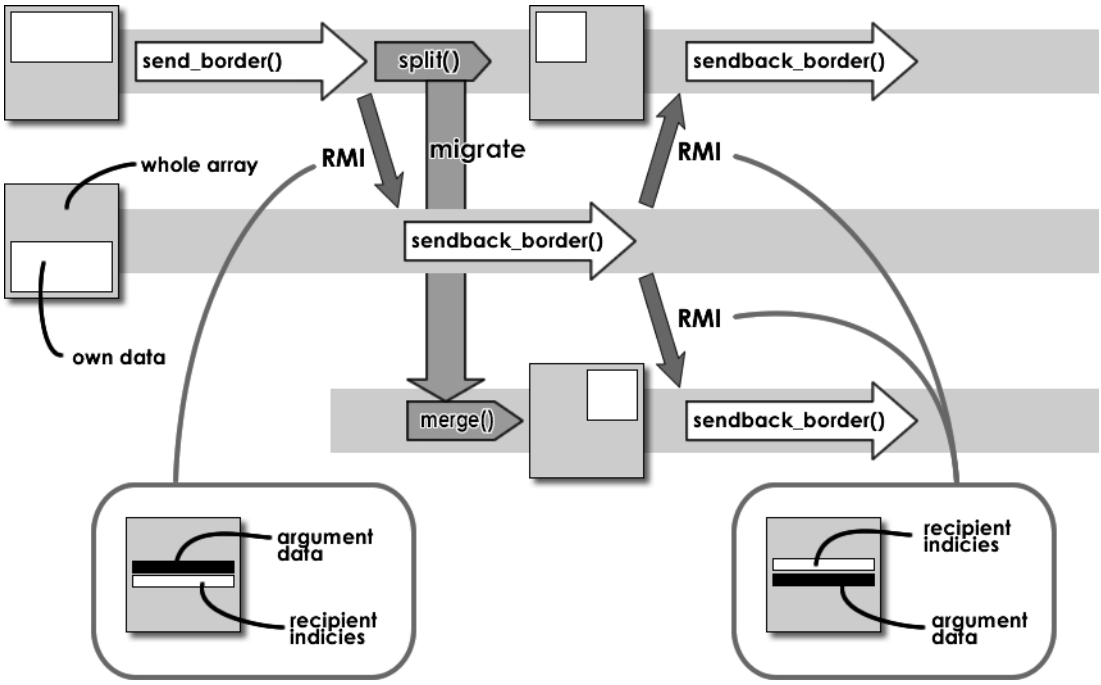

Figure 4.6. Migration

4.4 Performance

We executed this program under a computer cluster. The detail of this cluster is shown in the table 4.1. The program worked correctly when every processors joined in rapid succession.

Table 4.1. Experimental Environment

# of machines	108
MPU	Intel(R) Xeon(TM) CPU 2.40GHz
# of processors	2
main memory	2GB
Bandwidth	1Gbps

We measured the execution time of one loop, and calculated the perforamance. The performance is calculated by comparing to the time of one processor. The whole array size is set to 10000x10000, 20000x20000 and 30000x30000. We show the result in figure 4.7.

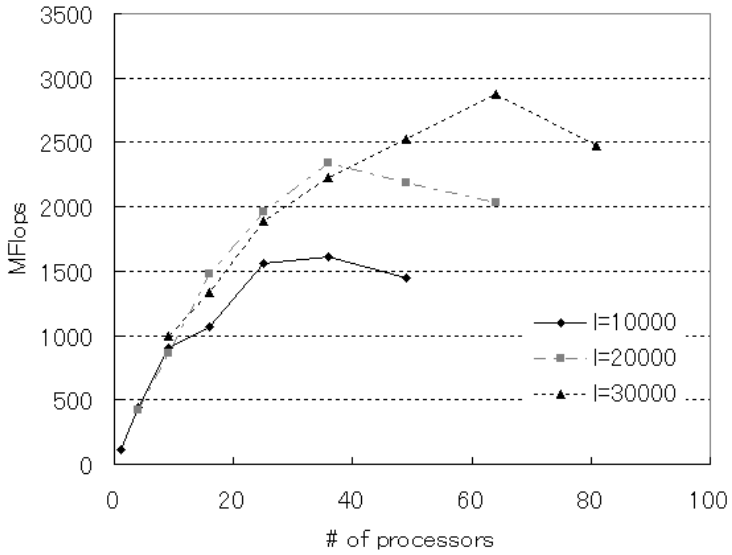

Figure 4.7. MFlops

The result was not so good or scalable. The increases of MFlops value stops at certain number of processors. We assume the residual gathering operation limits the speed, but not for sure.

We also measured the join time, and show it in figure 4.8. In this experiment, the whole array size is set to 10000x10000, and starts with 16 processors. We added 1 to 24 processors, and measured elapsed time for join. Each condition was tried twice.

The growth is not proportional to the number of joined processors, but it increases as the joined processors number up.

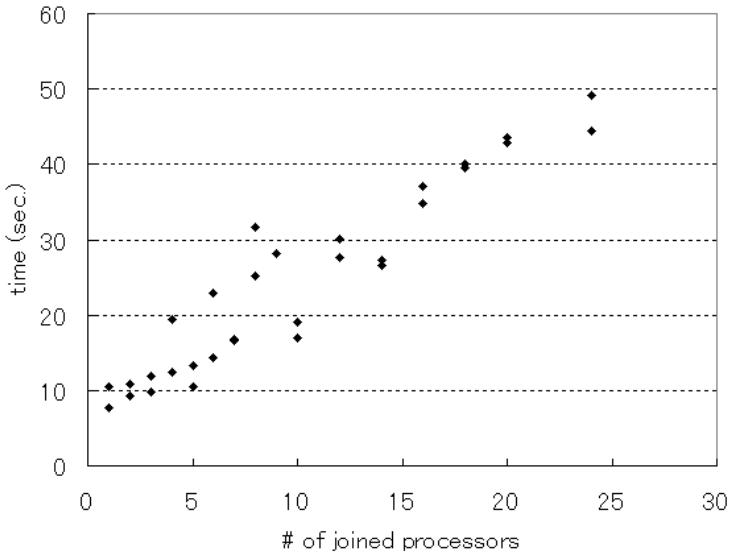

Figure 4.8. Join Time

Chapter 5

Conclusions

In this paper, we proposed *Distributed Aggregate Framework*. When we handle a data structure identified with indices, such as distributed array and distributed hash table, it is superior to existing distributed object models in its performance and ease of description. Like existing distributed objects, it consists of two basic technologies — remote method invocation (RMI) and migration. An RMI of this system takes a set of indices as an argument, and the method is called in fraction objects whose indices intersect with the argument. As we saw in section 3.3.3, a programmer only defines a fraction class. The programmer implements the `split()`, `merge()` and `serialize()`, in addition to his own methods. Then, as shown in section 3.3.6, just one function call is required for fraction migration. These two mechanisms make it easy to write a program with distribution changes.

Our system has a distributed routing table — mappings of elements and processors — in each processor, and messages for a method invocation arrive directly at the target processor. Thus, its performance is comparable to that of message passing model.

We presented a programming example of partial differential equation in chapter 4, and assured that it operates correctly even when the number of processors changes dynamically.

The effectiveness of Distributed Aggregate Model is proved thus far, but our library leaves much to be improved. Especially, the lack of return values drops the ease of description. In the future work, we are planning to support it.

References

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

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

[3] Andrew A. Chien, Vijay Karamcheti, John Plevyak, Xingbin Zbang . Concurrent Aggregates Language Report Version 2.0

(http://www-csag.ucsd.edu/papers/csag/external/ca-report.ps)

[4] Kenjiro Taura, Toshio Endo, Kenji Kaneda, and Akinori Yonezawa. Phoenix : a Parallel Programming Model for Accommodating Dynamically Joining/Leaving Resources. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2003).

[5] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick: A flexible, optimizing IDL compiler. In Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI), pages 44-56, Las Vegas, NV, June 1997.

[6] Charm++. http://charm.cs.uiuc.edu/research/charm/