An Object-Oriented Parallel Programming System for Network of Workstations

BY

Chan Wai Ming

A Thesis Presented to
The Hong Kong University of Science and Technology
In Partial Fulfillment
of the Requirements for
the Degree of Master of Philosophy
in Computer Science

Hong Kong, November 1996

Copyright © by Chan Wai Ming 1996
Authorization Page

I hereby declare that I am the sole author of the thesis.

I authorize The Hong Kong University of Science and Technology to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize The Hong Kong University of Science and Technology to reproduce the thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.
An Object-Oriented Parallel Programming System for Network of Workstations

BY

Chan Wai Ming

APPROVED:

PROF. SAMUEL T. CHANSON, SUPERVISOR

DR. MOUNIR HAMDI, CO-SUPERVISOR

PROF. ROLAND T. CHIN, HEAD OF DEPARTMENT

Department of Computer Science

25 November 1996
To God, my parents, all my teachers
and those people suffered in this world
Acknowledgments

I would like to express my sincere gratitude to my supervisors, Prof. Samuel Chanson and Dr. Mounir Hamdi, who have given valuable advice and guidance during the course of this research.

Thank Dr. Jogesh K. Muppala for evaluating my Master thesis.

Encouragement and support from my colleagues is gratefully acknowledged.

Thank my friends, Derek Ngok and Ada Yuen for helping me proof-read my writing.

Thank Ding Dong who made me give a sudden decision of applying graduation school. Without her shock news, I think I will not get into the graduate school.

Finally, I would especially like to thank Janice who has given me the most precious support during the most horrible period of my life at the final stage of my M.Phil study. Without her help, I think I cannot finish this thesis.

CHAN WAI MING

The Hong Kong University of Science and Technology
November 1996
Contents

Title Page i
Authorization Page ii
Signature Page iii
Dedication iv
Acknowledgments v
Table of Contents vi
List of Tables x
List of Figures xi
Abstract xii

Chapter 1 Introduction
  1.1 Objective of thesis ........................................... 1
  1.2 Organization of thesis ....................................... 2

Chapter 2 Object-Oriented Parallel Programming 3
  2.1 Chapter Overview ........................................... 3
  2.2 Introduction ............................................... 3
  2.3 Parallel Programming ..................................... 5
  2.4 Parallel Programming Paradigms ......................... 6
  2.5 Object-Oriented Programming ........................... 8
<table>
<thead>
<tr>
<th>Section</th>
<th>Title and Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6</td>
<td>From Object-Oriented Programming to Parallel Programming</td>
</tr>
<tr>
<td>2.7</td>
<td>Chapter Summary</td>
</tr>
<tr>
<td>3.1</td>
<td>Chapter Overview</td>
</tr>
<tr>
<td>3.2</td>
<td>Introduction</td>
</tr>
<tr>
<td>3.3</td>
<td>Programming Systems</td>
</tr>
<tr>
<td>3.4</td>
<td>Chapter Summary</td>
</tr>
<tr>
<td>3.5</td>
<td>Chapter 3 Parallel Programming System on Network of Workstations</td>
</tr>
<tr>
<td>4.1</td>
<td>Chapter Overview</td>
</tr>
<tr>
<td>4.2</td>
<td>Introduction</td>
</tr>
<tr>
<td>4.3</td>
<td>Overview of the Programming System</td>
</tr>
<tr>
<td>4.4</td>
<td>Synchronous Object Model</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Compatibility with the Object-oriented Model</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Provision of a Simple Structured Template</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Deadlock Prevention</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Encapsulation of Synchronization Code</td>
</tr>
<tr>
<td>4.4.5</td>
<td>Serializable Property</td>
</tr>
<tr>
<td>4.4.6</td>
<td>Support of Different Forms of Parallelism</td>
</tr>
<tr>
<td>4.5</td>
<td>Programming Library Framework</td>
</tr>
<tr>
<td>4.5.1</td>
<td>SyncObject Class</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Virtual Geometric Topology</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Parallel File I/O Library</td>
</tr>
<tr>
<td>4.5.4</td>
<td>SyncAppObject Class</td>
</tr>
<tr>
<td>4.6</td>
<td>Run-time System</td>
</tr>
<tr>
<td>4.7</td>
<td>Debugging Tools</td>
</tr>
<tr>
<td>4.8</td>
<td>Conclusions</td>
</tr>
<tr>
<td>4.9</td>
<td>Chapter Summary</td>
</tr>
<tr>
<td>5.1</td>
<td>Chapter Overview</td>
</tr>
</tbody>
</table>

Chapter 5 ObjectBalance: A prototype of the proposed Object-Oriented Parallel Programming System
List of Tables

6.1 Estimate values of $K_{ops}$ ........................................... 61
List of Figures

4.1 Overall architecture of the programming system .................... 22
4.2 Structure of SyncObject ........................................... 24
4.3 Control Parallelism .................................................. 28
4.4 Data Parallelism .................................................... 29
4.5 Programming Library Framework for Parallel Programming .......... 30
4.6 A virtual mesh topology ............................................ 33
4.7 Role of SyncAppObject class ....................................... 35

5.1 Architecture of ObjectBalance ...................................... 41
5.2 Synchronization Algorithm I (SA-I) ................................ 45
5.3 Synchronization Algorithm II (SA-II) .............................. 46
5.4 Usage of the Synchronization Algorithms .......................... 47

6.1 SyncAppObject Class Tree .......................................... 57
6.2 Fox's Algorithm ..................................................... 61
6.3 New Multiplication Algorithm ...................................... 62

7.1 Decompositions of the matrices .................................... 65
7.2 Experiment I ......................................................... 67
7.3 Experiment II ....................................................... 67
An Object-Oriented Parallel Programming System for Network of Workstations

BY

Chan Wai Ming

A Thesis Presented to
The Hong Kong University of Science and Technology
In Partial Fulfillment
of the Requirements for
the Degree of Master of Philosophy
in Computer Science

Hong Kong, November 1996

Abstract

As workstation and networking technologies advance, the aggregated computing power of a network of workstations (NOW) could be more than that of an expensive parallel computing system. This makes NOW an attractive alternative platform for many classes of parallel applications which were limited to expensive parallel machines in the past. Scientific computing is one of the application classes that require high computational power. However, there is still a lack of tools for developing and running parallel applications on the NOW environment. As a result, few applications exist that can take advantage of the potential of this common and inexpensive computing platform.
The aim of this thesis is to develop and implement an object-oriented parallel programming system for the NOW environment. The major contribution of this thesis is that we have considered the complete parallel programming system, covering both the human aspects and the system aspects.

In our proposed programming system, a conceptual model, named Synchronous Object Model, is provided to help programmers write parallel programs. The object model makes use of the inherent parallel nature of the object-oriented model to express parallelism. With the Synchronous Object Model, programmers can design and write their parallel programs using existing object-oriented techniques without having to think in terms of parallelism in designing the control flow. This is useful since human beings are weak in thinking parallelism. Apart from that, an object-oriented framework is provided in our programming system. The framework approach reduces the effort of parallel programming by code reuse and design reuse.

In addition, our design takes advantage of the underlying hardware architecture of the NOW environment to maximize performance by integrating multicast and load balancing supports. For load balancing, we have proposed a Dynamic Decomposition technique and a technique of mapping time complexity to real execution time. The Dynamic Decomposition technique allows a parallel program to be automatically decomposed by the programming system and allocated to processors for execution while the technique of mapping time complexity to real execution time provides useful information for improving the performance of scheduling algorithms.

The support of multicast is also a unique feature of our programming system. In the Ethernet environment, data can be sent to multiple hosts efficiently using multicast. A new matrix multiplication algorithm which makes use of multicast support is proposed and is compared with Fox's algorithm. The experimental results showed that our proposed algorithm outperforms Fox's algorithm in terms of scalability.
Chapter 1

Introduction

1.1 Objective of thesis

With the advances in workstations and networking technology, the aggregated computing power of a network of workstations (NOW) is much more than that of an expensive parallel machine. This makes NOW an attractive alternative platform for certain classes of parallel applications which were limited to parallel hardware in the past. Scientific application is one of the application classes that require high computational power. However, there is still a lack of tools for developing and running parallel applications on the NOW environment.

In this thesis, an object-oriented parallel programming system is proposed for helping programmers to design and implement parallel programs.

It is complicated to design and implement parallel programming systems. The problems involved can be classified into two main categories — the human and the system aspects. Since human being is prone to error and not used to thinking in parallel, parallel programming is inherently difficult. In our proposed programming system, a conceptual model is provided to help tackle these problems. In the system aspects, efforts are spent in taking advantage of the underlying hardware architecture of the NOW environment to maximize performance.
1.2 Organization of thesis

This thesis is organized as follows. Chapter 2 provides a historical background on the evolution and benefits of object-oriented parallel programming. A brief description of existing parallel programming systems on NOW is presented in chapter 3. The design and the implementation of our proposed programming system is described in chapters 4 and 5. Issues of how load balancing and parallel algorithms affect the performance of a parallel program are discussed in chapters 6 and 7 respectively. Finally, chapter 8 concludes the thesis with a summary of the contributions of this thesis and a brief discussion of future work.
Chapter 2

Object-Oriented Parallel Programming

2.1 Chapter Overview

In this chapter, we provide a history on the evolution and benefits of object-oriented parallel programming. In section 2.2, the reasons why parallel programming is not widely used are presented. Sections 2.3 and 2.4 describe the different components and paradigms of parallel programming respectively. Section 2.5 traces the evolution of object-oriented programming, and section 2.6 links object-oriented programming to parallel programming. Section 2.7 provides a summary of the chapter.

2.2 Introduction

Parallel computing is not a new concept. It has been around for more than two decades[15]. However, compared to sequential computing, relatively few parallel applications have been developed. People are reluctant to write parallel programs and there are two main reasons for it: the primitive software technology of software and the high cost of hardware platforms.

In the past, few people have had the opportunity to write parallel program because parallel machines were very expensive. Nowadays, the price of a parallel machine is still very high but with the advances in personal computer and workstation technology, and the popularity of local area networks, a network of personal computers or workstations
can serve as a powerful parallel machine. In fact, the aggregate computing power of
a network of workstations is often higher than that of an expensive super-computer.
Nowadays, parallel machines in the form of a network of workstations (NOW) are
available to most companies and institutions. As a result, parallel programs need not
be restricted to run on expensive parallel machines. In this thesis, we shall focus our
discussions of parallel programming and the supporting run-time system on the NOW
architecture. NOW is selected because it is commonly available and likely to increase
in popularity in the future.

Software development is a difficult task. Some people even argue that software
development is much more difficult than hardware development due to the lack of com-
prehensive theories on designing and implementing software. Parallel programming is
even more difficult than sequential programming. The software crisis has been identified
by practitioners and researchers for a long time. The complexity of parallel software is
much higher than that of sequential programs. Without introducing new software tech-
nology for parallel programming to reduce the effort of developing parallel programs,
the use of parallel computing will not take off.

Moreover, the great variety of parallel architectures further slows down the pace
of developing software tools for parallel programming. Different parallel programming
paradigms have been proposed which are tailored to different parallel hardware archi-
tectures 1.

Parallel software is expensive to develop and different parallel hardware architec-
tures are being introduced from time to time. Consequently, existing parallel software
may not be able to run on the new generations of parallel hardware without changes.
The changes required are often extensive when porting parallel applications to a new
platform. As a result, researchers are proposing to create a virtual machine. In this
approach, parallel software is written based on a virtual machine. Parallel programs
written for the virtual machine can run on any new architecture as long as the run-time
system is available on the new hardware platform. PVM is a well-known example which
follows this approach.

However, the major drawback of this approach is that the parallel programs written
for the virtual machine may not be able to exploit the strengths of the new parallel ma-

---

1 see section 2.4
chines. Also, mapping of the virtual machine to the underlying hardware architecture is a non-trivial task, and it may not be possible to map some features of the virtual machine onto some hardware architectures. Therefore, the virtual machine approach pays a performance penalty.

In the following sections, we explain why parallel programming is more difficult than sequential programming and argue that object-oriented technology is helpful in developing parallel programs.

2.3 Parallel Programming

A parallel program is composed of a set of processes running on different processors and exchanging partial results through the communication infrastructure provided by the underlying platform. Parallel programming requires the following components[17]:

- **Decomposition**
  The way in which the computation is divided into sub-pieces

- **Sequential code**
  The sequential program running on each processor.

- **Communication**
  Specification of how data are moved from processors to processors.

- **Mapping**
  The strategy of how the decomposed tasks are mapped onto the processors.

- **Synchronization**
  Points at which execution units must synchronize, and how this is accomplished.

In different types of parallel programming, the items listed above are either done by the programming system such as the compiler or by the programmer. In one extreme, the parallel programmer only has to write the program as sequential code. The parallelism is then explored by the compiler which inserts communication and synchronization code into the executable code while the run-time system is responsible for mapping the segments of code which will run concurrently onto the processors. In this
case, the work of the parallel programmer is simple. Unfortunately, efficient compilers and run-time systems able to do this is not yet available.

In another extreme, the parallel programmer performs his/her programming in traditional procedural language with communication library routines. The programmer needs to think about how to decompose the program in an efficient manner and how data are to be transferred among processors. He/She must decide when and how the processors should be synchronized. Furthermore, when the parallel program is ready for execution, the programmer needs to select a suitable number of processors for the parallel program and map each sub-program to a suitable processor manually. The last step is needed as the loading of the processors will be uneven which will lead to poor performance of the parallel program. In this case, the amount of work required of the parallel programmer is huge and tedious. In reality, most parallel programmers are working in this way currently.

Recently, many parallel programming systems have been proposed and written to simplify the tasks needed by a parallel programmer. Usually, the services provided by the systems sit between the two extremes mentioned above. In chapter 3, some representative parallel programming systems are discussed. Every programming system provides its supporting programming paradigm. The design and the services offered for parallel programming by these systems are affected by the underlying programming paradigms.

2.4 Parallel Programming Paradigms

Parallel programming paradigms can be classified into the following categories [17]:

- Explicit and Implicit Parallel Programming
  
The difference between explicit and implicit parallel programming lies in the way a parallel program is decomposed. Explicit parallel programming requires the parallel algorithms to contain explicit specifications on how the processors cooperate to solve the problem. In implicit parallel programming, the responsibility of exploring parallelism is shifted from the programmer to the compiler.

Unfortunately, implicit programming works well only on certain hardware architectures supporting fine grain parallelism and it is not suitable for hardware
platforms supporting large grain parallelism. It is very difficult for a compiler to explore the parallelism in a program since high level knowledge of the program is needed for minimizing the time spent in communication.

For explicit parallel programming, the compiler's task is straightforward since the programmer is responsible for specifying the parallelism. Existing programming systems lie somewhere between these two extremes. Special syntax constructs or function calls are usually found in languages or code libraries supporting parallel programming. These constructs are helpful to programmers for expressing parallelism in their programs and informing the compiler which parts of the program are to be executed concurrently.

- **Shared Memory and Message Passing**

  In the shared memory programming paradigm, the processes of a parallel program run on different processors but share a central pool of variables. The communication among the processes are achieved by reading and writing data to the shared variables. This paradigm is naturally suited to shared-address-space computers. Also, this paradigm is much more convenient to programmers since they do not need to explicitly specify how data are going to be transferred among the processors.

  However, this paradigm is difficult to be implemented in message-passing architectures since data have to be transferred from one processor's local memory to another if a processor accesses variables not in its local memory. If the sequence of reads and writes are not arranged in an appropriate order, there will be unnecessary network traffic among the processors.

  In message passing programming paradigm, the processes of a parallel program running on different processors do not share any address space. Each process of the parallel program performs its computation on the local memory of the processor. Processes exchange data by passing messages. The advantages of this paradigm are that the supporting run-time system is relatively easy to be implemented, and it is well suited to the message-passing hardware architectures. This paradigm can also be implemented efficiently on shared-memory architectures. In addition, it provides more control to the programmer on designing the commu-
nication part of the parallel algorithm. This is important since communication among the processes of a parallel program plays an important part in overall performance\(^2\). The main drawback of the message passing paradigm is that it is more tedious for the programmer to write programs in this paradigm.

- **Data Parallelism and Control Parallelism**

  Data Parallelism and Control Parallelism (or Task Parallelism) are concerned with how parallelism is achieved in a parallel program. In some problems, the same operations are applied to a large data set. In this case, data parallelism can be achieved by dividing the data set to run on different processors. Control parallelism refers to the simultaneous execution of different operations.

  Many problems exhibit both data parallelism and control parallelism. However, some classes of problems can be solved efficiently using only data parallelism or control parallelism alone. Some languages are specially designed to allow the user to specify data parallelism or control parallelism in a program.

### 2.5 Object-Oriented Programming

Object-Oriented Programming did not come about as a revolution. Rather, it went through an evolutionary programming process.

The evolution of programming paradigms can be traced from how people created programs\(^{16}\). The focus of the earlier programming style was on the processing level — the algorithms needed to perform the desired computation. This is known as Procedural Programming. Languages like Algol60, Pascal and C are procedural.

As time passed, the focus was shifted from the processing level to a higher level — the organization of data. This was due to the rapid increase in the size of the programs. The increase in the volume of data needed to be handled in a program forced people to program in a more organized manner. As a result, modular programming was developed. The principle became known as the "data-hiding principle". The idea is to define a set of related procedures with the data they manipulate. The set of operations and the data are often called a module unit. In modular programming, programmers

\(^2\)This will be further discussed in chapter 7.
need to decide which functions are needed to work on the data and then partition the program so that the data are hidden in the modules. Modular-2 is a language which directly supports this notion.

Programming with modules has led to the centralization of all data of the same type under the control of a type manager module. For example, a stack module contains an array of characters/integers and a set of operations (e.g. push, pop) on this array. In a program, more than one stack may be required. The stack itself can be viewed as another data-type apart from those built-in data types (e.g. char, integers). This has led to the notion of Abstract Data Type. Languages like Ada, Clu, and C++ provide ways for programmers to define types. In the program design process, programmers decide the types they want (e.g. stack, list) and provide a full set of operations for each type.

An abstract type is a black box. Once it has been defined, there is no way of changing it for a new use except redefining it. As the size of software components increases from small programs to large application programs, or even system level programs, a large number of abstract data types are needed. However, most of the abstract data types are similar. There may only be a few lines of code which are different in different abstract data types. Since humans are prone to errors, a small change in the code to create another abstract data type may lead to hours of debugging. This is one of the reasons accounting for the current software crisis in the software industry[21]. So, the focus of programming was shifted again to find a more systematic way of writing programs.

The concepts of encapsulation and inheritance seem to be a summary of what programmers needed based on their past experience in programming. Encapsulation and Inheritance are two concepts introduced in object-oriented programming. The idea of object-orientation is borrowed from the real world which is to combine state and behavior - data and operations on data - in a high level entity which is called an object. The concept of encapsulation is to make the data of an object invisible to the outside world. The data inside can only be altered through a set of methods defined for the object. This idea resembles that of modular programming. Objects with the same properties are said to belong to a class. A new class of objects can also be created by inheriting an existing class. Class hierarchies are formed as new classes are created by inheritance. Objects in the new class have all the properties of the original class
as well as the new properties added. The concepts of encapsulation and inheritance change the way programmers design their programs. Programmers need to decide which classes they need, provide a full set of operations for each class and make commonality explicit by using inheritance. The way of designing a program is quite different from the methodology used in the past which separates the design of data structures and the design of algorithms into two different processes. Languages like Smalltalk, C++ and Objective C provide language features for supporting the object-oriented programming model. Programmers who use the object-oriented languages without changing the methodology of designing programs will not benefit from the new approach. Some studies show that people without traditional knowledge of programming models can learn well and adapt to object-oriented model better. These people benefit most from object-oriented languages.

2.6 From Object-Oriented Programming to Parallel Programming

Object-Oriented Programming is naturally suitable for Parallel Programming because the nature of Object Model itself is inherently parallel. In an object model, a program is composed of objects. Each object is responsible for a specific task in the overall functionalities of the program. The interactions between objects are through message passing (which can be viewed as activating the methods defined in other objects). This model resembles the explicit parallel programming and message passing paradigms discussed in the previous section.

The parallelism that exists in the object-oriented model can be classified into the following categories:

- Inter-object parallelism
  - object groups
  - object specialization

- Intra-object parallelism
  - coding of an object is parallelized
The idea of inter-object parallelism is that each object is an active entity in the form of a process (or thread) which may run on different computers. A program is composed of these objects and data flow among objects is in the form of message passing. Although the control flow within an object is sequential, all objects execute concurrently. Thus, parallelism is achieved. Inter-object parallelism can be expressed in two forms — object groups and object specialization. This classification is based on the idea of assigning responsibilities to objects.

For object groups, a set of objects having identical functionalities form a group. Requests on the functions provided by an object group are shared by each object in the group. Since each object may run on a different processor, parallelization is not limited by the sequential nature of the object. This form of parallelism is analogous to data parallelism.

For object specialization, the functions provided by the different objects of a program are different. Parallelism is achieved by having different object taking different responsibilities in the program. This form of parallelism is analogous to control parallelism.

Intra-object parallelism is obtained by replacing the sequential control flow within an object with parallelized control flow. Intra-object parallelism is not directly related to the nature of the object model. However, the encapsulation property in the object model is important to the notion of intra-object parallelism found in existing object-oriented parallel systems such as [7] [10]. A problem in parallel programming is that people find it difficult to think in terms of parallelism. It is hard for a typical programmer to handle the synchronization and passing of data among different execution units. The notion of intra-object parallelism can alleviate this problem.

The current technique of applying intra-object parallelism is to create a pre-defined skeleton of a certain class of parallel programs. All synchronization and communication codes needed in this class of parallel programs are provided for use by an application programmer and encapsulated in the class. When the programmer writes a parallel program belonging to this class, he/she only needs to create a sub-class and write the specific code tailored to his/her problem without having to write the synchronization and communication codes. In other words, a template of a particular type of parallel program is created and the programmer is only required to fill in the missing parts in
the template. This technique has two obvious advantages:

1. A structured framework guides the programmer to write parallel programs by providing an overview of what should be in a parallel program. The job of the parallel programmers is to fill in the codes which are needed to solve their specific problems.

2. Reduces the effort of a programmer by shifting the work of writing those frequently used routines such as synchronization to the library designer.

Software reuse is known to be helpful in software development. The inheritance of object-oriented language is a nice property for constructing code libraries with high reusability. With intra-object parallelism, a programmer can concentrate on the overall design of a program without having to worry about the details of structuring the parallel code when he/she reuses those objects which encapsulate parallel codes.

Parallelism can exist in different forms in an object-oriented model. The most important point is that there are no conflicts between the forms of parallelism mentioned above. In theory, all forms of parallelism can co-exist in an object-oriented parallel system.

### 2.7 Chapter Summary

Object-oriented concepts provide nice features which may solve some fundamental problems in parallel programming. Humans are not used to thinking in parallel but the object model itself is close to real life which allows us to adopt the object model well. The most important point is that the object model is inherently parallel. If the process of translating the thoughts represented in an object model into parallelism is straightforward, parallel computation will not be limited to a few application areas such as scientific computation. It is obvious that parallel software is difficult to develop. The technique of reuse can also be applied to parallel programming. The probability of reusing existing codes which are written in object-oriented languages is much higher than that of traditional ones due to the emphasis on the structure of the object-oriented program.
Chapter 3

Parallel Programming System on Network of Workstations

3.1 Chapter Overview

This chapter gives a brief description of existing parallel programming systems on network of workstations. Section 3.2 describes some background information. In section 3.3, different parallel programming systems are discussed. Section 3.4 provides a chapter summary.

3.2 Introduction

Many parallel programming systems have been proposed, designed and implemented for different types of parallel hardware architectures. However, there is still no clear winner due to the fact that the design goal of each system is different. The different parallel programming systems were designed to address different problems found in their problem domains.

Most existing parallel software systems are not specifically tailored to the underlying hardware architecture. Rather, they create a virtual view that is convenient for the programmer to write programs. A reason is that a large scale parallel software development project may last for years. With the rapid development of parallel hardware architectures, it is not desirable to have the software tailored for a specific hardware platform. If the parallel software is written on top of a virtual machine, the whole
parallel software can easily run on other hardware platforms as long as the underlying virtual machine is ported to the other machines.

The complexity of parallel programming systems are different. In general, there are two main aspects in these systems: language aspect and run-time aspect. In one extreme, only a library of frequently used routines is provided to the programmer. In the other extreme, a whole new language is developed for parallel programming. In the latter case, the programming system requires a new compiler and sometimes even a new operating system designed to support some special functions provided.

In this chapter, a selection of existing object-oriented parallel programming systems on network of workstations is reviewed. They are chosen because the target hardware platform of the proposed programming system in this thesis is a network of workstations (see Section 2.2). Also, each reviewed system has some nice features which have influenced the design decisions of the proposed programming system in this thesis.

3.3 Programming Systems

- **DOME [2]**

  The objective of this system is to build a set of distributed objects which can be used to program heterogeneous network of computers as a single computing resource. Dome is implemented as a library of C++ classes which uses PVM for process control and communication. When an object of one of the classes is instantiated, it is automatically partitioned and allocated to processors within the distributed environment.

  In the Dome environment, objects migrate among a set of heterogeneous machines connected by heterogeneous networks to achieve load balancing. As machine load and network latency change, objects are redistributed in the network to provide good overall execution time.

  Intra-object parallelism plays an important role in the design of this system. Programming support is provided by a library of distributed objects to the parallel programmer. The objects hide the parallelism from the programmer. From the programmer's view point, parallel programs are written as if they were sequential programs. However, the construction of the object libraries is not discussed.
It will be difficult for the programmer to develop new object classes without adequate information.

By contrast, the parallel programming tools in our system are general enough for building any type of synchronous parallel application. Users can apply pre-written class library in our system in their programs, they may also create their own class library easily.

- **Parallel Object (PO)** [6] [8] [7]

The objective of this system is to make use of the advantage of object-orientation to tackle the problems of re-usability and extensibility of parallel programs. This is one of the earlier systems which advocated use of object-oriented techniques.

Parallel programs are built from the Parallel Object(PO) which has some features encapsulated in the object model. Users of the PO environment create their objects by sub-classing from PO. Therefore, the facilities provided by PO are inherited in the user's new object class. Inter-object and Intra-object parallelism are supported.

One special feature of the PO is that scheduling activities are encapsulated in an object for intra-object parallelism. The scheduling activities will be inherited by the sub-class of PO. This allows complicated scheduling code encapsulated in the PO to be made available to the programmer. Also, a special construct is provided in the system for specifying constraints which can alter the scheduling decision.

In our system, object-orientation is also applied to tackle the problems of re-usability and extensibility of parallel programs. In addition, we make use of the inherent parallel nature of object-oriented model for programmers to express parallelism.

- **Mentat** [11] [10]

The objective of this system is to provide easy development of architecture-independent parallel programs.

The programming support in this system is obtained by extending C++ with mechanisms that let the programmer indicate application classes that are computationally complex enough to warrant parallel execution. The compiler will
explore the possibility of parallelism in codes which are marked. The compiler and runtime system are responsible for the scheduling, communication and synchronization of the parallel code.

The highlight of this system is that the compiler and the run-time system take most of the responsibility for exploring parallelism in a program. Also, the Mentat language itself is a task parallel language since data parallelism is not easy to be done efficiently by the compiler without high level knowledge of the data structure. However, extension has been proposed to include support of data parallelism in the programming environment [19].

It is claimed that object-oriented supports are provided in Mentat due to the fact that the parallel language is extended from an object-oriented language. However, the effort spent on the issue of merging object model and parallel model in the design of the system is weak compared with what has been done in Parallel Object (PO)

The Mentat system relies on compiler technology to explore the parallelism of the program. The performance of the parallel program depends on how intelligent the compiler is. In our system, parallelism is specified by the programmers with the aid of a conceptual model and programming tools.

• POOL/DOOM [14]

The underlying structure of DOOM, a decentralized object-oriented machine, is a network of workstations. POOL, an object-oriented parallel language, is designed to run on DOOM. Some interesting features are found in this programming system. Users are provided with parallelism and granularity controls. The programming language is designed to support program verification. However, details of the verification part are not available in the published literature.

The program verification feature is useful since it is easy for humans to make errors in developing parallel programs. Verification may help programmers to discover their mistakes in the program design stage.

One of the common problems in writing parallel applications is deadlocks may be created unintentionally in handling communication routines. When programmers
use our object model in writing parallel programs, the deadlock problem can be avoided due to the restrictions imposed on the object model. Like programming verification, our approach can help programmers make less mistakes but without introducing the complexity of programming verification into the system.

- **XENOOPS** [20]

  XENOOPS is an eExecution ENvironment for Object-Oriented Parallel System. Although XENOOPS is implemented on transputers and not on a network of workstations, this system includes some features for load balancing not available in the previous systems discussed.

  XENOOPS provides a hierarchy of load balancing strategies, from which the user will be able to select the one most appropriate for his/her application. Also, object migration is supported for load balancing purposes. In addition, splitting and joining of objects are supported to deal with the problem of balancing large objects evenly over different nodes while objects that are too small may induce a serious overhead on the system.

  New load balancing strategies can be easily added to the our system. This provides flexibility for programmers to use tailor-made load balancing strategy for their parallel applications.

- **Ellie** [1]

  The objective of the design of Ellie is to develop an object-oriented language for architecture independent parallel programming with fine-grained objects and fine-grained parallelism as an integrated and natural part of the language.

  Ellie allows the programmer to specify programs with a large number of small processes. The compiler then tries to adapt the logical processes to the underlying hardware. The necessary information needed for load balancing is retrieved at compile time. The Ellie compiler may produce a reduced number of virtual processes suitable for the actual hardware based on some heuristics.

  The highlight of this system is that it allows the parallel programmer to avoid effort spent on load balancing. In this research work, the properties of object-oriented programming are employed for creating efficient parallel programs. One
of the properties is that an object-oriented program typically consists of a number of objects. So, objects can be viewed as measurement units for resource usage. Better estimates of resource consumption in the parallel program will lead to better strategies for load balancing decision.

Our design philosophy is not to implement a new language. This is different from the philosophy of Ellie. Our approach is to provide parallel programming support without introducing new language constructs onto the existing object-oriented language. This is further discussed in chapter 4.

3.4 Chapter Summary

A number of object-oriented parallel programming systems has been discussed in this chapter. It is obvious that almost all existing systems share a main goal which is to help programmers in developing parallel programs, but the ways to achieve the goal are different in different systems. Almost none of them consider all the aspects listed in section 2.3 since the development of some of the system components is a complicated task. For example, the design of a language or a language extension is non-trivial. The implementation of the compiler or run-time system which supports parallelism is even harder. The divergence of parallel programming paradigms also hampers the developer's effort in providing a complete system.
Chapter 4

Design of an Object-Oriented Programming System

4.1 Chapter Overview

In this chapter, we present the design of a new object-oriented programming framework. The rationale behind the design will be discussed in section 4.2. An overview of the programming framework is described in section 4.3. The two main components of the programming system, namely, Synchronous Object Model and the library framework, are described in sections 4.4 and 4.5. Sections 4.6 explains how the framework is runtime system independent. Debugging is discussed in section 4.7. Finally, a conclusion and a chapter summary are presented at the end of the chapter in sections 4.8 and 4.9 respectively.

4.2 Introduction

One of the fundamental problems of parallel programming is that the thinking process of human beings is sequential. It is easy for parallel programmers to make logical errors on handling message passing among the processes of a parallel program. Deadlocks are sometimes introduced unintentionally. One way to help is to provide a restricted programming model which guides the programmers in designing their programs.

The shared-memory parallel programming paradigm is an example of restricted programming models. It is known that writing parallel programs in shared-memory
paradigm is easier than in message-passing paradigm[22]. Communications among processes are mapped to read/write operations on global data structures which is a familiar concept for most programmers. This is an example of how a restricted programming model can help programmers in their thinking processes.

In chapter 2, it was mentioned that currently the shared-memory programming paradigm cannot be implemented efficiently on the message-passing or distributed memory hardware such as a network of workstations. This is still a subject of active research. In the shared-memory paradigm, programmers cannot specify the route of data transfers among processors even though it is important to the performance of parallel algorithms in multiprocessor systems. For the above reasons, the message-passing paradigm is the more popular programming paradigm on distributed memory hardware.

The design of our programming system was motivated by the desire to make parallel programming possible on the existing object-oriented paradigm. This design approach is different from other design approaches which mainly consisted of adding object-oriented capabilities to the existing parallel programming paradigms. The main advantage of an object-oriented extension on existing programming paradigm is higher reusability of the code. However, the inherently parallel nature of the object-oriented model is not fully employed in most current systems for solving the fundamental problem that programmers are not used to thinking in terms of parallelism.

In section 2.6, it was pointed out that object-oriented model is inherently parallel and the model fits into the message-passing paradigm. Since new generations of computer science students learn to design and program in the object-oriented way, the object-oriented model will be familiar to a typical programmer. If programmers write parallel programs based on models familiar to them, writing parallel programs will not be more difficult than writing sequential ones. Therefore, the design of our programming system is guided by the philosophy that programmers will find the inherently parallel object-oriented model easy for use in writing parallel applications.

Now the characteristics of NOW must be taken into consideration in designing the programming system. When a parallel program is executed on a multiprocessor machine, the behavior of processes on each processor is more predictable since the processors are dedicated to the jobs allocated. In the NOW environment, this is not the
case. Each workstation can be running a multitasking or multi-users operating system. The workload of each workstation will vary from time to time as people sign on and off. Static load balancing techniques will not be effective in this dynamic environment. Parallel programs do not work well on a network of workstations environment without load balancing.

Load balancing is integrated into our programming system. Programmers are forced to use the programming framework provided by the programming system for writing parallel programs. The programming framework becomes a mediator between the parallel programmer and the underlying run-time system. Programmers are guided to provide the needed information which is then used to make load balancing decisions.

The feasibility and the overhead of the implementation of the proposed programming system is an important issue. It is meaningless to have a good design on paper which is too costly or impossible to implement. A programming library approach is preferred to extending existing programming languages or designing a new language. Although the library approach would create more restrictions on the design aspects, the complexity involved in the language design and the implementation of compilers may exceed the benefits of creating or extending a programming language. Also, the library approach has the advantage that the design of the programming system would be independent of programming language issues. In other words, the programming system could be implemented in different programming languages for different needs of the users.

4.3 Overview of the Programming System

An overall structure of the proposed programming system is illustrated in figure 4.1. The programming system consists of the following components.

- A Synchronous Object Model
- An Object-oriented Programming Library Framework
- A Run-time System

The Synchronous Object Model helps programmers in their thinking process in writing parallel programs. The main structure of a parallel program is the SyncObject.
The SyncObject fits into the object-oriented model without introducing new language constructs for expressing parallelism. Parallelism and synchronizations in the parallel program are expressed through the properties of the SyncObject which is described in detail in the next section.

Apart from the Synchronous Object Model, a programming library framework is also provided for the programmer. A parallel program is composed of different parts. It is a tedious job for parallel programmers to write a lot of routine code for different parts of the parallel program. Integration of the various components into a parallel program also takes a lot of effort. Since human beings are prone to error, it would be difficult to make sure each part of the parallel program is correct and without problems when different components are put together. The programming library framework provides an organized parallel program skeleton which guides the programmer to fill in the missing parts that are unique to his/her application. Also, most of the routine code common to many parallel applications are encapsulated in the object-oriented framework for reuse. This in turn reduces the effort of parallel programming.
The last component of the programming system is the run-time system. The run-time system provides functionalities which cannot be implemented in the programming library level. In section 4.6, issues of the supporting run-time system are discussed.

4.4 Synchronous Object Model

The structure of most parallel programs consists of the cycle of computation, synchronization and communication[9]. The Synchronous Object Model is designed based on this observation. Programmers are required to use an object class, named SyncObject class, as building blocks to handle the synchronization and communication needed in the parallel programs. The SyncObject class is a programming interface of SyncObject — the conceptual Synchronous Object Model.

SyncObject is composed of two main modules:

- Computation Module
- Communication Module

The computation module contains code written by the programmer. Only two types of objects are allowed to exist inside the computation module. They are objects which do not involve other remote operations and objects which are sub-classes of object classes defined in the Synchronous Object Model.

The communication module contains two buffers — one for the incoming message and another for the outgoing message.

Multiple instances of the SyncObject class are created on different processors during execution. The instances are activated on receipt of an activation message. When the instances are activated, an execution cycle of the SyncObject starts. The execution cycle consists of two phases: a computation phase and a communication phase. The cycle starts with the computation phase followed by the communication phase.

In the computation phase, the code written by the programmer inside the computation module is executed by the processor. When the computation module of a SyncObject finishes execution, the instances of the SyncObject proceed to synchronize with one another before the communication phase starts.
In the communication phase, the data in the outgoing buffer are sent out and incoming data are stored in the incoming buffer. The order of sending and receiving is arbitrary. Sending and receiving data can be viewed as taking place at the same time. After that, the execution cycle ends. The cycle starts again when another activation message is received by the object.

![SyncObject Diagram]

Figure 4.2: Structure of SyncObject

The Synchronous Object Model possesses some useful properties:

- *Compatible with the object-oriented model*
- *Provides a simple structured template*
- *Deadlock free*
- *Encapsulates the synchronization code*
- *Serializable*
- *Allows different forms of parallelism*

In the following sections, the above properties are described in more details.

4.4.1 Compatibility with the Object-oriented Model

Instead of providing language constructs for expressing parallelism in the parallel program, a SyncObject class is used to develop parallel programs. This approach takes into the consideration the way object-oriented programming is done. In object-oriented
programming, programmers extend the functionalities of existing object classes by subclassing. And then, the class methods of the objects are added or modified for their applications. In this process, programmers are matching the functionalities of their applications to those of existing object classes in the library. Therefore, if an object is designed to execute in parallel, the SyncObject class is selected and extended by subclassing. In other words, SyncObject class serves as a building block. Parallel programmers can directly apply the functions and the properties of the SyncObject to solve their programming problems.

4.4.2 Provision of a Simple Structured Template

A problem of using parallel language for parallel programming is that the programming constructs are often low level. On the other hand, the parallel programming constructs are usually abstract. Programmers are required to learn the meaning of the syntactic constructs and how to use them for their programming task. This causes problems because the language constructs are designed to express parallelism and people are not good in thinking in terms of parallelism. Therefore, the learning curve may be steep.

The advantages of the SyncObject approach over the traditional language methods are that the programmers can use the object with well defined meaning in writing parallel programs. The SyncObject can be viewed as a skeleton of the control flow of typical parallel programs. The template serves as a reference guide on what the structure and the components should be in a parallel program. This in turn helps the programmers to organize their parallel programs. The amount of effort needed in writing the structure of a parallel program is reduced since the code and the design of the SyncObject class can be reused. Also, the behaviors and the properties of the SyncObject class are defined. Programmers think of their design in terms of objects with meaning instead of creating their programs from scratch by applying programming primitives to implement the needed functions.

To be fair, the Synchronous Object Model and the traditional parallel programming approach lie in different programming levels. In reality, the Synchronous Object Model could be used in conjunction with existing parallel programming languages.
4.4.3 Deadlock Prevention

A problem in using message passing primitives for parallel programming is that deadlock could occur. Parallel programmers must consider how to arrange the send and the receive procedure calls to avoid deadlock. This is difficult to achieve since deadlocks are not easy to detect when there are many send and receive procedure calls in a program.

In the Synchronous Object Model, this problem is solved due to a property of the communication module of the model. Since there is no send/receive procedure calls defined in the model, communication is done through two buffers in the model — the outgoing buffer and the incoming buffer. The responsibility of how to send and receive data is shifted to the run-time system and prevent the programmers from creating deadlocks in their parallel programs.

4.4.4 Encapsulation of Synchronization Code

The synchronization algorithm for distributed architectures is complicated. Also, algorithms for synchronization may vary with different communication hardware architectures.

For efficiency reasons, programmers are usually required to re-implement the synchronization part of the code when porting parallel programs to different hardware platforms. Often, knowledge in other areas, like distributed algorithms, is needed in writing efficient synchronization code. For example, there are many proposed algorithms for doing synchronization in distributed environments like a network of workstations. The parallel programmer needs to have a good understanding of the backgrounds related to synchronization in order to perform some tasks which may be regarded as trivial in sequential programming like matrix multiplication.

In the Synchronous Object Model, the task of synchronization is done by the run-time system. The advantage of this approach is that programmers can design and implement their parallel programs without having to worry about synchronization. Also, parallel programs based on the Synchronous Object Model can be ported to other hardware platforms without modifications and be able to take advantage of the new hardware platforms as long as the underlying run-time system is ported to the other hardware platforms as well.
When to do synchronization during program execution is also important. Logical errors may be introduced in the program like asking processor A to synchronize with processor B but processor B is waiting for data from processor A. In the Synchronization Object Model, programmers are required to design their objects based on the structure provided by SyncObject. The structure of SyncObject is designed to prevent some improper usage of synchronization. As a result, programmers make less mistakes in developing their parallel programs.

4.4.5 Serializable Property

The SyncObject may be implemented to exhibit the serializable property. This property is very important in designing debugging tools. Section 4.7 discusses this issue in details.

The serializable property can be implemented in the execution cycle of the SyncObject instances as described below.

In the computation phase of the execution cycle, processors execute computation modules. Assume there are $n$ processors. Let processor $P_i$ execute the code $Code_i$ of the instance $SO_i$ and take $T_i$ units of time for computation, where $i = 0 \cdots n$.

Due to the synchronous property of the SyncObject, the communication phase starts only when the computation phases of all the instances end. The order of the execution sequence of the computation modules of the instances can be arbitrary and is under program control. If we arrange that $Code_0$ starts at time $t = 0$, and $Code_i$ starts at time $t = \sum_{k=0}^{i-1} T_k$ for $i = 1, 2, \cdots, n - 1$. Then, the execution of the computation module are straightly in a sequential order. Also, in this scheme, only one processor is working at a time. As a result, a single processor can be used instead of $n$ processors to process the computation modules. In the communication phase, there is no fixed order for sending and receiving data inside the two buffers of the SyncObject. Consequently, we can do data transfer sequentially. Therefore, the serializable property can be imposed on the execution cycle of the SyncObject in a straightforward manner.

4.4.6 Support of Different Forms of Parallelism

The ability of expressing different forms of parallelism is important for a parallel programming model. The Synchronous Object Model is designed for the class of parallel
programs which requires internal synchronization as opposed to synchronization involving signals from outside of the programming system. This class of parallel programs accounts for the majority of existing scientific applications. SPMD (Single Program Multiple Data) also belongs in this category.

Two major types of parallelism — data parallelism and control parallelism can be expressed in the Synchronous Object Model easily. Data parallelism can be achieved by having multiple instances of the SyncObject on different processors processing different data but having the same code in the computation module. Control parallelism can be achieved by having instances of the SyncObject execute different code in the computation module.

![Diagram of control parallelism]

Different codes in the computation module

Figure 4.3: Control Parallelism

In figure 4.3, each circular item represents an instance. The square boxes with an alphabet represent code in the computation modules. Different letter means different instance of SyncObject contains different code. This illustrates how control parallelism is constructed. In figure 4.4, the code in the computation modules of different instances are identical. This figure illustrates how data parallelism is constructed.

### 4.5 Programming Library Framework

The conceptual model presented in the previous section offers a framework for guiding programmers in the design and structure of their parallel programs. However, the Synchronous Object Model cannot cover all the components in a parallel program such as load balancing and file I/O. In this section, an object-oriented library framework
is described. The object-oriented library framework includes library routines which handle the tasks in a parallel program (decomposition, processors mapping, load balancing and so on) to be executed on a network of workstations. The library routines are structured in template form. As a result, the programmer only needs to fill in the code which is specific to his programs without having to be concerned with the systems oriented tasks mentioned above.

Extensibility is considered in the overall design of the programming library framework. Advanced parallel programmers are allowed to add or extend the facilities provided in the library framework.

The programming interfaces are relatively simple compared with those of the other programming systems. Parallel programmers typically only need to use a few classes in the programming library listed in figure 4.5. Those complicated tasks which are known to be difficult are pre-written and encapsulated in the object classes. The following sections describe the various components of the programming library framework.

4.5.1 SyncObject Class

The SyncObject class is the realization of the Synchronous Object Model. The amount of code encapsulated inside SyncObject is more than that of the other classes. It plays a key role in the programming framework both in the conceptual level and the programming level. The internal implementation is complicated but the number of methods (programming interfaces) is small.
The programmer only needs to learn three groups of methods defined in the SyncObject class. They are concerned with the controls of instance object, definition of the computation module and manipulation of communication module.

A number of methods\(^1\) are defined in the control group but only a few of them are required for writing parallel programs. These are the methods defined for the creation of instance object. One of the methods activates the start of the execution cycle. Some are defined for more advanced process control of the instances of SyncObject and its sub-classes. Other methods are defined for querying information of the instances.

There is only one method (\texttt{mainCode}) defined for the computation module. This method is not used by the programmers but by the run-time system. For example, programmers are asked to write the main program with the procedure name \texttt{main} in the following format in C language.

\begin{verbatim}
main ()
{
    /* code */
\end{verbatim}

\(^1\)The methods are listed in the header file \texttt{SyncObject.h} in the source distribution of our prototype system
In the SyncObject class, programmers are similarly required to define the main content of the computation module. The corresponding format in Objective C language is listed below:

```objective-c
- mainCode
{
    /* code */
}
```

The last group of methods is defined for manipulating the communication module. In the synchronous object module, there are buffers located inside the SyncObject class but it is not recommended to do so in the implementation. If the buffers are located inside the SyncObject, it is likely that data will be copied into and out of the buffers frequently. Therefore, the methods are defined to store the addresses of the two data buffers (incoming data buffer and outgoing data buffer) instead of keeping the two buffers inside the object. This allows programmers to have more flexibility in buffer management. Since communication is involved, methods (see Section 4.5.2) are provided for specifying the destinations of the data stored in the outgoing buffer.

### 4.5.2 Virtual Geometric Topology

A problem which does not exist in sequential programming is to manage a set of processes on a pool of processors. Since processes need to communicate, a mechanism for naming the processes is needed. Also, the processes are arranged in some topology (like mesh) for mapping onto the input data set. As a result, programmers have to manage the naming of different processes and decompose the data domain for processes to compute on, which adds burden to the parallel programmer. Therefore, an extensible Virtual Geometric Topology is designed to help the programmer in addressing these issues.

The design of the Virtual Geometric Topology combines the two programming issues mentioned above in a unified model. The Virtual Geometric Topology exists in the form of a set of protocols and is integrated with the synchronous object. A protocol
is a set of methods. An object class which confirms to the protocol must provide
the implementation of the methods defined in the protocol. Programmers use the
methods defined in a virtual topology for handling process management and specifying
the destinations of outgoing messages.

In the proposed programming library framework, naming is done by assigning geo-
metric coordinates to every instance created at execution time. The coordinates depend
on the virtual geometric topology defined in the programming library. For example, if
the topology is defined in as a matrix, a row number and a column number are used to
represent the coordinates. If the virtual topology is a ring, a simple integer sequence can
be used. In the programming library framework, various Virtual Geometric Topologies
can be defined for different needs.

Apart from providing geometric coordinates for naming, the programming library
provides a way of specifying the destinations of outgoing messages. For example, assume
instances of the SyncObject class are created in a virtual mesh topology (see figure 4.6).
When an instance sends the data in the output buffer to another instance which is its
left neighbor, it can instruct the system to use the method sendToLeft defined in
the virtual mesh topology for sending the data. In this way, programmers can use a
higher level representation for specifying the destinations of outgoing messages. In the
traditional approaches, codes are needed to query the positions of the sender objects
and then additional codes are needed to look up the id of the object at the left side of
the sender object. Then, the message sending primitive is invoked to handle the actual
message delivery.

From the above example, it is observed that the only information the system needs
is the name of the method used for data delivery, not its contents. Therefore, a virtual
geometric topology can be defined for any topological structure.

Three protocols have been defined in our library framework. They are OBGeo-
metry, MatrixCell2D and MatrixCell3D. The OBGeometry protocol defines a minimal
set of methods as building blocks for constructing more complicated virtual geometry
topologies. The other two protocols are extensions of OBGeometry and they can be
used directly for writing parallel programs. Based on the existing protocols, program-
mers have the freedom of designing and implementing protocols for specifying other
virtual geometry topologies for their own object-oriented programming libraries.
The combination of the Synchronous Object Model and the Virtual Geometric Topology provide an infrastructure for programmers to design and implement object classes. Multiple instances of an object class can perform the same or different operations on their data. This resembles the SPMD (Single Program Multiple Data) model of computation. Also, the infrastructure makes it feasible to use the Dynamic Decomposition technique which is discussed in chapter 6.

4.5.3 Parallel File I/O Library

In data parallelism, the data set for computation is usually large. Parallelism is achieved by having the same code run on different processors which compute the sections of data allocated to them. The data are stored in a file-server and each processor only needs to read a segment of data for processing. This raises the issue of how data should be stored so that the processors can find and read the data needed efficiently. Most parallel programming systems do not provide specific support for parallel file I/O. Instead, programmers have to design their file formats and operations for concurrent reading/writing of the data files from different processors. In our programming system, a class for file I/O is available for helping programmers to handle file operations.

In our current design, the file I/O class provides methods for creating/reading/writing
a data file where the data are organized in a matrix structure. Programmers are required to specify which rows and columns of the data file are needed for read/write. The matrix structure is selected since it is easy to understand and the data in scientific computations are often in matrix form. It seems to be a basic format for most scientific parallel applications.

4.5.4 SyncAppObject Class

The SyncObject class, the virtual geometric topology and the file I/O class library are designed to form the core logics of a parallel program. However, some system oriented tasks are needed to complete the parallel program. These tasks are:

- Creation and termination of processes on the processors;
- Control of the processes distributed on the processors;
- Mapping the processes onto the processors;
- Load balancing.

The SyncAppObject class is designed to encapsulate the above tasks and provide simple interfaces for programmers to interact with the underlying run-time system. Figure 4.7 illustrates the role of SyncAppObject class in the programming library framework.

4.6 Run-time System

Although the target platform of the proposed programming system is a network of workstations, the programming system is not necessarily limited to this network architecture as the design of our programming system is based on what is helpful to parallel programmers and is very general. Since no assumption is made on the underlying hardware platform and the programming system itself is relatively high level, system designers can create efficient run-time systems for executing the parallel programs written based on the model defined in the programming system.
4.7 Debugging Tools

It is well known that programmers spend most of their time in debugging programs. A little error like an uninitialized variable may lead to hours of debugging. The availability of debugging tools to trace the execution of source code is known to be very helpful. In sequential programming, high quality debugging tools are available either from commercial vendors or in the public domain. However, the quality of parallel debuggers available is still much inferior to that developed for sequential programming in terms of user interfaces and functionalities.

The requirements of building a parallel debugger is much more complicated than those required for sequential debuggers. A simple requirement like monitoring different processes on different processors is complicated to design and implement. Also, a debugger is tightly coupled with the underlying programming system. As a result, different programming systems usually come with their native debuggers. This makes the sharing of debuggers on different programming systems very difficult.
Without suitable debugging tools, a parallel programming system will not be widely used since it will be difficult and time consuming to write correct and reliable code.

While most research work is on how to design and implement parallel debuggers running on different processors, we propose an alternative strategy — simulate the parallel environment on a sequential computer.

Simulation is often used to study the behaviors of complex and real world problems since most real world problems are difficult to control for experimentation. The techniques of simulation may also help in tracing the behavior of the parallel program. However, this technique is difficult to generalize to different types of parallel programs since the complexity of serializing some parallel programs may exceed the advantages of using simulation to trace and debug the parallel programs.

In section 4.4.5, the serialize property of SyncObject is discussed. With this property, parallel programs built from SyncObject class can be easily executed on a sequential machine. Since the Synchronous Object Model can be implemented at the library level, existing high quality sequential debuggers which are familiar to programmers can be used to trace and debug the logics of a parallel program.

## 4.8 Conclusions

This chapter has presented a proposed programming system which is designed to help programmers write parallel programs. The design goal of the proposed system is different from the other programming systems in the following ways:

- Instead of adding supports for object-oriented programming to existing parallel languages, effort is spent on unifying the object-oriented paradigm and the parallel programming paradigm by the Synchronous Object Model which serves as a conceptual tool and a programming tool (SyncObject class) for parallel programming.

- Parallelism can be expressed using the existing object-oriented model without any extension.

- Encapsulation is used for hiding all complex tasks required for parallel programming in the object classes. Thus, programmers with little experience in parallel
programming can write parallel programs easily by using the class libraries.

- Experienced programmers can use the building blocks provided to create more complex objects and extend the virtual geometric topology library.

The major limitation of the programming system that it is designed for synchronous type parallel applications which account for the majority of the scientific parallel applications. Those parallel applications that need asynchronous messaging services are not supported by the programming system.

4.9 Chapter Summary

In this chapter, we have presented the design of a new programming system, from the abstract programming model to issues in the run-time system. The programming system helps programmers to develop parallel programs in two ways. First, it provides a conceptual model called Synchronous Object Model. This model guides programmers in designing and writing their parallel programs. Due to the properties of the synchronous object model, programmers can avoid some common errors in parallel programming like deadlocks. Second, many complex programming tasks needed in parallel programming are encapsulated in the classes library and hidden from the programmers.
Chapter 5

ObjectBalance: A prototype of the proposed Object-Oriented Parallel Programming System

5.1 Chapter Overview

This chapter describes an implementation of a prototype of the proposed Object-Oriented Parallel Programming System named ObjectBalance which is running on a network of workstation. Section 5.3 describes the overall architecture of the system. Synchronization is the core part of the Synchronous Object Model described in the previous chapter. In section 5.4, we propose and present a synchronization algorithm which has been implemented in ObjectBalance. Multicast is needed in ObjectBalance because it is important to design efficient parallel algorithms. Section 5.5 discusses the reliable multicast service. We summarize the chapter in section 5.6.

5.2 Introduction

Chapter 4 describes the general structure of the proposed programming system from a programmer's view point. The details of the underlying structure of the programming system is invisible to the programmers. In order to support the Synchronous Object Model and the programming library framework described, an implementation design is
required. This layering design approach allows more flexibilities to system architects to design the underlying system. Moreover, the proposed programming system is portable across different parallel hardware platforms.

In this thesis, NOW (network of workstations) is selected as the target platform for the programming system. The prototype, named ObjectBalance, of the proposed Object-Oriented Parallel Programming System is implemented on a network of Sparc workstations connected to a 10Mbps Ethernet. In fact, the main goal of the implementation is to verify whether the idea of the programming system described in chapter 4 is implementable. Since the focus is not on the performance aspect of the system components, the performance of the ObjectBalance has not been optimized.

ObjectBalance is written in the Objective C language[13]. Programmers are required to write their parallel programs in Objective C. Objective C is chosen due to the following reasons. First, the dynamic loading and dynamic binding in the language simplify the implementation of object control in the system. Second, the language is simple and easy to learn. Finally, the language is supported in the GNU C compiler which is available in many platforms.

ObjectBalance was written and tested on Sparc workstations running SunOS 4.x. However, it can be easily ported across different Unix platforms with multicast support. In the run-time system, the implementation of distributed object control is based on a package, named libobjects, obtained from Free Software Foundation. Further, the BALANCE system, developed at HKUST, provides the underlying infrastructure for load balancing. The synchronization routines and the reliable multicast library are our own design. The overall architecture of the programming system is outlined in section 5.3 and a description of how the run-time system operates is also presented.

Section 5.4 describes an synchronization algorithm used in our programming system. The main idea of the SyncObject is to have synchronization encapsulated in the SyncObject class for programmers to use without knowing how to implement synchronization code. As discussed in chapter 4, different network architectures require different algorithms for performance reasons. Our synchronization algorithm is designed for unreliable broadcasting networks to which the Ethernet belongs.

To exploit the architecture of the Ethernet network, multicast is incorporated into the programming system. Ethernet is the most popular network technology in the
LAN environment. Since Ethernet is a broadcasting medium, it is more efficient for sending data to a set of hosts using multicast instead of sending the data repeatedly to hosts one by one. In parallel programming, it is common to have the same sets of data sent to multiple hosts. The support of multicast in the programming system allows efficient data transfers among processors. Chapter 7 further discusses the issues of incorporating the idea of multicast in parallel algorithms. In section 5.5, the problems and the solutions adopted for the implementation of multicast are presented.

5.3 System Architecture

The ObjectBalance can be divided into two main parts — the programming interface part and the underlying system. The programming interface part includes the library classes used for writing parallel applications. Moreover, it includes the Synchronous Object Model which guides parallel programmers in algorithm development. The second part of ObjectBalance is the underlying system which provides the infrastructure for implementing the programming interface part. From the programmers' view point, the underlying system is completely invisible. Programmers only need to learn about the programming interface part in order to write parallel programs. For that reason, the discussions in chapter 4 are focused on programming interface. Hence, the programming system is described from the programmers' point of view.

The underlying system is not discussed in chapter 4. Since the issues involved in its design and implementation depend on the underlying hardware platforms, we focus on the issues of the implementation of the prototype, ObjectBalance, in a LAN environment. Figure 5.1 illustrates the structure of the various components in ObjectBalance.
The underlying system of ObjectBalance is composed of the following modules:

- Communication Module
- File I/O Module
- Object Manager
- BALANCE
- Classes Library

In ObjectBalance, the data exchanged among instances of SyncObject and the internal control messages of the programming system are transferred by the Communication Module. The implementation of the SyncObject class and the Virtual Geometric Topology\(^1\) are based on the services of this Communication Module. The Communication Module is built from the UDP multicast service. The target network architecture of the

\(^1\)see sections 4.5.1 and 4.5.2
ObjectBalance is an Ethernet network which is a broadcasting medium. Multicast is the strength of this type of network architecture. Also, it is frequent to have the same data messages sent to multiple hosts in parallel computations. Multicast is the best way in handling this type of data transfer in terms of performance. Multicast services are provided through UDP in most operating systems. However, UDP is an unreliable protocol. As a result, extra work is required to create a reliable multicast service for the programming system. Multicast will be further discussed in section 5.5.

The File I/O Module provides service for organizing large data files. The file operations provided in most operating systems are often primitive. The basic operations defined are not adequate for manipulating large structural data files typically used in parallel computations. As a result, the File I/O Module is responsible for more complicated file handling such as efficient I/O when accessing a specific part of a large data file. In fact, the File I/O Module provides the infrastructure for the implementation of the FileIO class in the Programming Library Framework described in the section 4.5.3.

The Object Manager is responsible for managing the distributed objects on the workstations. The services provided by this module include object creation and termination on remote workstations, passing signals and methods to the remote objects and recording the output of the remote processes. The SyncAppObject class described in section 4.5.4 is built from the services of this module.

BALANCE[4] is a load balancing system developed at HKUST. The role of BALANCE in our programming system is to provide load balancing information. BALANCE provides the workload information of each workstation to our programming system. Any parallel program can automatically make use of workload information to distribute the objects to suitable workstations.

The last components of ObjectBalance is the Classes Library which serves as the programming interface to parallel programmers. The purpose of the Classes Library is to make use of the services from the modules described above to create the programming interface part described in chapter 4.
5.4 Synchronization

The central principle of the object model proposed in this thesis is based on synchronization. Hence, it is very important to implement an efficient synchronization scheme in the system. In order to make good use of the underlying communication hardware (i.e. Ethernet), we designed a synchronization algorithm based on UDP multicast service which is an unreliable multicast service. To the best of our knowledge, no other synchronization algorithm for large parallel computation has been designed on this type of network topology.

In this section, our new synchronization algorithm is described. The advantage of this algorithm is that it is built on top of the lightweight protocol UDP. Thus, the control messages can be transferred efficiently. Also, the use of multicast on a broadcasting medium can help reduce network traffic when one copy of message is required to be delivered to more than one host. The algorithm has been implemented and tested in ObjectBalance. In the following sections, the synchronization algorithm will be described.

5.4.1 Algorithms

Our synchronization algorithm is a distributed algorithm. Each workstation node executes the same algorithm. The pseudo code of this algorithm is listed in figures 5.2 and 5.3. The underlying communication hardware (i.e. Ethernet) has the following properties:

1. Messages sent from a node are multicasted to all nodes in the bus network;

2. A node may not receive messages sent from other nodes since the underlying network is unreliable.

The algorithm assumes that the nodes are arranged in a chain topology. Each node has an ID which indicates its position in the chain. For example, the $i$th node in the chain has the node ID $i$.

In our algorithm, each time unit is defined by the interval between two successive synchronization stages. A synchronization stage means that the nodes are performing
the synchronization algorithm. CLOCK defines the number of the past synchronization stages during the life time of a node.

The synchronization algorithm is composed of two separate algorithms listed in figures 5.2 and 5.3 respectively. The purpose of the SA-I algorithm is to synchronize with other nodes. The SA-II algorithm is applied when no further synchronization is needed. The purpose of SA-II is to end the session of synchronization. Therefore, SA-II is applied at the end of a parallel program. Figure 5.4 illustrates the use of SA-I and SA-II.

5.4.2 Explanation of Algorithm

The SA-I and SA-II algorithms are basically the same except that their conditions for termination are different. The SA-I algorithm is described first followed by the differences between these two.

The SA-I algorithm is composed of two phases. The purpose of the first phase is to ensure that all nodes are ready to synchronize and the second phase ensures that each node knows whether the other nodes are synchronized or not. There are two types of messages used in the algorithm. They are the READY message and PROCEED message. The READY message means that the sender of the READY message is in the synchronization stage and ready to proceed into the next stage. The PROCEED message means that the sender knows that all nodes are READY and all nodes can proceed into the next stage.

The algorithm works as follows. All nodes are arranged in a chain topology. Each node waits for a READY message from the preceding node. Since there is no preceding node for the first node, it goes straight into the second phase. In this phase, every node waits for a READY message from the previous node. Apart from that, the nodes in this phase help the nodes which stayed in the previous synchronization stage to proceed. This part is discussed later.

As mentioned before, the first node will not wait for the first stage and gets into second phase immediately when it is ready. After getting into the second phase, the first node sends out a READY message which informs the others that it is READY. If the second node has entered into the synchronization stage and is able to receive the message from the first node, it will go into the second phase as well. Similarly, the last
/*
* First Phase
*/
CLOCK is increased by one;

if (this is not the first node)
    wait for a READY message from preceding node;

    if (receiving READY message with time stamp < value of CLOCK)
        send PROCEED message with time stamp = the time stamp
        of the READY message;

        goto the previous step 'wait for a READY message ... ' 
    else
        goto second Phase;

/*
* Second Phase
*/
send a READY message with time stamp equal to the value of CLOCK;

if (this is not the last node)
    wait for message;

    if (time out)
        re-send the READY message;

    if (READY message with ID < my node ID)
        reset the timer;
        goto the previous step 'wait for message' ;

    if (received PROCEED message with time stamp = current value of CLOCK)
        send PROCEED message;
        exit; /* leave this synchronization phase */

else
    send PROCEED message;

Figure 5.2: Synchronization Algorithm I (SA-I)
/*
 * First Phase
 */
CLOCK is increased by one;

if ( this is not the first node )
   wait for a READY message from the preceding node;

   if (receiving READY message with time stamp < value of CLOCK)
      send PROCEED message with time stamp = the time stamp
      of the READY message
   else
      goto the previous step 'wait for a READY message ... '

   goto second Phase

/*/  
 * Second Phase
 */
send READY message with time stamp equal to the value of CLOCK;

if ( this is not the last node )

   wait for message;

   if (time out)
      if (no. of time out > MAX. TIMEOUT)
         exit; /* leave the synchronization stage */

      re-sent the READY message;

   if (received READY message with ID > my node ID)
      reset the timer ;
      goto the previous step 'wait for message' ;

   if (received PROCEED message with time stamp = current value of CLOCK)
      send PROCEED message;
      exit; /* leave the synchronization stage */

   else

      send PROCEED message;

Figure 5.3: Synchronization Algorithm II (SA-II)
Figure 5.4: Usage of the Synchronization Algorithms
node, let us define it as the Nth node, will get into the second phase when it receives the READY message from the N - 1th node.

At this point, the last node knows that all nodes are synchronized. As a result, the last node is responsible for sending out a PROCEED message and then, it leaves the synchronization stage. By the time the last node sends out the PROCEED message, the other nodes are already in the second phase and are waiting to receive the PROCEED message. When a node received a PROCEED message, it will send out a PROCEED message as well and leaves the synchronization stage. The retransmission of the PROCEED message is to reduce the probability of a node not receiving a PROCEED message because the message delivery service is unreliable.

The synchronization stage is completed when all nodes receive their PROCEED messages. However, there is a chance that some nodes may not receive any PROCEED message. Since each node sends at most one PROCEED message in the second phase and there is no re-sending from that node, it is possible for a node to miss the message. In algorithm SA-I, this problem is solved by having the nodes send PROCEED messages in the first phase. When the nodes that have missed the PROCEED message are waiting, the other nodes are performing computation and eventually will get into the next synchronization stage. At that time, the nodes will receive some READY messages from the nodes locked in the previous synchronization stage. Then, the nodes can send PROCEED messages to help the locked nodes leave the synchronization stage.

The algorithm SA-II is almost identical to SA-I except that there is an alternative condition for termination. In SA-I, when a node is in the second phase, the node will not leave the synchronization stage unless it receives a PROCEED message. On the other hand, in SA-II, a node can leave the synchronization stage if it times out. The purpose of having SA-II at the end of a synchronization session is to ensure that there is no locked node in the previous synchronization stage. This is ensured by the first phase of SA-II. If there are nodes locked in the previous synchronization stages, the nodes in the first phase of SA-II would send out PROCEED messages to the locked nodes. Eventually, all nodes will reach the first phase of SA-II. Subsequently, they will reach the second phase of SA-II. In this phase, the nodes are also waiting for the PROCEED message but they can leave the synchronization stage after time out without receiving

---

2However, this probability is very small and will be discussed in section 5.4.3
any message. Since the main purpose of SA-II is not for synchronization, it is not a problem for a node to leave without knowing that the other nodes are synchronized. The time out mechanism applied in SA-II ensures the termination of the SA-II.

The algorithm may give the impression that most of the time is spent in the second phase of SA-I and SA-II. In fact, the error handling mechanism is to make the algorithm error free. In most cases, the PROCEED messages will be received and the nodes are not likely to be locked in the synchronization stage. This will be further discussed in the following section.

5.4.3 Observations

In the design process of the synchronization algorithms, effort is spent on minimizing the number of control messages sent to the network. Since the bus network is a shared medium, the performance of the underlying network will suffer if too many messages are send to the network. This would affect other parties which share the same network. Therefore, the design of the algorithm should avoid that.

In the synchronization stage, a node could be waiting for other nodes to finish their tasks and to enter the synchronization stage. The waiting nodes are trapped in the first phase of algorithm SA-I so the first phase can also be viewed as a waiting phase. In the waiting phase, the nodes are only waiting for READY messages from their preceding nodes. They will not send any message except when there are nodes trapped in the previous synchronization stage; but this situation is rare. It is observed that only one of the nodes entering into the second phase sends out READY messages which can be viewed as polling messages. The following condition in the algorithm ensures that.

```plaintext
if (received READY message with ID > my node ID)
    reset the timer;
    goto the previous step 'wait for message';
```

49
The node that successfully sends out READY messages is the one with the largest node ID. In the second phase, each node would try to re-send a READY messages in case its READY message not received by the others. However, the action of re-sending is stopped when it receives a READY message from a node with ID greater than its own. This policy minimizes the number of READY messages floating in the network.

In the previous section, it was claimed that the probability of having a node in the second phase unable to receive at least one PROCEED message is very low. This property can be observed by the following argument. In a LAN environment, the probability of success in multicasting a message from a node to a small number of nodes is very high. The probability of success decreases when the number of recipient nodes increases. However, every node which receives the PROCEED message would send out a PROCEED message. As a result, the number of PROCEED messages floating in the network increases when the number of recipients increases, since the recipient will become a sender. This cancels the effect that the larger the number of recipient nodes, the smaller the probability of success in multicasting a message to the recipient group.

5.5 Reliable Multicast on Unreliable Network

In parallel computation, it is frequent to have data delivered to multiple processors for computation. The best way to handle this type of data transfer is to use multicast in a broadcasting network since the time required to send a data packet to a host or multiple hosts is the same. Therefore, the support for multicast is included in the implementation of ObjectBalance and this feature makes ObjectBalance different from existing parallel programming systems.

However, the development of a reliable multicast service on an unreliable network is a complicated task. It is realized that multicasting is a useful tool in many applications in the research community. Hence, reliable multicast protocols have been proposed by many researchers. An implementation described in [3] claims to provide higher throughput than any protocol that use multicast and broadcast. Parallel algorithms that make use of multicast may be better than the traditional ones which show scala-
bility problems especially on the NOW environment. A study of this issue is discussed in chapter 7.

In our programming system, a simple reliable multicast protocol is designed and implemented. Due to the complexity of the existing multicast protocols, it is more economical to design and implement a simple one for our immediate need before we gain to an implementation with good performance. Since the objective of this thesis is not on the implementation issues of multicast protocols, this approach is acceptable. The transfer rate of our multicast implementation is about 300KB/sec with hosts up to 18 workstations. The figure was measured under a real working environment. This is not very good compared with the existing implementations elsewhere. However, the performance is good enough for our experiments on testing the proposed parallel algorithms based on multicast.

5.6 Chapter Summary

In this chapter, we present the system architecture of a prototype, ObjectBalance, of the proposed programming system. Since the proposed programming system is independent of the underlying hardware architecture, some features are added to the prototype to exploit the characteristics of the target network architecture, namely, the Ethernet. A new synchronization algorithm designed for the ObjectBalance is described. The implementation of the synchronization algorithm is based on the UDP multicast. Finally, the advantages of using reliable multicast in a parallel programming system are discussed.
Chapter 6

Load Balancing

6.1 Chapter Overview

Load Balancing is a critical issue in implementing parallel applications, especially on a network of workstations where the computing power is shared by many users. In this chapter, various load balancing issues are discussed. The relationship between load balancing and parallel programming is presented in section 6.2. Section 6.3 discusses how the encapsulation of load balancing strategies helps in parallel programming. Section 6.4 describes a technique named Dynamic Decomposition which provides flexibility in designing scheduling algorithms. In section 6.5, a technique is proposed for the estimation of the elapsed time of running objects using our parallel programming model. We verify the accuracy of this technique using experiments in section 6.5.2. Finally, a chapter summary is presented in section 6.6.

6.2 Introduction

In previous chapters, we have discussed how to help programmers write parallel programs and also showed how to make use of the underlying hardware architecture to create an efficient implementation of a programming system. Another issue in getting good performance is the scheduling of processes of a parallel program to processors. In our case, the problem is how to schedule objects of a parallel program on different workstations. The scheduling decision is critical to the performance of parallel programs.
One of the goals of scheduling is to balance the load among the processors. In a parallel program, the elapsed time of a parallel program is measured by the elapsed time of the longest running process of the parallel program. Therefore, a good scheduling decision is to aim at allocating tasks to processors so that they take more or less the same time to finish.

Load balancing schemes may be classified into two categories — static and dynamic load balancing. In a static load balancing scheme, scheduling decisions are based on the computational resources required and are decided beforehand. Thus, run-time load information is not taken into consideration. This type of scheduling scheme may work well in parallel machines. In this case, the processors are initially idle and dedicated to a single parallel program. As a result, no special consideration may be needed for the workload of the processors. However, it is not the case when executing parallel programs in a network of workstations.

Workstations are usually shared by many users. Each workstation is responsible for different computational tasks from different users. Since users do not have a global view of the utilization of the workstations in their working sites, jobs are submitted to the workstations according to the users’ habit. This creates uneven workload among the workstations. Some workstations may be overloaded while others may be idle. As a result, load balancing systems are needed for networked workstations to maximize the utilization of the workstation. BALANCE[4] mentioned in this thesis is an example of a load balancing system for NOW. The load balancing issues discussed here are a little bit different from those related to parallel programming. Load balancing systems like BALANCE aim at scheduling a set of independent jobs onto the workstations. The goal of the scheduling is to maintain even distribution of the workload among the workstations. The load balancing in parallel programs is to schedule processes of a parallel program onto workstations. The goal is to maintain the elapsed time of the processes more or less the same. Therefore, a parallel programming system with load balancing features on NOW need to achieve the above two different goals. This makes the programming system very complicated. Moreover, a static load balancing scheme is no longer able to give good scheduling decisions because it does not account for the workload of workstations which fluctuates from time to time.

Thus, a dynamic load balancing scheme is more suitable to the networked worksta-
tions environment since the workload of the workstations are also taken into account in the scheduling decision. Although the additional requirements make the scheduling problem more complicated, a lot of research efforts have been spent on this challenging problem. Various theoretical models and heuristic algorithms are found in the literature to solve these problems[5, 12]. However, few parallel programming systems apply these existing results in their implementations due to some fundamental problems found in many practical situations.

In most parallel programming systems, programmers are required to handle the load balancing of their parallel programs themselves. The scheduling decision not only depends on the workload of the workstations but also by the computational resources used by the processes of a parallel program. Since the programming system does not understand the semantic of a parallel program and the exact resources requirements of the parallel program before hand, the system cannot make accurate decisions on the distribution of the parallel job onto the workstations. In fact, only the author of the program knows the program semantics, but he may not have information on the instantaneous workload on the other workstations. Furthermore, while programmers may have a general idea of the resource usages of their parallel programs, they need to learn how to apply known scheduling algorithms in order to design good scheduling schemes for their programs. As a result, programmers may give up some good but complicated scheduling algorithms and adopt some simple and intuitive ones for their programming needs. Because of this situation, some programming systems include some general heuristic load balancing schemes for programmers to use without requiring them to know the details of the load balancing strategies. However, most of these programming systems lack a general framework for convenient addition of new scheduling algorithms. In section 6.3, we propose a way of using encapsulation and inheritance to help solve these programming problems.

In typical scheduling algorithms, the number of processes of a parallel program is fixed and the scheduler simply allocates the processes to the processors. If programmers divide their programs into too many processes, the overhead of the system may lead to poor performance no matter how good the scheduler is. On the other hand, if the number of processes of a parallel program is too small and only a few workstations are used, the parallel program may not utilize the existing computation power of idle
workstations. Therefore, the decomposition of parallel programs should depend on how many workstations are selected for executing the parallel programs. A load balancing system has the capability of monitoring the loading of workstations and thus can select an appropriate number of workstations for executing a given parallel program. As a result, our programming system, ObjectBalance, supports a scheme named Dynamic Decomposition which combines the knowledge of programmers and the capability of load balancing systems to make better scheduling decisions. This is further discussed in section 6.4.

Another practical problem which has prevented widespread use of existing scheduling algorithms from practical programming systems is that it is difficult to estimate the actual execution time of the processes of a parallel program. Most computing models require knowledge of the elapsed time of the processes in their scheduling decisions. However, it is known that it is almost impossible to accurately estimate the elapsed time of the processes since each process does many different types of operations in its execution time such as network I/O, file I/O and local computations which rely on different programs. Without actually measuring the execution time of each part of the program in a running system, it is almost impossible to predict the behaviors of the program with any accuracy. Moreover, the results measured from one program cannot be used for estimating the execution of the other programs even if the programs are written for the same functions since the algorithms and the degree of optimization of the written code in different programs are different. Also, the measurement process is complicated without appropriate supports from the underlying operating system. With the advocacy of reuse in object-oriented programming, the method of measuring the resource usage of individual objects can be used to obtain the elapsed time of each object for existing scheduling algorithms. In section 6.5, a technique for estimating the elapsed time of running objects is presented.

6.3 Encapsulation of Load Balancing Strategies

Handling load balancing in a parallel program without any support from a programming system will add a burden to parallel programmers. In ObjectBalance, we provide a way for programmers to use the load balancing strategies encapsulated in the Syn-
cAppObject class. Also, the issue of extending the system to incorporate additional load balancing algorithms is considered in the design of the programming framework. Therefore, proficient programmers can add more options of load balancing strategies into the system.

The SyncAppObject class in our programming framework is responsible for the creation of object instances on the workstations. The working procedures of SyncAppObject are that the instance of a SyncAppObject class will establish a connection to BALANCE\(^1\) when the SyncAppObject is created in the program. Then, programmers can use the following method to create remote objects by supplying the class name.

\[- \text{initWithClassName} : (\text{const char} *) \text{cName};\]

Within the initWithClassName class method, the requestHosts method defined in SyncAppObject is called to obtain a list of hosts name and then the objects will be created on the hosts in the list. The load balancing strategies and corresponding scheduling code is encapsulated in the requestHosts method. Inside the requestHosts method, the workload distribution of all workstations are provided by BALANCE. Based on the workload informations, a set of workstation is selected.

Under normal situations, parallel programmers can use a SyncAppObject class or its sub-classes without the need to care about the requestHosts method. This is because each class in the tree classes of SyncAppObject encapsulates a scheduling algorithms within it (see figure 6.1). Programmers can select the scheduling algorithm suitable for their needs. Then, the programming system is responsible for distributing the objects to workstations according to the scheduling algorithm encapsulated in SyncAppObject.

Furthermore, the programming system can be enhanced by adding new scheduling algorithms. Scheduling algorithms can be created as a new class by sub-classing the SyncAppObject and then overriding the requestHosts method. Since the necessary code for communicating with the programming system is also inherited in the new class, the scheduling algorithm is implemented without adding code for its integration with the programming system. Therefore, it is much easier to add new scheduling schemes into the programming system. In particular, the programming framework proposed in

\(^1\)see section 5.3
the system can act as a bridge for combining the efforts of algorithm researchers and the parallel programmers.

6.4 Dynamic Decomposition of Parallel Program

One additional problem of parallel programming on NOW is to determine how many processes a parallel program should be decomposed. In a dedicated parallel computer, the question can be simply solved by using the number of processors available for the parallel program. In a NOW environment, this question is more difficult to be solved because computational resources are shared so that the workload of each workstation is dynamically fluctuating. As a result, the best number of workstations needed by a parallel job can only be determined during run-time. If the number of processes of a parallel program is greater than the number of workstations available, some workstations may be assigned more than one process. Uneven distribution of jobs may result. Also, each process of the parallel program incurs an amount of system overhead. Thus, if the processes assigned to the same workstation can be combined to just one pro-
cess, the overhead due to process management could be reduced. On the other hand, if the number of processes is smaller than the number of workstations available, the computational resources may not be utilized efficiently.

The ObjectBalance provides a scheme named Dynamic Decomposition as a solution to the above problems. In this scheme, programmers are responsible for informing the system how to decompose their parallel programs, and the programming system would decompose the parallel programs according to the workload distribution at moment execution time. This scheme provides more flexibility for scheduling decisions.

Dynamic Decomposition is achieved through the Virtual Geometric Topology introduced in the programming system (see Section 4.5.2). The Virtual Geometric Topology not only helps programmers in organizing the parallel programs but also provides a method for the system to decompose the system. For example, suppose a program is written as a virtual mesh topology. The mesh is defined as a $m \times n$ grid. A programmer can write his/her parallel program with the variables $m$ and $n$ as the size of the mesh. When the parallel program is executed, the system will select a suitable number of workstations based on the scheduling decision from the SyncAppObject or its subclasses which is selected by the programmer. Then, the system assigns the values of $m$ and $n$ to the parallel program based on the number of workstations available and also assigns the corresponding coordinates of the virtual geometry to the instance objects of the parallel program.

6.5 Techniques for Converting Time Complexity to Real Computation Time

The elapsed times of the processes of a parallel program are useful information for scheduling algorithms. However, this information is difficult to obtain accurately. Profiling can be used to measure the elapsed times. However, the technique is limited to those programs that are frequently used.

In object-oriented programming paradigms, the profiling technique may be useful for estimating the elapsed times of a parallel program composed of objects. Reuse is highly advocated in object-oriented programming. Programmers are encouraged to develop their programs by using the objects in exiting classes. The class library can be designed
in a way that it not only contains the source code of each object but also the runtime information of the objects. Hence, the elapsed time of the program built from the objects with history may be better estimated. In object-oriented parallel programming, some classes are frequently used as building blocks and these classes may be consuming most of the computational resources. For example, matrix manipulations are frequently used in scientific parallel programs. The runtime information of parallelized objects for matrix operations can be kept in a programming library and the information stored in the system can be used for estimating the parallel programs built from these objects.

In ObjectBalance, objects, which are responsible for parallel computation, are inherited from the SyncObject. The code for measuring the runtime behaviors are encapsulated in the SyncObject classes. This information may be kept in a database for future use. When the objects are used again, information such as elapsed time which is kept in the system’s database can be used by the programming system for scheduling decisions.

6.5.1 Polynomial Time to Real Execution Time

After recording down the runtime behaviors of the objects, we propose a technique to make use of the information for estimating the elapsed time of an object during execution. The method is based on the time complexity model.

Let \( T \) be the elapsed time of an executable object with time complexity \( O(n^k) \). It can be expressed as follows:

\[
T = a_k n^k + a_{k-1} n^{k-1} + a_{k-2} n^{k-2} + \cdots + a_0
\]

From the above expression, we can derive the following equations when \( n \) is sufficiently large.

\[
\frac{T}{n^k} = a_k + \frac{a_{k-1}}{n} + \frac{a_{k-2}}{n^2} + \cdots + \frac{a_0}{n^k}
\]

\[
\frac{T}{n^k} = A \left( a_k + \frac{a_{k-1}}{n} + \frac{a_{k-2}}{n^2} + \cdots + \frac{a_0}{n^k} \right)
\]

\[
\frac{T}{n^k} \approx A
\]

\[
T \approx An^k
\]
where \( A \) is the largest constant among \( a_i, \; i = 0 \cdots k \).

Equation 6.1 shows the relationship between the elapsed time of executable objects and the input data size. When the elapsed time of an object and its input data size is known, we can estimate the constant \( A \). By applying equation 6.2, the elapsed time of the same object can be estimated with different input data sizes. We define a constant \( K_{ops} \) where

\[
A = \frac{1}{1000K_{ops}}
\]

Then,

\[
K_{ops} = \frac{n^k}{1000T} \quad \text{and} \quad T = \frac{n^k}{1000K_{ops}} \quad (6.3)
\]

\( n^k \) can be viewed as the number of operations needed in the algorithm and \( K_{ops} \) can be viewed as kilo operations per second. Equation 6.3 is used for determining the value of \( K_{ops} \) and the elapsed time of an object with given input data size and the order of the complexity of the algorithm in the object. This estimation is extremely useful for the scheduling algorithms in order to make better scheduling decision.

### 6.5.2 Experiments

Equations 6.1 and 6.2 only give us an approximation of the values of \( A \) and \( T \). The accuracy of the estimation depends on the value \( n \). An experiment was performed to examine the accuracy of these two equations. Two different parallel matrix multiplication algorithms were implemented on ObjectBalance. One is Fox's algorithm and the other is designed with the use of multicast\(^2\). The time complexities of both algorithms are \( O(n^3) \). These two parallel programs run on a network of Sparc10 workstations connected by a 10Mbps Ethernet. The machines used for the experiments were dedicated to the parallel programs and operated in a controlled environment.

The elapsed time of the instance objects of these two matrix multiplication programs were measured. The data points obtained from the experiments were divided equally into two sets. One set of data is used for estimating the constant \( K_{ops} \) and the other set is used for testing purposes. The division was done randomly. Table 6.1 shows the estimates of \( K_{ops} \).

\(^2\)These two algorithms are further discussed in chapter 7.
Based on the values of $K_{ops}$, the elapsed times of the objects with different input data size were calculated and the results were compared with the elapsed time measured in the experiment. The comparisons are shown in figures 6.2 and 6.3. From the two graphs, it is observed that the estimated elapsed time is very close to the actual measured values. The mean errors are 1.8083\% and 4.1566\% for Fox’s algorithm and the new multiplication algorithm with multicast respectively.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>$K_{ops}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fox’s Algorithm</td>
<td>167.4832</td>
</tr>
<tr>
<td>Multicast Multiplication Algorithm</td>
<td>163.7376</td>
</tr>
</tbody>
</table>

Table 6.1: Estimate values of $K_{ops}$

![Results of Fox's Algorithm](image)

Figure 6.2: Fox’s Algorithm
6.6 Chapter Summary

Problems involved in the load balancing and processes scheduling issues in a parallel programming environment has been presented. We then proposed some techniques to address those problems. The idea of encapsulation in load balancing strategies reduces the parallel programmer's effort in writing load balancing code. Our programming framework provides programming facilities for adding new scheduling algorithms to the programming system. Also, we have proposed a Dynamic Decomposition technique which is helpful to make better scheduling decisions. Finally, a technique for estimating the elapsed time of running objects was presented.
Chapter 7

Parallel Algorithms for Network of Workstations

7.1 Chapter Overview

In this chapter, we discuss parallel algorithms for the NOW environment. Section 7.2 describes background information of parallel algorithms. A new matrix multiplication algorithm is presented in section 7.3. In section 7.4, an experiment of comparing the new algorithm to Fox's multiplication algorithm is presented. Finally, a chapter summary in section 7.5 is presented at the end of this chapter.

7.2 Introduction

Parallel algorithms not only consider the computation and memory usage of a program but also the traffic in data exchanges which is not needed in sequential algorithms. The variety of parallel architectures has created the need for designing different algorithms to solve a single problem. It is because different communication networks affects the way of handling data exchanges.

There are many algorithms designed for parallel machines with mesh, ring and hypercube communication topologies but only a few for NOW. This phenomenon is easy to explain. The use of networked workstations as a parallel machine is a recent area of research. Also, high level programming support for using hardware multicast is not included in the operating system in the past so the advantages of the underlying archi-
tectures cannot be exploited. Without programming support, the design of algorithms based on the properties of a shared bus network is not useful. As a result, most parallel programs written for NOW employ the same parallel algorithms designed for other parallel architectures. Thus, the performance and scalability of these parallel programs are rather poor.

Since multicast is included in ObjectBalance, we can design algorithms which make use of the underlying communication hardware and compare their performance with parallel algorithms designed for other parallel hardware. In the following sections, matrix multiplication is used as an example for showing the advantages of using multicast in parallel algorithm design for a network of workstations. Matrix multiplication is selected because of its fundamental importance in many scientific applications.

### 7.3 Matrix Multiplication

In order to take advantage of the broadcasting medium in the NOW environment, we have designed a new matrix multiplication algorithm. This new algorithm is simple compared with existing matrix multiplication algorithms designed for other parallel architectures.

Let $A$ and $B$ be two given matrices, and $C$ be the product of $A$ and $B$. There are $m$ processors available for computation. Denote $p_i$ where $i = 1, \cdots, m$. Matrices $A$ and $B$ are decomposed in strip format and each matrix strip is allocated to different processors. Figure 7.1 illustrates the decomposition.

Matrices $A$ and $B$ are decomposed into $m$ strips. The strips $A_i$ and $B_i$, for $i = 1, 2, \cdots, m$, are located on processor $p_i$. The pseudo code of the parallel multiplication algorithm is listed below:

```plaintext
for p = 1 to m;
    if (p == myId)
        send A(p) to all other processor;
        C(p,p) = A(p) * B(p);
    else
        receive A(p) from sender;
        C(p,myId) = A(p) * B(myId);
```

64
Figure 7.1: Decompositions of the matrices
endfor

The parallel algorithm consists of \( m \) phases. The value \( m \) depends on the number of processors available. Each processor is given an id. In the \( p \)th phase, the processor with id equal to \( p \) is responsible for sending the matrix strip \( A_p \) stored in the processors to the other processors through multicast. Then, each processor \( (p_i) \) performs the multiplication on the received matrix strip \( (A_p) \) and the stored matrix strip \( (B_i) \) and puts the result in \( C_{pi} \) which is kept by processor \( (p_i) \). After the completion of all the phases, the result of the multiplication (matrix \( C \) ) is stored in the processors.

Let the size of matrices \( A \) and \( B \) be \( n \times n \), the time complexity of computation of the above parallel algorithm is \( O(\frac{n^3}{m}) \) and that of communication is \( O(n^2) \).

### 7.4 Experiment

Fox's algorithm is selected for comparison with the new multiplication algorithm. This algorithm is designed for the hypercube and the mesh architectures. The explanation and the analysis of this algorithm can be found in [18].

The time complexity of computation is \( O(\frac{n^3}{m}) \) and that of communication is \( O(\sqrt{mn^2}) \). Notice, the time spent on communication in the new algorithm does not increase as the data size increases, and this is not the case for Fox's algorithm.

An experiment was taken to compare the performance of these two algorithms. Both algorithms were implemented on the ObjectBalance and tested using a set of dedicated workstations. Due to resource limitations, the network which connects the workstations cannot be dedicated to the testing programs. However, the workload of the network is slight when the experiments were undertaken. Each program performs two sets of experiments with input matrices of size \( 400 \times 400 \) and \( 700 \times 700 \). The results are shown in figures 7.2 and 7.3. Another sequential program is written for computing the same data files. The time measured for multiplying matrices of size \( 400 \times 400 \) and \( 700 \times 700 \) were 389.32 sec and 2245.08 sec respectively for the sequential programs.

The experimental results agree with the theoretical result. The scalability of the new algorithm is better than Fox's algorithm. This shows the need for re-designing a set of new algorithms for the NOW environment. Otherwise, the computational power of NOW will not be fully exploited.
Figure 7.2: Experiment I

Figure 7.3: Experiment II
7.5 Chapter Summary

In this chapter, the issues of parallel algorithms are discussed. Parallel programmers will likely use existing algorithms for their needs. However, there are only a few algorithms designed for the NOW environment. We have conducted an experiment to show that parallel algorithms designed for other parallel architectures may not be suitable for the NOW environment. Also, a new matrix multiplication algorithm was designed and is compared with Fox's matrix multiplication algorithm. The experimental results agree with the expectation that the new algorithm is better than Fox's algorithm in terms of the scalability by reducing network traffic as the number of processors increases.
Chapter 8

Conclusions

8.1 Overview

The thesis describes the design and the implementation of an object-oriented parallel programming system. The design of a programming system needs to consider not only the human aspects but also the system aspects. In the human aspects, the Synchronous Object Model is designed to help parallel programmers organize their programs. This model works on existing object-oriented model without requiring additional constructs for expressing parallelism. In the system aspects, a new synchronous algorithm is tailor made for the broadcasting medium commonly found in the NOW environment. Supports of multicast and load balancing are also provided.

8.2 Contributions

The major contribution of this thesis is that we have considered the complete parallel programming system, covering both the human aspects and the system aspects. In most existing parallel programming systems, the focus is either heavily weighted on the human or the system aspects. This may lead to a programming system which may be user friendly but inefficient. In some cases, the system is very efficient in terms of performance but lacks convenient high level programming constructs for the programmers. This makes the system difficult to use. In our programming system, the parallel programming interfaces are carefully designed so that programmers can write parallel programs easily and the supporting run-time system can be implemented.
efficiently.

In order to achieve this goal, an object-oriented Synchronous Object Model is proposed. The object model makes use of the inherent parallel nature of the object-oriented model to express parallelism. With the Synchronous Object Model, programmers can design and write their parallel programs using existing object-oriented techniques without the needs of thinking in terms of parallelism in designing the control flow. This is useful since human beings are weak in thinking parallelism.

Since the Synchronous Object Model uses the existing object-oriented model for expressing parallelism, it can be implemented with any existing object-oriented languages. Programmers need not learn a new language in order to use the programming system. This also simplifies the implementation of the programming system as there is no need to design a new language and write a new complier.

Apart from that, an object-oriented framework is provided in our programming system. The framework approach reduces the effort of parallel programming by code reuse and design reuse. The code commonly needed in most parallel programs is encapsulated in the object-oriented framework. Also, the framework can be treated as a pre-defined structure of a parallel program. Therefore, parallel programmers spend much less time in designing and writing parallel programs.

In the system aspects, the support of multicast and load balancing are provided to exploit the characteristics of the NOW environments.

Load balancing is known to improve the performance of parallel programs. In the NOW environment, the scheduling algorithms required to achieve load balance are complicated. Also, it is difficult to have a single scheduling algorithm which suits different types of parallel programs. In our programming system, the framework provides facilities for algorithm designers to incorporate their selected scheduling algorithms into the programming system. The extensible architecture of our programming framework makes our programming system different from the existing systems.

The support of multicast is also a unique feature of our programming system. In the Ethernet environment, data can be sent to multiple hosts efficiently through multicast. It is common to have data sent to multiple hosts in parallel computations. Therefore, multicast support is important in the performance of parallel programs and this is confirmed by our experimental results.
8.3 Future Works

We have proposed a few techniques for load balancing, including the Dynamic Decomposition and the techniques of mapping time complexity to real execution time. The experimental results showed that the techniques work well in practical situations. In our prototype implementation, programmers are responsible for employing these techniques in their programs. It would be nice to automate the techniques. This requires work in designing a distributed database for storing the run-time information of the objects.

The experimental results shown in chapter 7 showed that existing parallel algorithms do not work well in the NOW environment. Since data transfer time plays a key role in the performance of parallel algorithms, and the communication topology of NOW is completely different from that in parallel machines, there is a need for a new set of parallel algorithms designed for the NOW environment for different applications in order to fully exploit the power of NOW.
Appendix A

Source Code of Multicast Matrix Multiplication Algorithm

The following are the header file and source code of our new multicast matrix algorithm.

/*
   Interface for a parallel Math Library

Written by: Kenneth Chan <waming@cs.ust.hk>
Date: June 1996

*/

#ifndef _ObMath_h
#define _ObMath_h

#include <ObjectBalance/SyncObject.h>
#include <ObjectBalance/ObGeometry.h>
#include <ObjectBalance/FileIO.h>
#include <MatrixClass.h>
#include <FloatMatrixClass.h>

typedef unsigned char byte;

72
@interface ObMatrix : SyncObject
{
    id         subMatrixA, subMatrixB, subMatrixC;
    int        phase;
    byte       *buffer;
    size_t     bSize;
}

- init;
- (BOOL) multiplyWithDataFile: (char *) fileA : (char *) fileB;
- multiply;
- readData: (char *) fileA: (char *) fileB;
- mainCode;

@end

/
*

Implementation of a parallel Math Library

Written by: Kenneth Chan <waiming@cs.ust.hk>
Date: June 1996

*/

#include "ObMath.h"

@implementation ObMatrix

- init
{
    /*
     * This is a typical initial session
     */
    phase = 0;

    /*
     * Tell the system to use "brocast" method in sending data.
     */
    [self setMsgHandler: @selector(brocast)];

    return self;
}

-(BOOL) multiplyWithDataFile: (char *) fileA : (char*)fileB
{
    char outFileName[255];
    FileBlock *rltFile;

    /*
     * Read in the data files
     */
    [self readData: fileA : fileB];

    /*
     * Multiply the two matrices
     */
    [self multiply];

    printf (stderr, "Finish calculation\n");
    printf ("matrix(0,0) = %0.2f\n", [subMatrixC value:0:0]);
sprintf (outFileName, "result%d.dat\0", self->nodeNum);
fprintf (stderr, "write results to file:%s\n", outFileName);

/**
 * Write the results to the file using the FileIO Class
 */
rltFile = [[FileBlock alloc] init];
[rltFile create: outFileName itemSize: sizeof(double)
     numOfRows: [subMatrixC numOfRows]
     numOfCols: [subMatrixC numOfCols]];
[rltFile writeDataOfRegion: 0 : 0 : [subMatrixC numOfRows]-1
     : [subMatrixC numOfCols] -1 from: (char*)[subMatrixC matrixPtr]];

[rltFile close];
[rltFile release];

[self showStatistic];

fflush(stdout);

return YES;
}

- multiply
{
    [self setInputBuff: self->buffer size: self->bSize];
    for (phase = 0; phase <= self->totNodes ; ++phase) {
        fprintf(stderr,"multiply at phase: \%d\n", phase);
        /*
         * This Object inherits SyncObject Properties
         */
* When receiving "run" message, the execution cycle of SyncObject
* starts:
*
* 1. - mainCode { ... } get executed
* 2. Instances of object will synchronize
* 3. Data in out buffer will sent to other objects' in buffer
*
*/

[self run];
}

return self;
}

- mainCode
{
  StripMatrix *inMatrixStrip;
  id tmpM;
  double *matPtr;
  int ncols, rstart;
  size_t s;

  if (phase != 0) {

    if ((phase -1) == [self coordinate]) {
      rstart = [subMatrixA rowStart];
      tmpM = [[DoubleMatrixClass alloc]
        init: [subMatrixA numOfrows] : [subMatrixB numOfcols]];
      [subMatrixA multiply:subMatrixB storeTo: tmpM];

    }
```c
} else {
  inMatrixStrip = [StripMatrix newWithPackedData: (byte*)self->buffer];
  tmpM = [inMatrixStrip multiply: subMatrixB ];
  rstart = [inMatrixStrip rowStart];
}
  matPtr = (double*) [subMatrixC matrixPtr];
  ncols = [subMatrixC numGfCols];
  bcopy ([tmpM matrixPtr], &matPtr[rstart * ncols],
          [tmpM memSize] );

  [tmpM free];
}

if (phase == [self coordinate]) {
  s = [subMatrixA packMatrixTo: (byte*) self->buffer];

  [self setOutputBuff: self->buffer size: s];
  [self setInputBuff: NULL size: 0];
} else {
  [self setOutputBuff: NULL size: 0];
  [self setInputBuff: self->buffer size: bSize];
}

if (phase == self->totNodes) {
  [self setInputBuff: NULL size: 0];
  [self setOutputBuff: NULL size: 0];
}

return self;
}
```
- readData: (char **) fileA: (char **) fileB

{
    int msSizeA, msSizeB; /* num. of rows in the matrix strip */
    int matrixStripPos, totMatrixStrip;
    int *ds;
    int tot;
    int matrixARowsNum, matrixAColsNum, matrixBRowsNum, matrixBColsNum;
    int stripSizeA, stripSizeB, extraRows, extraCols;
    int rowStart, colStart;
    FileBlock *fileObjA, *fileObjB;

    fileObjA = [FileBlock alloc] init];
    fileObjB = [FileBlock alloc] init];
    [fileObjA open: fileA : "r"];  
    [fileObjB open: fileB : "r"];  

    matrixAColsNum = [fileObjA numOfCols];
    matrixARowsNum = [fileObjA numOfRows];
    matrixBColsNum = [fileObjB numOfCols];
    matrixBRowsNum = [fileObjB numOfRows];

    if (matrixAColsNum != matrixBRowsNum) {
        err_ret ("readData: matrix’s A num. of Cols NOT equal to matrix’s B num. of rows");
        [self free];
        exit(1);
    }

    /*
     * When this instance object is created, the system will assign a
     * coordinate to this object.
     */
*  
*  [self coordinate] is to query this object's identity 
*  
*/

matrixStripPos = [self coordinate];
ds = [self dimSize];
totMatrixStrip = *ds;

stripSizeA = matrixARowsNum / totMatrixStrip;
extraRows = matrixARowsNum % totMatrixStrip;
stripSizeB = matrixBColsNum / totMatrixStrip;
extraCols = matrixBColsNum % totMatrixStrip;

msSizeA = stripSizeA;
msSizeA += ((matrixStripPos < extraRows) ? 1 : 0);

msSizeB = stripSizeB;
msSizeB += ((matrixStripPos < extraRows) ? 1 : 0);

rowStart = matrixStripPos * stripSizeA +
    ((matrixStripPos < extraRows) ? matrixStripPos : extraRows);
colStart = matrixStripPos * stripSizeB +
    ((matrixStripPos < extraCols) ? matrixStripPos : extraCols);

subMatrixA = [[StripMatrix alloc] init: msSizeA : matrixAColsNum
    rowStartAt: rowStart
    colStartAt: 0];
subMatrixB = [[StripMatrix alloc] init: matrixBRowsNum : msSizeB
    rowStartAt: 0
    colStartAt: colStart];
subMatrixC = [[StripMatrix alloc] init: matrixARowsNum: mSizeB
  rowStartAt: 0
  colStartAt: colStart];

/*
 * Initialize the matrix and read the data into memory through
 * the FileIO Class interface
 */

[fileObjA readDataOfRegion: rowStart: 0
  : rowStart+mSizeA-1 : matrixAColsNum-1
to: (char*)[subMatrixA matrixPtr]];

[fileObjB readDataOfRegion: 0: colStart
  : matrixBRowsNum-1 : colStart+mSizeB-1
to: (char*)[subMatrixB matrixPtr]];

[subMatrixC zero];

/*
 * Allocating buffer or incoming/outgoing data
 */

self->bSize= [StripMatrix packContentSize: stripSizeA+1 : matrixAColsNum];
self->buffer= (byte*) alloc_block(self->bSize);

return self;
}

@end
@implementation StripMatrix

+ (size_t) packDataSize: (int) numRows: (int) numCols
{
    size_t s;

    s = 4 * sizeof(int);

    s += (s % sizeof(double)) > 0 ? sizeof(double) - (s % sizeof(double)) : 0;
    s += numRows * numCols * sizeof(double);

    return s;
}

+ newWithPackedData: (byte*)inData
{
    StripMatrix *sm;
    int *ipt;
    double *dpt;
    size_t s;

    sm = [self alloc];
    ipt = (int*) inData;
    dpt = (double*) inData;


    s = 4 * sizeof(int);
    dpt += s / sizeof(double) + (s % sizeof(double) ? 1 : 0);
    bcopy (dpt, sm->matrix, sm->memSize);

    return sm;
}
return sm;
}

- (int) packMatrixTo: (byte*) buff
{
  double *dpt;
  int *ipt;
  size_t s;

  dpt = (double*) buff;
  ipt = (int*) buff;
  ipt[0] = self->rowStart;
  ipt[1] = self->colStart;

  s = 4 * sizeof(int);
  dpt += s / sizeof(double)
      + (s % sizeof(double) ? 1 : 0);

  bcopy (self->matrix, (char *)dpt, self->memSize);
  s += self->memSize;

  return s;
}

- (int) rowStart
{
  return self->rowStart;
}

- (int) colStart
The following is the main program of the multicast multiplication algorithm.

```c
#include <stdio.h>
#include <ObjectBalance/SyncAppObject.h>
#include "ObMath.h"

char **NSArgv;

main(int argc, char *argv[])
{
    id paraMatrix;
    id matrixApp;
    DoubleMatrixClass *m1, *m2, *m3;
```
double d;

if (argc != 3) {
    fprintf(stderr, "Usage %s: <matrix A data> <matrix B data>\n", argv[0]);
    exit(1);
}

matrixApp = [SyncAppObject alloc];
[matrixApp initWithClassName: "ObMatrix" ];

paraMatrix = [matrixApp syncObjectGroup];


[matrixApp free];
}
Appendix B

Source Code of Fox's Algorithm

The following are the header file and source code of the Fox's matrix algorithm.

/*
   Interface for a parallel matrix example

Written by: Kenneth Chan <waiming@cs.ust.hk>
Date: December 1995

This example is to illustrate the use of ObjectBalance to write Object-Oriented parallel program
*/

#include <ObjectBalance/SyncObject.h>
#include <ObjectBalance/OBGeometry.h>
#include <ObjectBalance/FileIO.h>
#include "MatrixClass.h"
#include "FloatMatrixClass.h"

/*
 * In this example, another Virtual Geometric Topology (MatrixCell2D) is
 * used for writing the Fox's algorithm.

85
interface MatrixEngine : MatrixCell2D
{
    id subMatrixA, subMatrixB, subMatrixC; /* objects from MatrixClass */
    id tmpsubMatrixA, tmpsubMatrixB;
    id aRowSender, aColRotator;         /* objects from SyncObjects class */

    int xpos, ypos;                    /* the geometry of the running object*/
}

- mainCode;
- (BOOL) multiplyWithDataFile: (char *) fileA : (char *) fileB;
- readData : (char *) dataFileA: (char *) dataFileB ;
- printString: (char *) s;
- (void) showStatistic;

- (void) free;

@end

/*

Implementation of a parallel matrix example

Written by: Kenneth Chan <waiming@cs.ust.hk>
Date: December 1995

This example is to illustrate the use of ObjectBalance to write Object-Oriented parallel program

*/

#include "obmatrix.h"
implementation MatrixEngine

- (BOOL) multiplyWithDataFile: (char *) fileA : (char *) fileB
{
        char outFileName[255];
        FileBlock *rltFile;

        [self readData: fileA : fileB];
        [self run];
        /
        /*
         * Write data to file
         */
        sprintf (outFileName, "pm-result%d.dat\0", self->nodeNum);
        fprintf (stderr, "write results to file:\%s\n", outFileName);
        rltFile = [[FileBlock alloc] init];
        [rltFile create: outFileName itemSize: sizeof(double)
                numOfRows: [subMatrixC numOfRows]
                numOfCols: [subMatrixC numOfCols]];
        [rltFile writeDataOfRegion: 0 : 0 : [subMatrixC numOfRows]-1
                : [subMatrixC numOfCols] -1 from: (char*)[subMatrixC matrixPtr]];

        [rltFile close];
        [rltFile release];

        [aRowSender showStatistic];
        [aColRotator showStatistic];
        [self showStatistic];

        fflush(stdout);
return YES;
}

- (void) showStatistic
{
    long    totMain, totComm, totSys, totElap;

    totComm = [aRowSender totalCommTime] + [aColRotator totalCommTime];
    totSys = self->totElapsedTime - self->totMainCodeTime
        + [aRowSender totalElapsedTime] + [aColRotator totalElapsedTime]
        - totComm;

    totMain = self->totMainCodeTime - [aRowSender totalElapsedTime] +
        - [aColRotator totalElapsedTime];

    totElap = self->totElapsedTime;

    printf ("Statistic Info(MatrixEngine)\n");
    printf ("="*200\n");
    printf ("Total Execution time in mainCode: %ld\n", totMain);
    printf ("Total Communication time: %ld\n", totComm);
    printf ("Total System time: %ld\n", totSys);
    printf ("Total Elapsed time: %ld\n", totElap);
}

- mainCode
{
    int    phase, totPhase;

    printf ("Starting computational phase \n");
    aRowSender = [self newChildObjectWithClass: [MatrixCell2D class]];

    printf ("%d", phase);
    printf ("%d", totPhase);

    printf ("Total Execution time in mainCode: %ld\n", totMain);
    printf ("Total Communication time: %ld\n", totComm);
    printf ("Total System time: %ld\n", totSys);
    printf ("Total Elapsed time: %ld\n", totElap);
}
aColRotator = [self newChildObjectWithClass: [MatrixCell2D class]];

tmpsubMatrixA = [[DoubleMatrixClass alloc] init : [subMatrixA numOfRows]:
[subMatrixA numOfCols]];

tmpsubMatrixB = [[DoubleMatrixClass alloc] init : [subMatrixB numOfRows]:
[subMatrixB numOfCols]];

totPhase = [self xDimSize];

for (phase =0; phase < totPhase; ++phase) {
    size_t datasize;
    double *mptr, *destptr;
    DoubleMatrixClass *tmpMatrix;

    fprintf(stderr, "obmatrix: Phase %d\n",phase);
    /*
     * code for handling rowSender object
     */
    if (ypos == ((xpos+phase)%totPhase )) {
        /* broadcast subMatrixA to the workers in the same row */

        [aRowSender setMesgHandler: @selector(multicastOnRow)];
        mptr = [subMatrixA matrixPtr];
        datasize = [subMatrixA memSize];

        [aRowSender setOutputBuff : mptr size: datasize ];

        /* Tell the system no mesg. is going to receive */
        [aRowSender setInputBuff : NULL size: 0];
    }
[aRowSender run];

destptr = [tmbsubMatrixA matrixPtr];
bcopy((char *)mptr,(char *) destptr, datasize);

} else {
    /* wait for the subMatrixA to come */

    [aRowSender setMsgHandler: @selector(noMsgSend)];
    mptr = [tmbsubMatrixA matrixPtr];
    datasize = [subMatrixA memSize ];

    [aRowSender setInputBuff : mptr size: datasize];

    [aRowSender run];

}

/*
 * code for handling colRotator object
 */

if (phase != 0 ) {

    [aColRotator setOutputBuff: [subMatrixB matrixPtr ]
    size: [subMatrixB memSize]];  
    mptr = [tmbsubMatrixB matrixPtr];

    [aColRotator setInputBuff: mptr size: [tmbsubMatrixB memSize]];  
    [aColRotator setMsgHandler : @selector(sendtoTop)];

    [aColRotator run];

90
/* arrange the matrix pointers */
[tmpsubMatrixB setMatrixPtr: [subMatrixB matrixPtr]];
[subMatrixB setMatrixPtr: mptr];

}

/
* the final phase for matrix computation */

err_ret("obmatrix: start computation");

tmpMatrix = [tmpsubMatrixA multiply: subMatrixB];

err_ret("obmatrix: end computation");

[subMatrixC add:tmpMatrix storeTo: subMatrixC];

[tmpMatrix free];
}

fprintf (stderr, "mainCode: finish computation
");

}

- readData : (char *) dataFileA : (char *) dataFileB
{

/*
 * Code the reading a portion of the data file and place the data in
 * the matrix objects ....

91
* subMatrixA, subMatrixB, subMatrixC
*/

FileBlock *fpA, *fpB;
int num_rowsA, num_rowsB ;
int num_colsA, num_colsB ;
int subNumRowsA, subNumRowsB;
int subNumColsA, subNumColsB;
int i, j, tmp;
int x_dim, y_dim, x_pos, y_pos;
int nr, nc, r,c ;
double *mptr;

fprintf (stderr,"Reading data files ... \n");
fpA = [[[FileBlock alloc] init] open: dataFileA: "r"];
fpB = [[[FileBlock alloc] init] open: dataFileB: "r"];

/* initialize the data matrix */
um_rowsA = [fpA numOfRows];
um_colsA = [fpA numOfCols];
um_rowsB = [fpB numOfRows];
um_colsB = [fpB numOfCols];

x_dim = [self xDimSize];
y_dim = [self yDimSize];

x_pos = [self xCoordinate];
y_pos = [self yCoordinate];

subNumRowsA = (num_rowsA / x_dim) + ((num_rowsA % x_dim) ? 1 : 0 );
subNumColsA = (num_colsA / y_dim) + ((num_colsA % y_dim) ? 1 : 0 );
subNumRowsB = subNumColsA;

subNumColsB = (num_colsB / y_dim) + ((num_colsB % y_dim) ? 1 : 0);

subMatrixA = [[DoubleMatrixClass alloc] init: subNumRowsA: subNumColsA];
subMatrixB = [[DoubleMatrixClass alloc] init: subNumRowsB: subNumColsB];

[subMatrixA zero];
[subMatrixB zero];

nr = ((x_pos+1) * subNumRowsA < num_rowsA ?
(x_pos+1) * subNumRowsA : num_rowsA );

nc = ((y_pos +1) * subNumColsA < num_colsA ?
(y_pos +1) * subNumColsA : num_colsA);

mptr = (double*) [subMatrixA matrixPtr];
for ( r = 0; r < (nr-x_pos*subNumRowsA); ++ r){
    [fpA readDataOfRegion: r+x_pos*subNumRowsA:y_pos*subNumColsA : r+x_pos*subNumRowsA : nc-1 to: &mptr[r*subNumColsA]];
}

nr = (((x_pos+1) * subNumRowsB) < num_rowsB ?
(x_pos+1) * subNumRowsB : num_rowsB );

nc = (((y_pos +1) * subNumColsB) < num_colsB ?
(y_pos +1) * subNumColsB : num_colsB );

mptr = [subMatrixB matrixPtr];
for ( r = 0; r < (nr-x_pos*subNumRowsB); ++ r){
    [fpB readDataOfRegion: r+x_pos*subNumRowsB:y_pos*subNumColsB :
r+x_pos*subNumRowsB : nc-1 to: &mptr[r*subNumColsB];
}

[fpA close];
[fpB close];

subMatrixC = [[DoubleMatrixClass alloc] init: subNumRowsA: subNumColsB];
[subMatrixC zero];

return self;
;
-
init
{

/*
 * Initialize the object inheriting the MatrixCell2D
 */

xpos = [self xCoordinate];
ypos = [self yCoordinate];

}

- (void) free
{
	[tmpsubMatrixA free];
	[tmpsubMatrixB free];
	[subMatrixA free];
	[subMatrixB free];
	[subMatrixC free];
}
- printString: (char *) s
{
    printf("got the string -> %s\n", s);

    return self;
}
@end

The following is the main program of the Fox’s algorithm.

#include <stdio.h>
#include <ObjectBalance/SyncAppObject.h>
#include "obmatrix.h"

char **NSArgv;

main (int argc, char *argv[])
{
    id paraMatrix;
    id matrixApp;
    id *ro;
    char fileNameA[20], fileNameB[20];
    int i;

    if (argc != 3 ) {
        fprintf(stderr, "Usage %s: <matrix A data> <matrix B data>\n", argv[0]);
        exit(1);
    }

    matrixApp = [SyncAppObject alloc];
matrixApp initWithClassName: "MatrixEngine";

strcpy (fileNameA, argv[1]);
strcpy (fileNameB, argv[2]);

paraMatrix = [matrixApp syncObjectGroup];

[paraMatrix multiplyWithDataFile: fileNameA : fileNameB];

[matrixApp free];

}
Bibliography


