PARALLEL IMPLEMENTATION OF CONTAINER USING PARALLEL VIRTUAL MACHINE

by

Young Kwan Cho

A Thesis Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

MASTER OF SCIENCE

WITH A MAJOR IN ELECTRICAL ENGINEERING

In the Graduate College

THE UNIVERSITY OF ARIZONA

1 9 9 5

PARALLEL IMPLEMENTATION OF CONTAINER USING PARALLEL VIRTUAL MACHINE

by

Young Kwan Cho

An Abstract of a Thesis Submitted to the Graduate
Faculty of The University Of Arizona

in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

Major Subject: Computer Engineering

The original of the complete thesis is on
file in the University of Arizona Library

Approved by the
Examining Committee:

____
Dr. Bernard P. Zeigler

The University of Arizona
Tucson, Arizona

July 1995

STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED:____________________________

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

____________________________ __________________

Dr. Bernard P. Zeigler
Professor of
Electrical and Computer Engineering

Date

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Bernard P. Zeigler, for his guidance and help on my research. Without his valuable advice and encouragement, this thesis would not be accomplished. I also thank Dr. Martinez and Dr. Marefat for their serving as committee members and reviewing my thesis.

I want to express my gratitude to Dr. Hongki Sung, Dr. Jinwoo Kim, Mr. Hyupjae Cho, Mr. Yoonkeon Moon, Mr. Jeongkeun Kim, Mr. Doohwan Kim, Mr. Sankait Vahie, and Mr. Jerry Couretas for sharing their valuable experiences which enabled me to realize the existence of new areas where I have never been. I would like to thank many other friends not mentioned here for their help.

The Republic of Korea Air Force which made it possible for me to take this great opportunity in the States also deserves to get my appreciation.

TABLE OF CONTENTS

LIST OF FIGURES 7

ABSTRACT 9

1. INTRODUCTION 11
1.1. Objectives and Approach 11
1.2. Structure of Thesis 13

2. CONTAINER CLASS 15
2.1. An Overview of Containers 15
2.1.1. Classes in Object–Oriented Programming 17
2.1.2. Inheritance 17
2.1.3. Iterators 19
2.2. Container Class Hierarchy 19
2.2.1. Container Base Class 22
2.2.1.1. Basic Methods 22
2.2.1.2. Ensemble Methods 23
2.2.2. Unordered Subclasses 25
2.2.2.1. Bags 26
2.2.2.2. Sets 27
2.2.2.3. Relations 28
2.2.2.4. Functions 30

3. DISTRIBUTED NETWORK COMPUTING AND PVM 32
3.1. Distributed Computing 32
3.1.1. Heterogeneous Network Computing 33
3.1.2. Parallel Programming Systems 34
3.2. Parallel Virtual Machine 36
3.2.1. Computing Model of PVM 37
3.2.2. Features of PVM 38

4. IMPLEMENTATION OF PARALLEL CONTAINERS 40
4.1. Overall Architecture 40

4.1.1. Coordination of Elements 40
4.1.2. Design Considerations 42
4.1.3. Architecture of Parallel Container 43
4.2. Implementation 48
4.2.1. Communication Manager 48
4.2.1.1. Enrolling in and Exiting from PVM 49
4.2.1.2. Spawning and Terminating Processes 50
4.2.1.3. Passing Messages 52
4.2.2. Parallel Containers 56
4.2.2.1. The Object Identity 56
4.2.2.2. Constructor 56
4.2.2.3. Adding Elements 57
4.2.2.4. Referring Elements 60
4.2.2.5. Ensemble Methods 62
4.2.3. Derived Classes 64
4.2.3.1. Class Bags and Class Sets 64
4.2.3.2. Class Relations and Class Functions 65
4.2.4. An Overview of Interaction between Objects 66

5. AN EXAMPLE OF REAL APPLICATION: A Parse Tree 70
5.1. The Simulation of Watershed Model 70
5.2. The Role of A Parse Tree 73
5.3. Experimental Results 76

6. CONCLUSION AND FUTURE WORK 83
6.1. Conclusion 83
6.2. Future Work 85

Appendix A. Parallel Container User's Guide 87
A.1. Setting Up Environment 87
A.1.1. Obtaining Source Codes 87
A.1.2. Installing PVM 88
A.1.3. Starting PVM 88
A.1.4. Configuring Virtual Machine 89
A.2. Using Parallel Container 91
A.2.1. Writing Applications 91
A.2.2. Compiling and Debugging Source Codes 92
A.2.3. Running Applications 93

REFERENCES 95

LIST OF FIGURES

2.1. Class Hierarchy of Containers 20
2.2. Object behavior specification for entities 21
2.3. Object behavior specification for containers 22
2.4. Object behavior specification for ensemble methods 24
2.5. Object behavior specification for bags 26
2.6. Object behavior specification for sets 27
2.7. Object behavior specification for relations 29
2.8. Object behavior specification for functions 30

3.1. PVM architectural Model 37

4.1. Massively Parallel Implementation of Lists 41
4.2. Interactions between a coordinator and processors 42
4.3. Architecture of Parallel Containers 44
4.4. Class Hierarchy of Parallel Container 45
4.5. Hierarchical construction of objects in parallel containers 47
4.6. Methods for enrolling in and exiting from PVM 49
4.7. Method for spawning a new process 51
4.8. Control message format 52
4.9. Methods for sending and receiving messages 53
4.10. Methods for sending and receiving integers 55
4.11. Constructor of Class ENTITY 57
4.12. A method for Adding an element 58
4.13. The class definition of an element 59
4.14. Helper methods for adding elements 60
4.15. The method for asking existence of an ENTITY 61
4.16. An example of ensemble method implementation 63
4.17. A method for removing an element 64
4.18. The class definition of pairs 66
4.19. Object Creation 67
4.20. Interaction between objects 68

5.1. Architecture for Distributed Simulation Environment 71

5.2. Cellular Space Representation of Riparian Ecosystem 72
5.3. An Example of a Parse Tree 74
5.4. Class definition of AND Connective 75
5.5. Making a real parse tree from EVENT definition 77
5.6. The Result of Sequential Running (Unit is seconds) 79
5.7. The Result of Parallel Running (Unit is seconds) 80

A.1. An example of the hostfile 90

ABSTRACT

Parallel and distributed computing is becoming essential for high performance computing. Conventional supercomputers may be used for these kinds of computations or a collection of workstations may be used if an appropriate software system is provided.

In this thesis, a *Parallel Distributed Container* is proposed and implemented with the use of C++, a widely used object–oriented programming language, and PVM(Parallel Virtual Machine), a parallel programming software system. The goal is to provide primitives for utilizing a collection of heterogeneous computing resources to solve large problems and to speed up computations.

A *container* is an object that enables us to store and organize objects. In a parallel distributed environment, a container can be used to organize and control processing elements transparently to the user. The main features of object–oriented programming such as the concept of class, inheritance, information hiding, data abstraction, and polymorphism enable us to implement such a parallel distributed container class.

We demonstrate the utility of the parallel distributed container with significant event detection in large GIS databases in the context of a watershed model simulation.

To my wife, Soo–Young, and daughter, Ahyeon

for their love, support, and encouragement.

CHAPTER 1

INTRODUCTION

1.1 Objectives and Approach

Recently, object–oriented software construction is gaining popularity among many researchers. The main features of object–oriented programming such as *information hiding*, *data abstraction*, *polymorphism*, and *inheritance* are used in various areas. Any real world object can be expressed in an object–oriented system, and since real world objects may perform things in concurrent, object–oriented systems may also have great potential for being developed as concurrent and parallel systems. However, most of the object–oriented languages currently in use such as C++ [4], Smalltalk [10], and Eiffel [5] are sequential in nature.

Although object–oriented languages are sequential, it is natural to model a real world object as a unit of concurrency in the context of parallel computation. Thus an object can be treated as a concurrent unit to construct a whole parallel system. The message passing paradigms can be used to communicate with each other between active objects to cooperate for a single large problem. Much effort has been made

to provide maximum computational and modeling power through the Concurrent Object–Oriented Programming Languages(COOPL) [6, 7, 11, 27, 29, 13].

Class containers are a well known concept in object–oriented programming(OOP) area. There are many ways to implement class containers [1, 9, 10, 12]. Dr. Zeigler [1] has implemented a sequential version of class containers in Scheme, one of Lisp languages, and in C++. He has also proposed to extend the sequential version of class containers to the parallel and distributed containers.

A container is an object that contains other objects. As such, a container is a generalization of linked–list implementation. An entity is a basic object that identifies each object itself. Every object in the container class hierarchy inherits a class entity. This entity can be mapped to a single process or a task in PVM [21].

PVM provides methods that connect a collection of heterogeneous workstations as a single virtual machine with distributed memory architecture [21]. PVM was developed by Oak Ridge National Laboratory to provide a parallel and distributed computing environment by connecting the existing heterogeneous workstations. Using existing workstations for distributed computations should be cheaper than using conventional supercomputers. This is the most important motivation to use a cluster of workstations for the parallel distributed computations in this thesis.

The distributed computations are inherently parallel. Therefore, once a distributed system is developed, it can then be used for parallel computation to solve a large problem. We can distribute computing nodes over the hosts in a collection

of workstations connected with PVM and can get more computational power in parallel. However, planning how to create a real computing node and control these nodes efficiently and conveniently may be a non–trivial job as we construct a parallel distributed system.

The goal of this thesis is to provide primitives for the parallel distributed object–oriented container which is an extension of a sequential version of the class container implemented in C++ [1]. This parallel container will provide the ability to specify and control concurrent objects in C++.

1.2 Structure of Thesis

The remainder of this thesis is organized as follows: Chapter 2 discusses some important features of object–oriented programming with C++ as the central figure and then explains a sequential version of class container in terms of the specification of object behaviors. These will be the requirements and specifications of the parallel distributed container. Chapter 3 describes the parallel distributed computing within a heterogeneous workstation clustering environment and PVM which is the main communication tool for connecting distributed computing resources. In Chapter 4, the implementation of the parallel distributed container will be discussed. Chapter 5 shows an example of real applications. A parse tree is an example. The speed–up of parallel computing by the use of the parallel container will be compared to the result

of a sequential version of a parse tree evaluation. This evaluation will be taken on the same data. Chapter 6 concludes this thesis and proposes some future work.

CHAPTER 2

CONTAINER CLASS

This chapter describes class containers in terms of their object behavior specifications which serve as requirements and specifications of the parallel distributed container. The parallel distributed container should have the same behavior as the sequential containers do but runs in parallel.

Before class container is discussed, some terminologies should be clarified. *Sequential container* is a container class which is implemented on a single process in C++ [1]. This is based on a pointer in C++. *Parallel distributed container* or *parallel container* is an extension of sequential container. It will be implemented on a collection of heterogeneous machines connected by PVM.

2.1 An Overview of Containers

Many object–oriented programming languages have a container concept, and often an iterator concept as well. In smalltalk, a "collection" class is defined as a basic class which represents a group of objects [10]. In C++, there are two class libraries available from the Free Software Foundation: National Institutes of Health Class

Library(NIHCL) [9] and LIBG++ [12]. LIBG++ follows a *forest* approach, in which a number of independent classes are provided. NIHCL, on the other hand, follows a *hierarchical* approach, deriving all classes from a common root class. This approach is inspired from a collection class in smalltalk [8]. We also employ a hierarchical approach to develop the parallel container.

Basically, a container object is an object that contains other objects. As such, it is a generalization of the linked lists and vectors, found in traditional programming languages. Container classes are specialized through the way by which objects in a container are organized. Examples of containers are arrays, stacks, queues, linked lists, bags, sets, dictionaries, etc. Operations commonly performed on a container are as follows:

- adding new objects to a container
- removing objects from a container
- getting objects from a container
- iterating all objects through a container

From now on, some important properties of object–oriented programming(OOP) will be discussed briefly with C++ as the centering figure since C++ will be used to implement the parallel container classes in this thesis.

2.1.1 Classes in Object–Oriented Programming

In Object–Oriented programming languages, a class is defined as an abstract data type which supports inheritance [14]. An object is an instance of a class. Objects are usually not defined individually but by means of class definitions. A class defines a type, and its definition includes both the representation of any instance of the class as well as the operations that may be performed on an instance.

Instance variables of a class are called *member variables*, and class operations on those member variables are called *member functions* (or methods). Member functions are always applied to a specific instance: within the function, any unqualified reference to a member variable of the class is bound to that instance. In C++, the binding is realized through an implicit parameter, `this`, which is a pointer to the object on which the method was invoked.

2.1.2 Inheritance

Inheritance is an essential and powerful mechanism by which new classes can be constructed on the basis of existing classes. Inheritance is the property that enables instances of a subclass to access both data and methods associated with a superclass. A subclass can inherit features from several levels above classes since inheritance is always transitive.

Given a class Animal, we may define a class Cat which is a subclass of the class Animal. And it may also inherit a class Mammal which is another subclass of the class Animal as follows:

```
class Animal{...};
class Mammal : public Animal{...};
class Cat : public Mammal{...};
```

A class Animal is called the `base` class, and classes Mammal and Cat are the `derived` classes. Class Cat inherits both member variables and member functions of class Mammal as well as those of class Animal. The keyword `public` in this context specifies that public members of Mammal are also public members of Cat; without this keyword, public members of Mammal would become private members of Cat. Cat may declare additional member variables and member functions, and it may reimplement the member functions defined in higher classes.

One of the powerful mechanisms in object–oriented languages is the late binding of method call. Suppose a method A in class Animal is reimplemented in class Mammal and a program calls method A through an object pointer whose type is Animal. This object pointer may refer to class Mammal at runtime. Then the actual type of object is determined at this time and the actual method of the type determined is executed. Late binding allows a type hierarchy to be extended with new subtype without changing to existing code. In C++, keyword `virtual` makes it possible. To achieve heterogeneity in container class, this late binding is essential.

2.1.3 Iterators

Iterator is an abstraction that supports sequential access to the individual elements of a container, without modifying the container [24]. Some of the academic languages such as Alphard [25], CLU [26], and Sather [28] include special language constructs for iterators. However, most of the modern imperative languages such as Ada and C++ offer no special support for iterators. In these languages and most others, iterator must be designed and encapsulated using the same mechanisms that are used for other user–defined abstractions.

Iterator is usually implemented by the use of `ForEach` loop. A user specified operation with arguments is applied to each and every element in a container. All the elements may have the same functions and different data for evaluations. Or they may have the functions that behave differently and the same data.

2.2 Container Class Hierarchy

This section discusses a sequential container in terms of object behavior specifications. The behaviors that will be described in this section should be requirements for the parallel distributed container.

As mentioned in Section 2.1, *Containers* are basic classes that help to store, retrieve and organize interacting objects. Container classes are generalized forms of linked lists. Operations commonly performed on a container are adding objects to it,

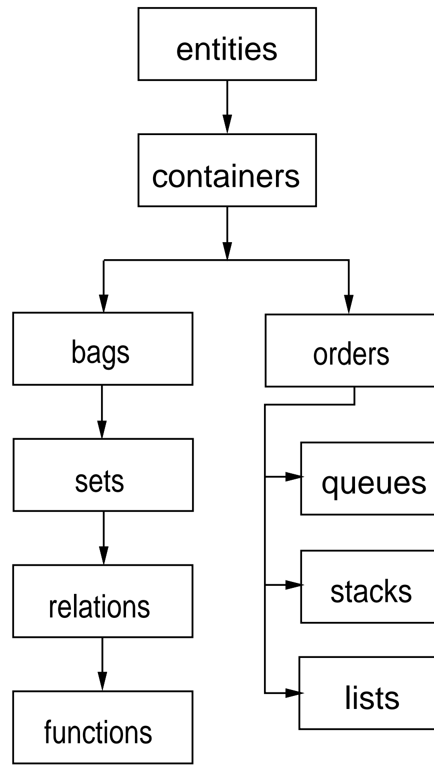

Figure 2.1: Class Hierarchy of Containers

getting objects from it, or iterating through it to perform a specific operation on all objects in the container.

Figure 2.1 shows the hierarchy of container classes and their basic methods. The classes are roughly characterized as follows:

- Containers – the base class, provides basic services for the derived classes
- Bags – counts numbers of object occurrences
- Sets – only one occurrence of any objects is allowed
- Relations – sets of key–value pairs, used in dictionary fashion

- Functions – relations in which only one occurrence of any key allowed
- Orders – maintains items in given order
- Queues – maintains items in FIFO order
- Stacks – maintains items in LIFO order
- Lists – maintains items in order determined by insertion index

As shown in Figure 2.1, container class is actually a subclass of the more basic class, `entity`. Class entity will provide methods that work closely those of container. Note that since container is derived from entity, container instances can be placed into other containers, thus setting up the basis for hierarchical construction.

```
Entity[Object]

constructor
    entity make–entity(name)

queries
    name name?(entity)
    entity equal–self?(entity,entity1)

Equivalences
    name?(make–entity(name)) = name
    equal–self?(entity,entity) = entity
    equal–self?(entity,entity1) = make–entity("null")
```

Figure 2.2: Object behavior specification for entities

Figure 2.2 shows the object behavior specification for class entity. As shown in Figure 2.2, class entity has variables for the identity of a object and functions for those instance variables. Every entity has its own name to be distinguished from other entities.

2.2.1 Container Base Class

2.2.1.1 Basic Methods

Class `containers` has only the most basic functionality. We can add instances of entities; we can ask how many items there are, and we can ask whether a specific object is included in the container.

```
Container[entity] inherit from Entity

constructor
    container make–container()

queries
    number size?(container)
    boolean is–in?(container,entity)

commands
    container' add(container,entity)

Equivalences
    size?(make–container()) = 0
    size?(add(container,entity)) = size?(container) + 1
    is–in?(make–container()) = F
    is–in?(add(container,entity),entity)) = T
```

Figure 2.3: Object behavior specification for containers

In Figure 2.3, the object behavior specification of container class is defined. Container class has instance variables like length, and a pointer of linked list to hold elements.

2.2.1.2 Ensemble Methods

Once elements are added into a container, we can define the basic methods that allow us to treat all the items in the container as a unit. These methods are called *ensemble* methods since they apply to the ensemble, or whole, container at once. These methods are essential for the parallel concurrent implementation in the distributed environment. Usually, object–oriented programming languages have this kind of concept. Some of them provide these methods as *iterators* [28, 8]. Iterators are realized by the use of ForEach methods in C++, which applies a specific method to all the items that a container holds. Ensemble methods always take a function name and arguments of the function as its arguments.

The one primitive command is `tell--all`, which is primitive in the sense that others can be synthesized from it. The behavior of the ensemble methods can be roughly described as follows:

- tell–all(container,command,args) – sends command(args) to each item in the container
- ask–all(container,query?,args) – sends query?(args) to each item in the container and collects the results in a container

```
Ensemble Methods

command
   container' tell-all(container,command,args)
      //results in changed states of all objects in container
   container' append(container,container1)      //adds in container1

query
   container1 ask-all(container,query?,args)      //returns a container
   container1 which?(container,query?,args)      //returns a container

Domain Restrictions
   tell-all(container,command,args) = defined provided that
      if is-in?(container,entity) = T
      then command(entity,args) is defined
   ask-all(container,query?,args) = defined provided that
      if is-in?(entity,container) = T
      then query?(entity,args) is defined
   which?(container,query?,args) = defined provided that
      if is-in?(entity,container) = T
      then query?(entity,args) returns a boolean

Equivalences
   is-in?(container,entity) = T
      ⇒ is-in?(tell-all(container,query?,args))
   size?(tell-all(container,query?,args)) = size?(container)
   is-in?(container,entity1) = T and query?(entity1, args) = entity
      ⇒ is-in?(ask-all(container,query?,args), entity) = T
   is-in?(ask-all(container,query?,args), entity) = F
   size?(ask-all(container,query?,args)) = size?(container)
   is-in?(container,entity) = T and query?(entity, args) = T
      ⇒ is-in?(which?(container,query?,args), entity) = T
   is-in?(which?(container,query?,args), entity) = F
   size?(which?(container,query?,args)) ¡= size?(container)
   is-in?(append(container,container1),entity) ==
      [is-in?(container,entity)= T or is-in?(container1,entity) = T]
   size?(append(container,container1)) = size?(container) + size?(container1)
```

Figure 2.4: Object behavior specification for ensemble methods

- which?(container,query?,args) – sends query?(args) to each item in the container and collects the items which returns T in a container
- append(container,container1) – adds the items in container1 to those in container

The object behavior specification for the above ensemble methods is given in terms of the state–representing queries in Figure 2.4.

2.2.2 Unordered Subclasses

Sets are the fundamental building blocks of modern mathematics. A set is an unordered collection in which no element occurs more than once. Set theory goes back to the beginning of the 20th century. Bags are a more recent generalization of sets that allow any number of occurrences of elements. Mathematically speaking, it would make sense to start with sets as the root class in a hierarchy for containers. However, our objective is not mathematical, but computational, and this drives our choice of a root class that is even more generalized than bags. As shown in Figure 2.3 of Section 2.2.1, our root class has the bare minimum of methods that will be common to all containers. This simple core enables us to derive sets and other traditional mathematical containers from the base class as well as ordered classes such as lists, stacks, and queues from that base class.

2.2.2.1 Bags

Class `bags` adds the capability to query for the number of occurrences of an entity. It also introduces a remove command. The number of occurrences of each element is a state-representing query and the effect of the commands add (inherited from the base class) and remove is easily characterized in terms of it. Note that the specification of a derived class only has to deal with the new combinations of queries and commands that were not covered in the specification of the base class.

```
Bag[entity] inherit from container

constructor
    bag make-bag()

queries
    number number-of? (bag,entity)

commands
    bag' remove(bag,entity)

Domain Restrictions
    remove(bag,entity) = defined provided that is-in?(entity,set) = T

Equivalences
    size?(remove(bag,entity)) = size?(bag) - 1
    number-of?(make-bag(),entity) = 0
    number-of?(add(bag,entity),entity)) = number-of?(bag,entity) + 1
    number-of?(add(bag,entity),entity1)) = number-of?(bag,entity1)
    number-of?(remove(bag,entity),entity)) = number-of?(bag,entity) - 1
        //remove acts to remove an instance of an entity
    number-of?(remove(bag,entity),entity1)) = number-of?(bag,entity1)
```

Figure 2.5: Object behavior specification for bags

2.2.2.2 Sets

The hierarchy shown in Figure 2.1 considers sets as a special case of bags. Sets, as mathematical objects, contain at most one occurrence of any entity. We specify this behavior by modifying the specification of add: we don't allow add to have an effect if the object to be added is already in the container. This is an example of a specification in a subclass overriding the specification inherited from the parent class. As shown in Figure 2.6, the size of a set only increases when a new entity is added. All equivalences inherited from bag apply except for the effect of add on size? and number-of? which are specified as in Figure 2.6.

```
Set[entity] inherit from bag

constructor
    set make-set()

commands
    set' add(set,entity)

Equivalences
    is-in?(set,entity) = F ⇒ size?(add(set,entity)) = size?(set) + 1
    size?(add(set,entity)) = size?(set)
    number-of?(add(bag,entity),entity)) = 1
    number-of?(add(bag,entity),entity1)) = number-of?(bag,entity1)
```

Figure 2.6: Object behavior specification for sets

Sets are used when keeping track of the very existence of an entity is important – not how many times an entity has occurred.

2.2.2.3 Relations

Mathematically, a binary relation is a set of pairs. Likewise, we specify the class relation to inherit from sets; however, the elements of such a set are so-called `key`,`value` pairs. Thus, the queries of relation have the form:

number size?(relation)
boolean is-in?(relation,key,value)

and the commands take the form:

relation' add(relation,key,value)
relation' remove(relation,key,value)

The new behavior obtainable from relations is to be able to retrieve all the values associated with a given key. This functionality is implemented in the query `assoc-all?`. We may also ask whether a key has already been given a value using `is-key-in?`. The specification is defined in Figure 2.7.

We obtain the equivalences of relation by replacing entity by key,value everywhere in the equivalences inherited from sets. The new equivalences are also described in Figure 2.7.

Relations are used to record and look-up various kinds of associated data. For example, a dictionary associates words with meanings – the same word may have multiple meanings and different words may have the same meanings.

```
Relation[entity] inherit from set

constructor
   relation make-relation()

queries
   number size?(relation) // inherited from set
   number number-of?(relation,key,value) // inherited from set
   boolean is-in?(relation,key,value) // inherited from set
   boolean key-is-in?(relation,key)
   set assoc-all?(relation,key) //returns a set of values

   commands
   relation' add(relation,key,value)
   relation' remove(relation,key,value)

Domain Restrictions
   remove(relation,key,value)
      defined provided that is-in?(relation,key,value) is True

Equivalences
   assoc-all?(make-relation(),key) = make-set()
   assoc-all?(add(relation,key,value),key)
      = add(assoc-all?(relation,key),value)
   assoc-all?(add(relation,key,value),key1)
      = assoc-all?(relation,key1)
   assoc-all?(remove(relation,key,value),key)
      = remove(assoc-all?(relation,key),value)
   assoc-all?(remove(relation,key,value),key1)
      = assoc-all?(relation,key1)
   key-is-in?(relation,key) = [not(empty?(assoc-all?(relation,key)))]
```

Figure 2.7: Object behavior specification for relations

2.2.2.4 Functions

A function is a relation for which at most one value is associated with any key. In the specification of function, the command add inherited from relation is hidden.

```
Function[entity] inherit from relation

constructor
    function make-function()

queries
    value assoc?(function,key)

commands
    function replace(function,key,value)

hidden
    function add(function,key,value)

Domain Restrictions
    assoc?(function,key) = defined provided that is-in?(function,key) = T

Equivalences
    is-in?(replace(function,key,value),key,value) = T
    is-in?(replace(function,key,value),key,value1) = F
    is-in?(replace(function,key,value),key,value) = F
    is-in?(replace(function,key1,value1),key,value)
        = is-in?(function,key,value)
    key-is-in?(function,key) = F
        ⇒ size?(replace(function,key,value)) = size(function) + 1
    size?(replace(function,key,value)) = size?(function)
    assoc?(replace(function,key,value),key) = value
    assoc?(replace(function,key,value),key1) = assoc?(function,key1)
    assoc?(remove(function,key),key) = undefined
```

Figure 2.8: Object behavior specification for functions

Instead, there is a new command, replace, to enforce the constraint on key-value pairs. However, method add can be used within the implementation of replace.

Table look–up is a common form of function usage. For example, you can look up a table of logarithms for the logarithm of a number or approximate the logarithm using numbers in the table that are close to the given number.

CHAPTER 3

DISTRIBUTED NETWORK COMPUTING AND PVM

3.1 Distributed Computing

Recently, *distributed computing* is becoming a popular way for both high performance scientific computing and more general purpose applications. Distributed computing is running programs across several or many computers on a network. It is fundamentally different from using a single machine.

Distributed computing has many attractive aspects for several reasons. Performance is one of the most important reasons for distributed computing. More computation power, memory and I/O bandwidth can be achieved by connecting a group of machines. Using several workstations together should be cheaper than traditional supercomputers in solving a challenging problem. Availability of machines to be connected is another reason for distributed computing. Usually, there are many workstations available at most universities and industrial companies. However, much of the processing power of those machines is unused. This portion of computing power can be shared for distributed computing. A fault tolerance is a third reason. If a program is running on several machines, it should be able to tolerate a few of them

going down. Distributed computing also enables us to share hardware or software resources. If each user of a single workstation uses his own machine to do things, it is hard to avoid the cost of data duplications. A large database should be shared rather than replicated.

3.1.1 Heterogeneous Network Computing

Heterogeneous, network–based, concurrent computing means a methodology for general purpose parallel computing where [21]

- The hardware platform consists of a collection of multifaceted computer systems of varying architectures, interconnected by one or more network types.
- Applications are viewed as comprising several sub–algorithms, each of which is potentially different in terms of its most appropriate programming model, implementation language, and resource requirements.

Heterogeneous network based computing uses mainly workstations for computing hardware frames. By connecting heterogeneous workstations already existing, we can get higher performance inexpensively.

Heterogeneous network computing refers to models, techniques, and toolkits to match heterogeneous environments on the one hand with complete applications, consisting of different subtasks, on the other. PVM system, which will be discussed later in this chapter, was designed to realize a more general and encompassing interpretation of heterogeneous computing, and had a pragmatic bias aimed at providing

a working system that could be used in existing environments [21]. PVM supports heterogeneous machines, applications, and networks.

3.1.2 Parallel Programming Systems

There are kinds of parallel programming systems that make distributed computing available to application programmers. C–Linda [15], P4 [16], PVM [18], and TCGMSG [19] are those examples. These parallel programming systems are based on the same idea that the nodes of a workstation cluster are made to behave like members of a loosely coupled, parallel computer. There are two kinds of high level protocols that these systems are using:

- Distributed Memory(Message passing)
- Shared virtual memory

PVM is a software system which connects a collection of heterogeneous computers based on a message–passing paradigm. It emulates a single virtual machine of distributed memories. P4 is a library of macros and subroutines developed at Argonne National Laboratory and GMD, for programming a variety of parallel machines. They support both the shared memory model and the distributed memory model. PVM and P4 both use *dynamic TCP sockets*. This means they establish a socket between two communicating nodes at run time when they first communicate with each other. The sockets are generally not reclaimed in the course of a computation. This method has the advantage that it will scale better on a large set of nodes as long as none of

the processors runs out of file descriptors. One disadvantage of dynamic TCP relative to static TCP is that the first communication is significantly slower than subsequent communications.

TCGMSG establishes point to point TCP sockets between every pair of nodes. This is done when the program is initiated and these sockets are not reclaimed in the course of the calculation. This approach is called *static TCP socket* system. The static TCP socket system can run into trouble scaling up to large number of nodes. However, it is fast, simple, and has the advantage of hiding the start up time of establishing connections between all of the nodes.

C–Linda is a concurrent programming model based on the concept of a *tuple–space*, a distributed shared memory abstraction via which cooperating processes communicate. C–Linda employs UDP to communicate directly with the processes involved in the parallel computation. All the processes involved in a computation agree on a UDP port and use it during the computation. C–Linda does not use TCP sockets or an intermediate daemon. Instead, an extra process on each node performs all of the local shared virtual memory management and communicates with the other nodes. This process gains control using a signal handler that initiates a context switch to the tuple space handler.

Craig and *et al.* [20] have studied the performance of these systems using a number of benchmarks. According to [20], the performance of each parallel system is substantially and significantly different for simple communication patterns and

large messages. However, as communication patterns become more complex, the differences in these environments decrease substantially. Therefore, the use of any parallel system does not affect the performance of computation significantly.

3.2 Parallel Virtual Machine

PVM is a software system that connects a collection of heterogeneous computers to be used as a single virtual machine [21, 23]. PVM can be installed on a hardware base consisting of heterogeneous machine architectures, including single CPU systems, vector machines, and multiprocessors. These computing elements may be interconnected by one or more networks(e.g. Ethernet, the Internet, and FDDI). Each computing elements interconnected by a different network is called a `host` and the whole system connected by PVM is called the `virtual machine`. Figure 3.1 [22] shows an architectural overview of the PVM system.

PVM provides a set of library routines which can be called from programs written in FORTRAN, C, or C++. Facilities are provided for creating processes on different machines, communicating between processes, and synchronizing processes. Applications suitable for PVM have subtasks with a moderately large level of granularity. PVM is a loosely coupled distributed memory computer. It allows good use to be made of underutilized workstations to solve problems which are computationally demanding for a single machine.

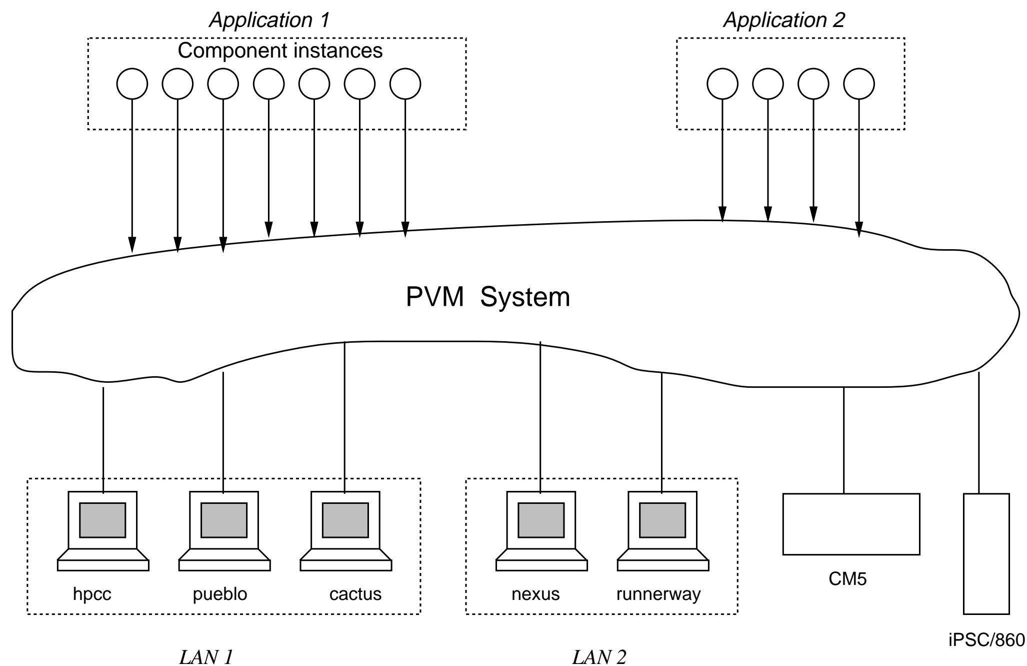

Figure 3.1: PVM architectural Model

3.2.1 Computing Model of PVM

PVM connects a user defined collection of serial, parallel, and vector computers as a large distributed–memory computer. As mentioned earlier, virtual machine means this logical distributed–memory computer, and host refers to one of the member computers. Multiple users can configure overlapping virtual machines, and each user can execute several PVM applications simultaneously. PVM supplies the functions to automatically start up tasks on the virtual machine and allows the tasks to communicate and synchronize with each other. A task is defined as a unit of computation in PVM similar to a Unix process. It is often a Unix process, but not necessarily so. Applications can be parallelized by using message–passing constructs common to

most distributed–memory computers. By sending and receiving messages, multiple tasks of an application can cooperate to solve a problem in parallel.

The model assumes that any task can send a message to any other PVM task, and that there is no limit to the size of or number of such messages. The PVM communication model provides the following message–passing methods:

- Asynchronous blocking send: returns as soon as the send buffer is free for reuse regardless of the state of the receiver.
- Asynchronous blocking receive: returns only when the data is in the receive buffer.
- Non–blocking receive: returns immediately with either the data or a flag that the data has not arrived.
- Multicast: to a set of tasks
- Broadcast: to a user defined group of tasks

The PVM model guarantees that message order is preserved between any pair of communicating entities.

3.2.2 Features of PVM

PVM supplies the following features:

- Routines for a user process to register or leave a collection of cooperating processes

- Routines to add and delete hosts from the virtual machine
- Routines to initiate and terminate PVM tasks
- Routines to synchronize with and send signals to other PVM tasks
- Routines to obtain information about the virtual machine configuration and active PVM tasks

Synchronization may be achieved in one of several ways such as sending a Unix signal to another task, or by using barriers. Another method notifies a set of tasks of an event occurrence by sending them a message with a user–specified tag that the application can check for. The notification events include the existing of a task, the deletion of a host, and the addition of a host.

PVM also provides routines for packing and sending messages between tasks. The core communication routines include an asynchronous send to a single task, and a multicast to a list of tasks. PVM transmits messages over the underlying network using the fastest mechanism available: either UDP, TCP on network based on the Internet protocols, or other high–speed interconnects available between the communicating processors. Messages can be received by filtering on source or message tag, with either blocking or non–blocking receive routines. A routine can be called to return information about received messages such as the source, tag, and size of the data. Message buffers are allocated dynamically, thereby permitting messages limited in size only by native machine parameters.

CHAPTER 4

IMPLEMENTATION OF PARALLEL CONTAINERS

4.1 Overall Architecture

4.1.1 Coordination of Elements

There are two types of processing elements, `coordinators` and `processors`, which cooperate to realize the list–like behavior shown in Figure 4.1. Active processors store data and their coordinator's ID(myboss) to maintain relationship in the container. A coordinator maintains a list of its elements(processors) and their sizes. A coordinator also receives commands and queries from external objects. When it receives external commands or queries, the coordinator distributes these(command and queries) to its elements. Newly activated processors are "tuned" to listen to their current "boss". A remove command results in the deactivation of a processor enabling it to return to pool of inactive processors. The interactions between a coordinator and its processors are shown in Figure 4.2.

Data stored in processors are distributed when processors are created through the coordinator. The coordinator also send out its process ID along with data to each active processors to enable interaction. Processors store data received from the

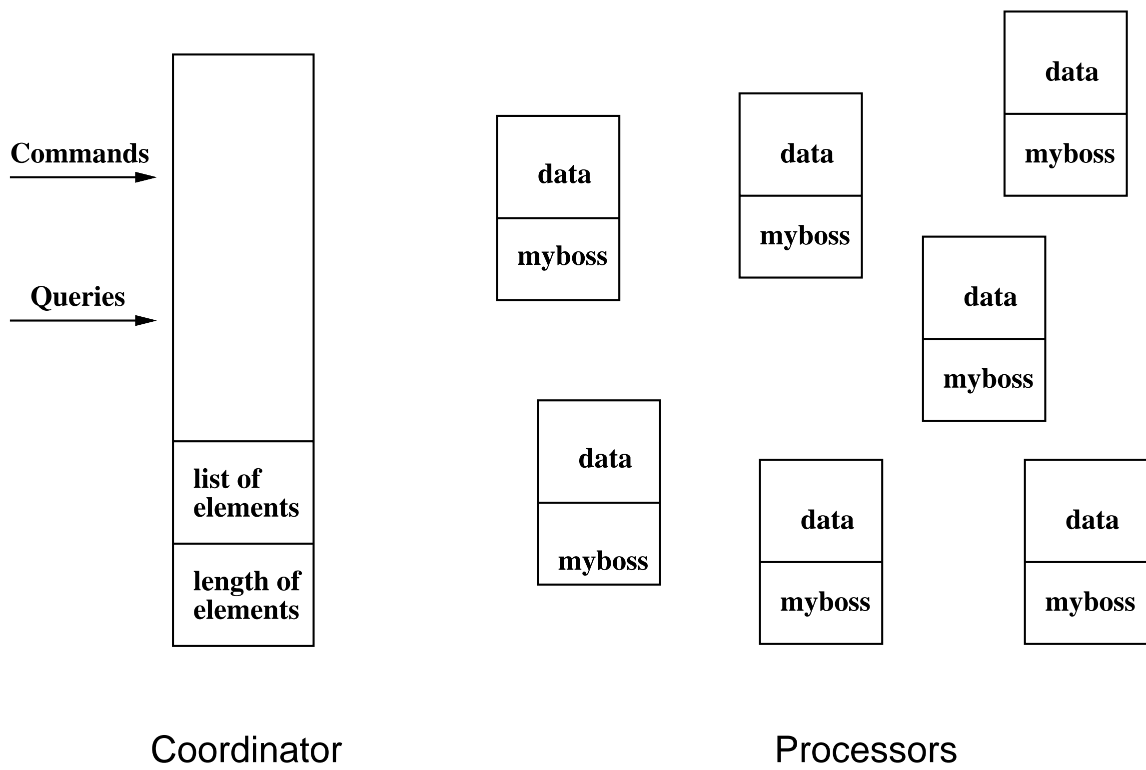

Figure 4.1: Massively Parallel Implementation of Lists

when the coordinator receives commands:
  it broadcasts the commands and executes the next command
  elements execute the commands in parallel upon receiving the commands

when the coordinator receives queries:
  it broadcasts the queries and waits for the responses from the elements
  elements execute the queries in parallel and return responses

Figure 4.2: Interactions between a coordinator and processors

coordinator and process ID of the coordinator until they are deactivated. Processors can belong to multiple coordinators.

4.1.2 Design Considerations

To implement a parallel distributed container, there are several issues to be considered. Transparency is the first issue to be considered for implementing a parallel container. Other issues such as heterogeneity, hierarchical construction, and multiple occurrences have been discussed in previous chapters. They are summarized as below:

- **Transparency**: Primitives should be constructed so that users can use those primitives without knowing the hidden mechanisms.
- **Heterogeneity**: There are two kinds of heterogeneities to be achieved: *computing resource heterogeneity* and *object heterogeneity*. In massively parallel computing environments, large–scale computing capability is required. We should be able to use all available computing resources to complete a large problem

regardless of the model, make, and hardware of the resources. This constitutes computing resource heterogeneity. The other, an object heterogeneity enables instances of different objects to be incorporated into a single container.

- **Hierarchical Construction**: An instance of container classes can hold other instances of containers or entities in it as elements. i.e., enable the construction of a container of containers.
- **Multiple Occurrences**: The same instance of an object can occur in one or more containers at the same time. For example, object A can belong to container C1 and C2.

4.1.3 Architecture of Parallel Container

The parallel version of a container is based on the sequential version of a container that has been implemented in C++ [1]. The sequential version of container class is a linked–list based on a `pointer` to an object as indicated in Section 2.1. However, the parallel distributed container is constructed on a virtual machine (a collection of heterogeneous workstations that are connected with PVM). This system has a distributed virtual memory architecture. Therefore, a pointer to a memory location has no meaning in this environment. A *message–passing* paradigm is useful for solving this problem. All information that any object is holding must be packaged within a message format to be delivered to another object.

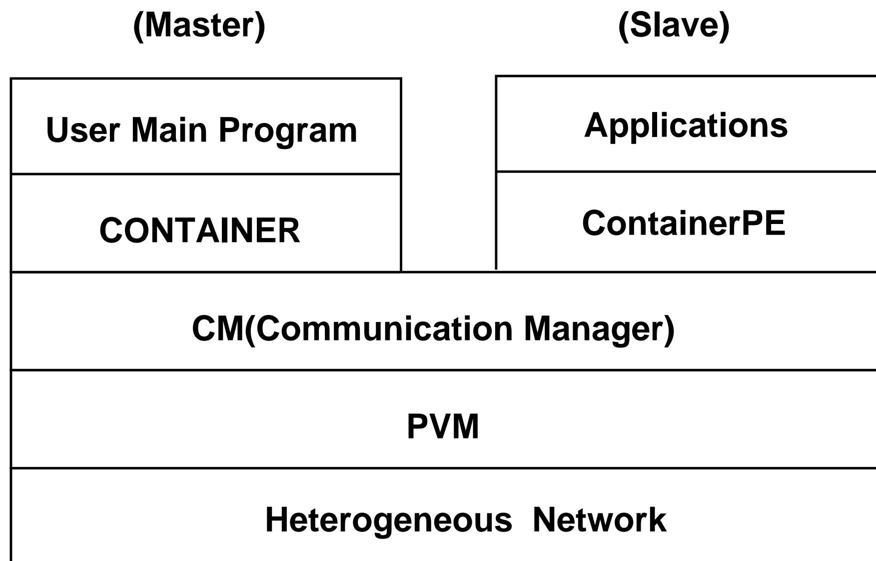

Figure 4.3: Architecture of Parallel Containers

There are several layers that makes up the parallel distributed container, as shown in Figure 4.3. PVM connects heterogeneous computing resources and provides primitives for process controlling and message passing between those processes. The Communication manager(CM) is designed to hide PVM primitives. It provides the interface between PVM primitives and parallel distributed container classes. PVM can be replaced by any other parallel systems due to the existence of CM.

Parallel container classes reside above the class CM. The parallel container consists of two groups of classes: CONTAINER and ContainerPE. Since a parallel container is the extension of a sequential container, the class hierarchy of a parallel container is almost the same as that of the sequential version.

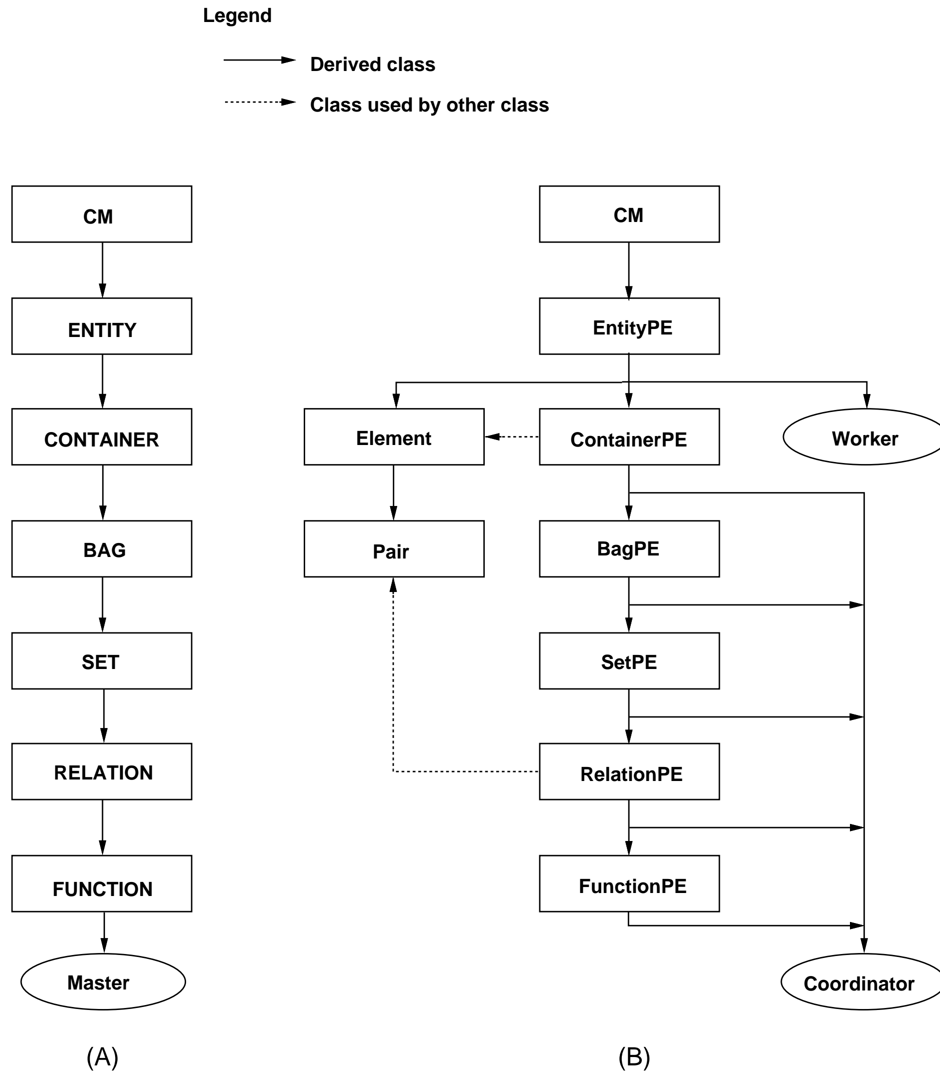

Figure 4.4: Class Hierarchy of Parallel Container

Figure 4.4(A) shows the class hierarchy of class CONTAINER. The CONTAINER class instantiates a master process to control the entire structure. Class CONTAINER is used as a user interface to the processes and a message delivery to containerPE. The class ContainerPE is used for creating slave processes for parallel computations. A containerPE can be either a coordinator or a simple processing element (PE) as explained in Section 4.1.1. While the coordinator inherits attributes and methods from the containerPE class or its derived classes, the PE inherits attributes and methods from entityPE class. This PE is called "worker" since it performs the actual computations.

We use capital letters to differentiate the class CONTAINER from its sequential version counterpart. The parallel container creates a process every time the user creates an object. This can be wasteful in terms of memory usage. Therefore, the sequential version of container may have to be used, when developing application programs. This allows a user to handle objects in two ways:

- A user can create objects without assigning processes to keep only information to be used for computations by the use of the sequential container, named *container*. These objects do not perform parallel computations.

- A user can assign processes to objects by using the parallel container implementation. These objects perform parallel computations.

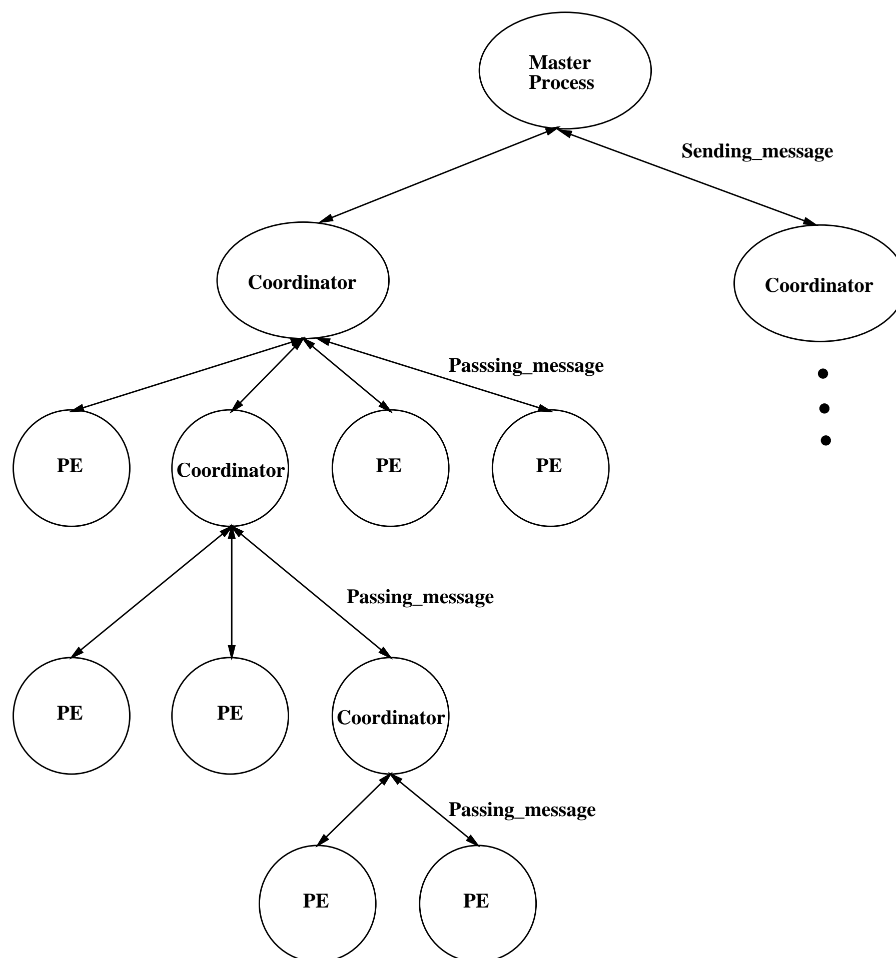

Figure 4.5: Hierarchical construction of objects in parallel containers

As shown in Figure 4.5, the parallel container usually constructs a tree structure for executing computations. Each of the objects has its own memory and works individually. All the commands and queries are exchanged by message passing.

After the objects are created, they are waiting for some instructions to be executed. This is based on `while--loop` implementation. At the top of the the `while--loop`, a method for receiving messages is executed. This method is implemented in blocking mode. i.e., every object is blocked until some command is received. If a command or query is received, the object executes that command. This `while--loop` continues until the object gets an 'end' message from the boss. The boss can be a master process or a coordinator.

This parallel distributed container implementation supports two kinds of computation models: SIMD(single instruction and mutiple data) and MIMD(multiple instruction and multiple data).

4.2 Implementation

4.2.1 Communication Manager

This section discusses the CM(Communication Manager) class. Class CM is the most basic class for communication between objects. Every class in parallel containers inherits its communication methods from this class.

Class CM is based on PVM message–passing methods. This class provides primitives for initiating processes, passing messages, receiving messages, terminating processes, etc. The CM class is designed to hide PVM commands inside the class definition enabling users to use simple primitives easily without having any knowledge of the underlying communication tool. PVM can be replaced with any other parallel programming languages which uses the message passing paradigm by changing the CM class.

PVM enables the use of heterogeneous resources. PVM itself provides the methods that enable us to connect heterogeneous as well as homogeneous workstations to be used as a virtual multiprocessor machine with distributed memory.

4.2.1.1 Enrolling in and Exiting from PVM

```
int CM::CM()
{
  myTID = pvm_mytid();     // Enroll in PVM
}

void CM::shutdown(void)
{
  pvm_exit();              // Exit from PVM
  exit(0);
}
```

Figure 4.6: Methods for enrolling in and exiting from PVM

Figure 4.6 shows the methods for enrolling in and exiting from PVM. The constructor of class CM initiates a PVM task process with the PVM method pvm_mytid(). This method enables a process to enroll in PVM so that the process can set up a connection with the PVM daemon. This method returns a PVM task ID if the enrollment is successful. Otherwise, it returns a negative integer. The returned TID is stored in a private member variable, `myTID`, and this ID is used as the unique PID during the life time of the process. The method `shutdown` removes the process from PVM.

4.2.1.2 Spawning and Terminating Processes

The parallel distributed container is based on the idea that each object has its own process or memory section. In this implementation, a new process should be assigned to each object whenever an object is created. Therefore, constructors in class definition of all container classes use this method to assign a process to each object. Class CM provides this method which creates a new process as an object is created.

Two options are provided by the CM to create new processes as shown in Figure 4.7: (a)Spawning processes on the machine that the PVM daemon selects, (b)Spawning processes on the machine specified by the user. PVM provides three task spawning options:

- **PvmTaskDefault**: spawning a task on an appropriate machine that PVM daemon selects
- **PvmTaskHost**: spawning a task on a host that user specifies
- **PvmTaskArch**: spawning a task on a host which has specified architecture

```
int CM::new_proc(char *pname, char *where)
{
  int id;
  int status;

  if(where == NULL) {
    status = pvm_spawn(pname,NULL,PvmTaskDefault,NULL,1,&id);
    if((status == 0) {
       fprintf(stderr,"Error in spawning tasks...\n");
       return status;
    }
  }
  else {
    status = pvm_spawn(pname,NULL,PvmTaskHost,where,1,&id);
    if((status == 0) {
      fprintf(stderr,"Error in spawning tasks...\n");
      return status;
    }
  }
  return id;
}
```

Figure 4.7: Method for spawning a new process

If the computation size is small, the first option can be used for parallel computations. If not, however, the load of computation should be distributed over the hosts

1	2	3	4	5	6	7

1: command or query
2: subcommand or subquery
3: name of object
4: name of pair object
5: process id of coordinator
6: process id of worker
7: process id of pair

Figure 4.8: Control message format

connected according to their CPU speed. The second option is necessary for this load distribution.

The `new--proc` method uses the pvm_spawn function to create a new process. This function spawns tasks on hosts as long as each host has the ability to handle another process. When memory is no longer available or there is an error in spawning a task, this function fails to spawn a new process and returns a zero.

To terminate a task process, an 'end' message is used. When a coordinator receives the 'end' message, it broadcasts the message to its elements and shuts down its own process. Workers exit their processes when they receive an end message.

4.2.1.3 Passing Messages

As mentioned before, since the parallel container class is implemented on a distributed memory architecture, all the commands and queries are delivered by messages. Therefore, a message is an essential part in this parallel container class implementation.

```
void CM::send_msg(int dst)
{
  pvm_initsend(PvmDataDefault);
  pvm_pkstr(message.query);
  pvm_pkstr(message.subquery);
  pvm_pkstr(message.ent_name);
  pvm_pkstr(message.value);
  pvm_pkint(&message.coord,1,1);
  pvm_pkint(&message.worker,1,1);
  pvm_pkint(&message.newid,1,1);
  pvm_send(dst,Mtype_msg);
}

void CM::recv_msg()
{
  pvm_recv(-1,Mtype_msg);
  pvm_upkstr(message.query);
  pvm_upkstr(message.subquery);
  pvm_upkstr(message.ent_name);
  pvm_upkstr(message.value);
  pvm_upkint(&message.coord,1,1);
  pvm_upkint(&message.worker,1,1);
  pvm_upkint(&message.newid,1,1);
}
```

Figure 4.9: Methods for sending and receiving messages

This parallel container class uses two kinds of messages: one is control message and the other is general message. A control message is used for controlling processes such as adding elements, removing elements, referring elements, and so on. The format of control message is described by Figure 4.8.

Field 1 and 2 are strings which contain commands or queries. The command field is used for packing commands or queries which are to be executed by the processing elements. If a command or query is an ensemble method, a subcommand or a subquery which is the name of a function that will be executed by processing elements should be packed in a message. This is because ensemble methods always take functions as arguments. The message format does not include those arguments that ensemble methods require. Arguments of the function of the ensemble methods are delivered by general messages which deliver integers, floating point numbers, or character strings. This will be explained later. Field 3 and 4 contain the names of objects. Each object has its name to distinguish itself from others. Field 5 through 7 include PIDs.

Figure 4.9 shows the methods for sending and receiving messages. The method `send_msg` packs each field of the message format into one PVM message and sends out the message to the destination process. Upon receiving the message, the destination process unpacks it into the message format with the method `recv_msg`.

General messages are used for sending and receiving arguments for computations or results of computations. All messages except for control messages can be covered by general message delivering methods. Methods for general message passing are as follows:

- send_int and receive_int
- send_float and receive_float

- send_string and receive_string

Figure 4.10 shows the methods for sending and receiving integers. Sending and receiving the other types of general messages, float and string, are also implemented analogously. A message can be sent to a single process or multiple processes. Point–to–point communication is used for sending a message to a single process and the multicasting method is used for sending a message to a collection of processes.

```
void CM::send_int(int dst, int *val, int len)
{
  pvm_initsend(PvmDataDefault);
  pvm_pkint(&len,1,1);
  pvm_pkint(val,len,1);
  pvm_send(dst,Mtype_int);
}

void CM::recv_int(int src, int *val)
{
  int len;

  pvm_recv(src,Mtype_int);
  pvm_upkint(&len,1,1);
  pvm_upkint(val,len,1);
}
```

Figure 4.10: Methods for sending and receiving integers

4.2.2 Parallel Containers

4.2.2.1 The Object Identity

In the sequential version of container classes [1], an object identity is determined by its name and object pointer. To check object equality, we need to compare the name and the object pointer. This implementation is only possible on a single process. An object pointer is not able to move around between the processes.

Since parallel container classes are implemented on a heterogeneous distributed memory architecture, the object pointer identity is no longer effective in this implementation. To distinguish the identity of each object, the task ID (process ID) which is provided by the PVM daemon [23] can be used instead of an object pointer. The task ID is always unique from the start of PVM daemon to the end of it.

4.2.2.2 Constructor

The constructor in each class definition creates an object on a local process on which the master process is running, and it also creates a process which is the counterpart of the object on a host connected by PVM. Each object stores a name and a process ID for identification purposes. The process ID that an object ENTITY holds is the task ID of a PVM task process that is assigned to the object.

Figure 4.11 shows a constructor for an ENTITY object. There are two kinds of constructors: one is for creating an object on the host that PVM daemon selects, and the other is for creating an object on the user specified host. In the sequential

container class, each object is identified by its name and object pointer. However, the process ID is the unique ID of an object in the parallel version of container class.

```
ENTITY::ENTITY(char *NAME)
{
   name = new char[strlen(NAME)+1];
   strcpy(name,NAME);
   set_mypid(new_proc(NAME,NULL));
   strcpy(message.ent_name,NAME);
   send_msg(MyProcID);
}

ENTITY::ENTITY(char *NAME, char *where)
{
   name = new char[strlen(NAME)+1];
   strcpy(name,NAME);
   set_mypid(new_proc(NAME,where));
   strcpy(message.ent_name,NAME);
   send_msg(MyProcID);
}
```

Figure 4.11: Constructor of Class ENTITY

4.2.2.3 Adding Elements

Figure 4.12 shows the methods for adding elements to a container. Method CONTAINER::add() is for preparing and sending a control message to a containerPE. This method takes an object ENTITY as an argument. The object ENTITY keeps its name and process ID that was assigned when it was created. When CONTAINER

executes the add() method, it extracts information that identifies the object and packs that information into the message format. It then sends the message that has been prepared to a coordinator. This coordinator inherits a containerPE class.

```
void CONTAINER::add(ENTITY *ent)
{
    strcpy(message.query,"add");
    if(ent->get_name() != NULL)
      strcpy(message.ent_name,ent->get_name());
    else strcpy(message.ent_name,"");
    message.coord = myID();
    message.newid = ent->get_mypid();

    send_msg(get_mypid());
}


void containerPE::add()
{
  entityPE *ent = new entityPE;
  ent->set_mypid(message.newid);
  if((strcmp(message.ent_name,"")) == 0) { ent->set_name("coord"); }
  else { ent->set_name(message.ent_name); }

  add_element(new element(ent));
}
```

Figure 4.12: A method for Adding an element

After the coordinator receives the message from the master, it creates a local object which will hold the information that is contained in the message received.

This object is actually a copy of the object that has been created on the master process.

```
class element: public entityPE {

  private:
    static char *classname;

  protected:
    entityPE *ent;
    element *right;

  public:
    element() { ent = NULL; right = NULL; }
    element(entityPE *ENT) { this->ent = ENT; right = NULL; }
    char *get_classname();
    element *get_right();
    entityPE *get_ent();
    void set_right(element *el);
    char *get_name();
};
```

Figure 4.13: The class definition of an element

To hold elements in the coordinator as a linked–list structure, a class `elements` is employed. Figure 4.13 shows the class definition of elements. The way to link the next element is almost the same as a conventional list structure. There are two protected instance variables: `ent`, a pointer to an `entityPE` to be wrapped in this class and `right`, a pointer to link the `element` on the right of this element.

The method `add` calls upon several private commands as shown in Figure 4.12. It creates a new element to hold its argument, entityPE. Elements are always added at

the head, by the helper method `add_element`, it treats empty and non–empty cases distinctly as shown in Figure 4.14.

```
void containerPE::add_element(element *el)
{
  if(empty())
    add_first(el);
  else
    add_at_head(el);
}

void containerPE::add_first(element *el)
{
  set_length(1);
  set_head(el);
}

void containerPE::add_at_head(element *el)
{
  increment_length(1);
  el->set_right(head);
  set_head(el);
}
```

Figure 4.14: Helper methods for adding elements

4.2.2.4 Referring Elements

Methods `is_in` and `length` are used for querying the elements in a container. The method `is_in(ENTITY *ent)` enables scanning of the elements in a container.

As shown in Figure 4.15, a coordinator compares a given PID with those of its elements. If there is an element that has the same PID as the given PID, it returns TRUE, and otherwise returns FALSE. The implementation of this method

looks somewhat sequential. However, since the coordinator should have PIDs of its elements to maintain communication connection between them, it can use that list for simple queries. This reduces the amount of communication and consequently enhances overall performance.

```
Boolean CONTAINER::is_in(ENTITY *ent)
{
  Boolean ans = F;

  strcpy(message.query,"isin");
  strcpy(message.ent_name,ent->get_name());
  message.coord = myID();
  message.worker = ent->get_mypid();
  send_msg(get_mypid());
  recv_int(-1,(int *)&ans);
  return (Boolean) ans;
}

void containerPE::is_in()
{
  Boolean ans=F;

  for(element *p=head; p!=NULL; p=p->get_right())
    if((p->get_ent()->get_mypid() == message.worker) &&
     ((strcmp(message.ent_name,p->get_ent()->get_name())) == 0) ans = T;

  send_int(get_myboss(),(int *)&ans,1);
}
```

Figure 4.15: The method for asking existence of an ENTITY

4.2.2.5 Ensemble Methods

As discussed in Section 2.4, *ensemble* methods are essential for implementation of parallel distributed container classes. There are kinds of ensemble methods defined:

- **tell_all** command args: send the command(args) message to all objects in the container
- **ask_all** query? args: send the query?(args) message to all objects in the container and collect the results in a returned container
- **which?** query? args: send the query?(args) message to all objects in the container and collect objects returning TRUE in a new container

Basically, the concept of ensemble methods is almost the same as that of iterators discussed in Section 2.1. One difference is that while ensemble methods are idealized for the use in the parallel containers, iterators are inherently sequential.

Figure 4.16 shows the C++ implementation of `ask_all`, one of the emsemble methods. Each worker has its own method `eval` and a coordinator has a method which handles `ask_all` commands. Since arguments of ensemble methods are variable based on the function used, this method should be tailored for each case.

The implementations of `tell_all` and `which?` are almost the same as that of `ask_all` except for returning values. The argument function of method `which?` is a boolean function. Therefore, the results of evaluation should be either TRUE or FALSE. If an object returns a TRUE as the result of evaluation, this object is added

to a newly created container. `Tell_all` does not require a response from the objects; it just changes the state of each object.

```
CONTAINER *CONTAINER::ask_all(char *q, ...)
{
  va_list args;
  char result[1024];

  strcpy(message.query,"askall");
  message.coord = myID();

  if((strcmp(q,"eval")) == 0) {
    va_start(args,MaxArg);
    va_end(args);

    strcpy(message.subquery,q);
    send_msg(get_mypid());
    send_args();
    CONTAINER *C = new CONTAINER;

    for(int i=0; i<length(); i++) {
      recv_str(-1,result);
      ENTITY *E = new ENTITY(result,NULL);
      C->add(E);
    }
  }
  return C;
}
```

Figure 4.16: An example of ensemble method implementation

4.2.3 Derived Classes

4.2.3.1 Class Bags and Class Sets

```
void BAG::remove(ENTITY *ent)
{
  while(is_in(ent)) {
    strcpy(message.query,"remove");
    strcpy(message.ent_name,ent->get_name());
    message.coord = myID();
    message.worker = ent->get_mypid();

    send_msg(get_mypid());
  }
  if(this->get_length() == 0) this->end();
}

void bagPE::remove()
{
  for(element *p=head; p!=NULL; p=p->get_right()) {
    if(((strcmp(message.ent_name,p->get_ent()->get_name())) == 0) &&
        (message.worker == p->get_mypid())) {
      strcpy(message.query,"end");
      send_msg(message.worker);

      remove_element(p);
     }
  }
}
```

Figure 4.17: A method for removing an element

Class bags add the capability to query for the number of occurrences of an element and the removal of an element. The implementation of the query for the number of occurrence is based on the element scanning mentioned in the previous section. Upon

receiving the query, a coordinator scans an element list and counts the number of elements which match the condition that a user has requested.

The removal of an element is first introduced in class bags. Figure 4.17 shows the methods implemented in C++. This method is also based on element scanning. A coordinator scans elements until an element to be deleted is met, and then removes the element by sending a message to the corresponding object process. Finally the coordinator updates its element list.

Class sets are almost same as class bags except for the ability to maintain multiple occurrences. The class sets can be created by modifying the add method in class container. The add method needs the check for the occurrence of an object before adding it to the set.

4.2.3.2 Class Relations and Class Functions

As mentioned in Section 2.2, class relations implement a binary relation between two objects. Thus a pair of objects has a certain relation and they cooperate with each other. Class relations keep each element as a pair of objects. Class pairs are employed to enable class relations to hold elements as pairs. Figure 4.18 shows the class definition of pairs.

One of the related objects has `key` as its name and the other has `value`. The object which has key plays semi-coordinator roles. When the key object receives a certain message it delivers the message to its pair and cooperates with a value object.

Class functions hold elements in the same way as class relations do except for one restriction. The restriction is that pairs of objects which have the same `key` objects can not be added as the elements of class functions.

```
class pair : public element {

  private:
    static char *classname;

  protected:
    entityPE *value;

  public:
    pair() { value = NULL; }
    pair( entityPE *key, entityPE *VALUE ) : element(key) {
       this->value = VALUE;
    }
    char *get_classname();
    entityPE *get_value();
    Boolean equal(entityPE *ent);
    char * get_name();
};
```

Figure 4.18: The class definition of pairs

4.2.4 An Overview of Interaction between Objects

This section discusses the creation and addition of objects to a container object, and their interactions for computational purposes. Several steps should be taken to perform real computations with this parallel distributed container class. The first step is creating an object. An object can be either a coordinator or a worker(PE). As

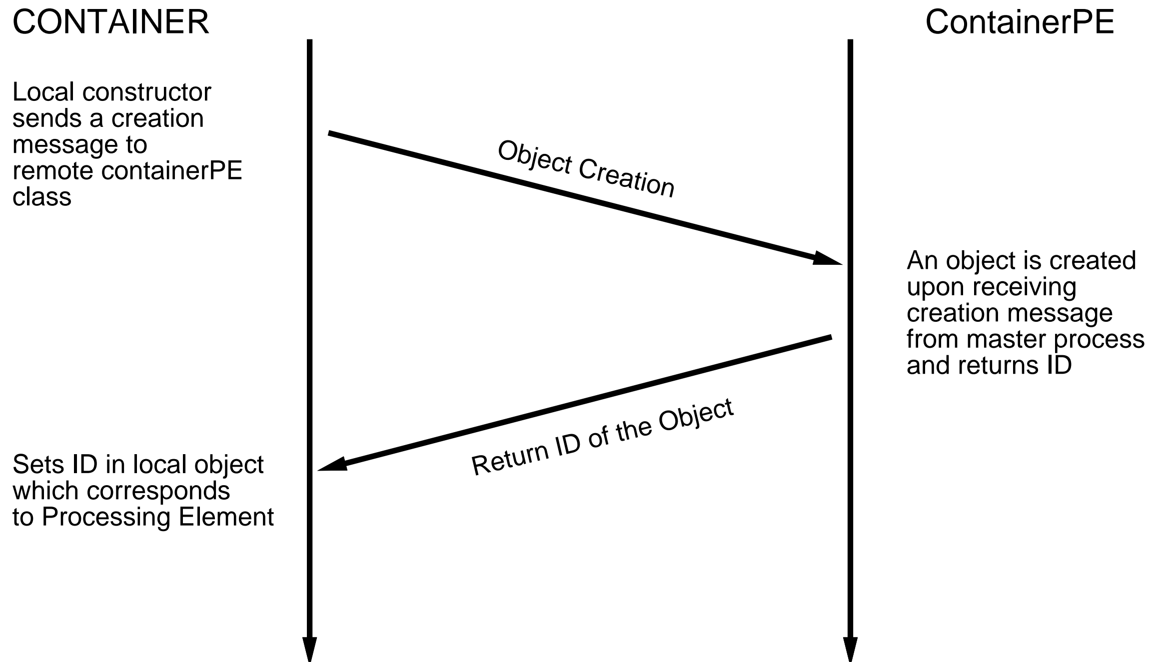

Figure 4.19: Object Creation

discussed earlier, a coordinator should inherit class containerPE or its derived classes, and a worker should inherit just entityPE class. Objects can be created on any hosts which are node members of a virtual machine. After a coordinator is created, it is able to hold other objects by the method add.

To create an object, a master process (which is an instance of class CONTAINER) sends a creation message to containerPE as shown in Figure 4.19. The procedure for creating an object is as follows: First, class CONTAINER creates a local object on the host where class CONTAINER is residing. This local object is used for holding the information about the corresponding process to be created. And then class CONTAINER sends an object creation message to class containerPE. This

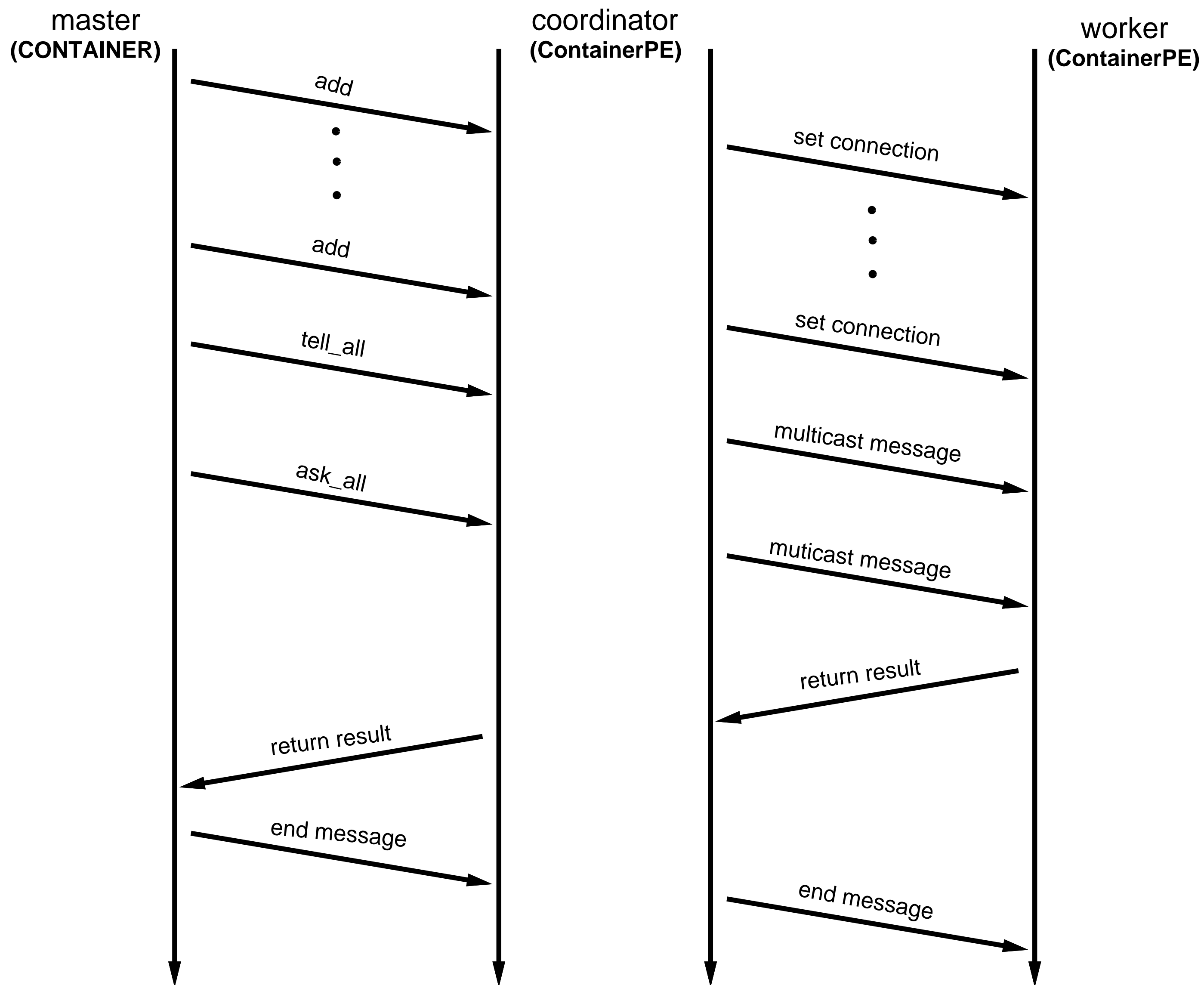

Figure 4.20: Interaction between objects

message spawns a task process on which a directed program is running. The created process returns its PID to class CONTAINER, and class CONTAINER sets PID to a corresponding local object. Class CONTAINER handles remote objects with corresponding local objects after creating objects.

As mentioned earlier, all the messages for controls and computations should travel through the coordinator. A coordinator receives incoming messages from master process and delivers them to worker PEs and vice versa. A coordinator can also perform evaluation functions as worker PEs. If a coordinator is assigned to a computation, it also performs that computation besides the message delivering jobs.

After being created, objects should be organized to construct a structure that is desirable to the user. This is achieved by the method `add`. Adding an object to a coordinator is setting a communication connection between the coordinator and the object. A process assigned to a coordinator creates a local object entity to keep the information about element object. This local object on the coordinator is a copy of the local object made on the master process. After finishing a construction of the desired structure, a user can apply ensemble methods to execute real computations. The real computations are usually done by methods `tell_all`, `ask_all`, and `which?`. Figure 4.20 shows the interactions between objects briefly.

CHAPTER 5

AN EXAMPLE OF REAL APPLICATION: A Parse Tree

5.1 The Simulation of Watershed Model

The NSF Grand Challenges Grant, "Massively Parallel Simulation of Large Scale, High Resolution Ecosystem Models",(ASC-9318169) is an ongoing project which seeks to:

1. construct a computer modeling and simulation environment that employs massively parallel processing to simulate interactions of ecosystem processes at selectable scales of space and time.

2. integrate as intrinsic to the environment, GIS (Geographical Information System) databases and discrete event cellular models as the descriptions of the 3-dimensional landscapes to accommodate spatial heterogeneity and necessities of scale.

3. support experimentation and interpretation through scientific visualization.

The objective of this project is modeling and simulating landscape and ecosystem to monitor and predict the behavior of ground water.

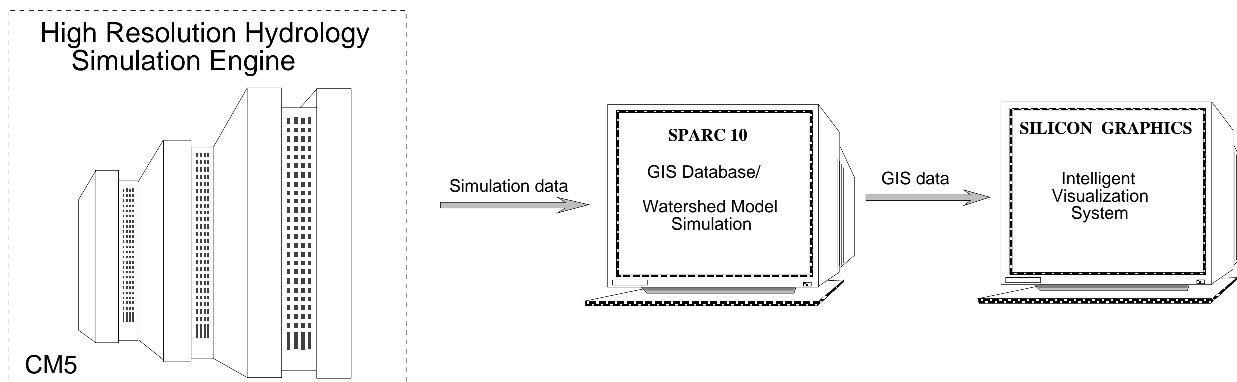

Figure 5.1: Architecture for Distributed Simulation Environment

Figure 5.1 shows the architecture for the distributed simulation environment for the landscape and ecosystem simulation and visualization. The simulation engine is currently using the Connection Machine Model-5 which is a massively parallel computing resource. This may be switched with a cluster of workstations environment using a parallel distributed container classes.

GIS database contains spatially referenced data in many different themes representing quantitative, qualitative, or logical information. This information is used for a cellular model of landscape which will be explained later on. The SUN Sparc-1000 workstation is assigned to GIS database server. Since this project is the large scale ecosystem simulation, an interactive visualizer is needed to show the results of simulation and interpret those results in different ways. The visualization system displays the dataset retrieved from either the GIS database server or simulation engine.

Figure 5.2 is the cellular space representation of a watershed model. As shown in Figure 5.2, the surface of the ground is modeled as 3–dimensional cells. The column

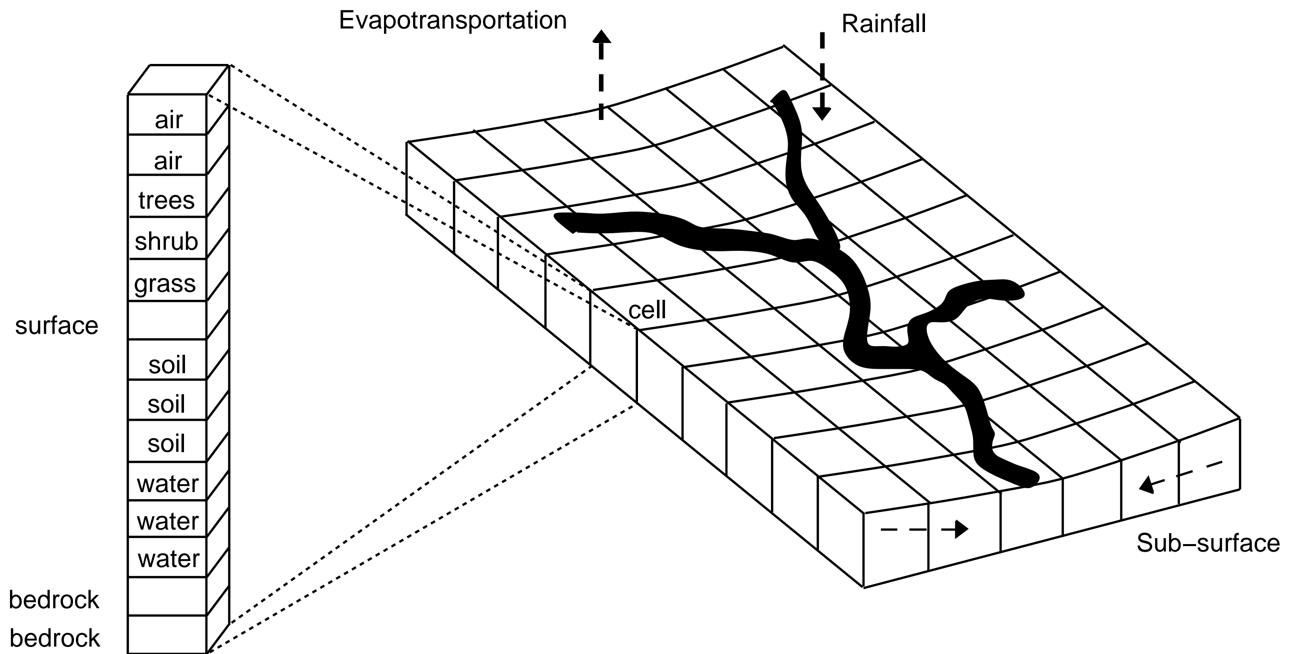

Figure 5.2: Cellular Space Representation of Riparian Ecosystem

of each cell consists of several layers from bedrock to air. Those layers are modeled by DEVS(Discrete Event System Specification) formalism [2, 3] with the information from the GIS database as input data. Each cell is treated as a reservoir which has its own water capacity. Rain is scattered in a specific area and cells in that area get rain as input data. The rain will infiltrate downward through layers in a cell. If the water level reaches the capacity of a cell, it flows to neighboring cells. These neighboring cells get influx water from the rain itself and it flows from neighboring cells. Each cell gets its own input data and computes discrete transition until it meets certain output conditions . Experimental frame [2] will gather output from cells and download to the designated hosts to evaluate and show the results.

5.2 The Role of A Parse Tree

For the visualization, an "Intelligent Visualization System" [30] has been proposed. This system has dynamic control structure and performs overall control functions such as control of simulation, data monitoring, and data management as well as visualization. The visualizer seeks a significant event from the results of simulation for those functions mentioned above.

Data Monitoring System(DMS) [30] is a subsystem of the Intelligent Visualization System to access the data generated by a simulation engine and to detect pre–defined events from the data. To detect the significant event that has been defined, DMS generates a parse tree of a certain event and evaluates the tree. The events are defined by "Event Description Language" [30] which consists of the following items:

- Predicates: LT, LE, EQ, GT, GE
- Functions: MIN, MAX, AVG, TOTAL, STD–DEV
- Connectives: AND, OR, NOT
- Variables
- Constants

The semantic meaning of a significant hydrology event can be defined by the use of the appropriate combination of the predicates and functions.

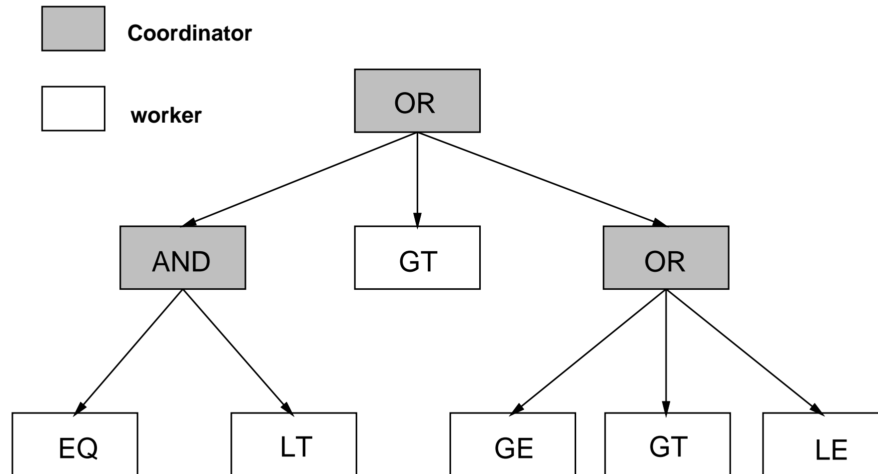

Figure 5.3: An Example of a Parse Tree

If an event description consists of only predicates, functions and connectives, a *parse tree* is generated according to the event description. Figure 5.3 shows an example of a parse tree. Predicates inherit entity objects and play worker roles. Connectives are assigned to coordinator objects which inherit class containerPE. A worker has information to execute simple tasks such as the location of data and the threshold value. Coordinators collect the boolean results from the workers and perform logic operations.

Coordinators use the *ensemble method* in their evaluation functions for the logic operations. Figure 5.4 shows the class definition of the AND connective. As shown in Figure 5.4, `which?`, one of the ensemble methods, is used for the AND operation in a member function `eval`. The `which` method is almost the same as the method `ask_all` shown in Figure 4.16 except for one thing. The difference is that `which` gets only TRUE results while `ask_all` collects all the results from the elements. Both methods

return a new container in which the results are contained. The logic operation AND can be calculated by comparing the size of elements that the coordinator holds with the size of the elements that the returned container holds. If the sizes of them are the same, this means that the result is TRUE. The other logic operations OR and NOT can also be calculated in the same way.

```
class ANDops : public containerPE {
  public:
    ANDops() { }
    void eval() {
      int rep;
      containerPE *ans = which();
      if (ans->size() == this->size()) {
        rep = T;
        send_int(get_myboss(),&rep,1);
      }
      else {
        rep = F;
        send_int(get_myboss(),&rep,1);
      }
    }
};
```

Figure 5.4: Class definition of AND Connective

According to the result of the evaluation of the parse tree, the DMS performs a prepared visualization plan. For example, if the runoff level through the specified area is greater than a certain level, the parse tree will detect this phenomenon as a significant event and will return TRUE value to the DMS controller. Then DMS will execute a predefined data visual plan.

Since the landscape model consists of hundreds of thousands of cells and it makes large result data, data monitoring can be a bottleneck in the overall simulation. Therefore, data monitoring should be fast enough to catch up with the speed of the simulation engine.

5.3 Experimental Results

Our goal of this experiment is to speed up the evaluation of a parse tree by the use of proposed parallel container classes under the distributed computing environment. We prepared an EVENT definition for this experiment as follows:

```
EVENT1:AND[GT(AVG(runoff1),1.32)];

EVENT2:OR[GT(AVG(runoff1),1.32),
          AND[EQ(AVG(runoff1),1.32),LT(AVG(runoff1),3.2),
          OR[GE(AVG(runoff1),1.32), GT(AVG(runoff1),3.2),
             LE(AVG(runoff1),1.32)]];
```

This EVENT definition is used for detecting the runoff of the valley. If the runoff level of the valley is greater than a certain level, the parse tree will return a designated result.

An EVENT is selected according to the EVENT in which the user is interested. At the evaluation stage, the master program reads this EVENT definition file and transforms it to a real parse tree. Figure 5.5 shows how the parse tree is constructed from the EVENT definition. The procedure shown in Figure 5.5 makes the parse tree which has the structure in Figure 5.3. Each node of the parse tree is assigned to an object PE during transformation. There are two kinds of processes as discussed

in Chapter 4: *worker PE* inherits the class entityPE and performs the evaluation of predicates(GE, GT, etc.). It reads the result data file and compares the data with the given threshold value, and then returns boolean value as a result. *Coordinator PE* which inherits class containerPE performs the evaluation of connectives(AND, OR, NOT). It collects the results of the evaluation from its children and computes the logic.

```
main() {
  CONTAINER *top = new CONTAINER("OR");
  ENTITY *gt1 = new ENTITY("GT");
  top->add(gt1);
  CONTAINER *and = new CONTAINER("AND");
  ENTITY *eq = new ENTITY("EQ");
  and->add(eq);
  ENTITY *lt = new ENTITY("LT");
  and->add(lt);
  top->add(and);
  CONTAINER *or = new CONTAINER("OR");
  ENTITY *ge = new ENTITY("GE");
  or->add(ge);
  ENTITY *gt2 = new ENTITY("GT");
  or->add(gt2);
  ENTITY *le = new ENTITY("LE");
  or->add(le);
  top->add(or);
}
```

Figure 5.5: Making a real parse tree from EVENT definition

For the experiments, we connected heterogeneous computers which all have a different architecture as follows:

- nexus.srnr.arizona.edu : Sparc10, SUN Micro Systems
- leopard.ece.arizona.edu : SGI, Silicon Graphics
- cactus.ece.arizona.edu : ALPHA, DEC
- zeta.ece.arizona.edu : ALPHA, DEC
- clipper.cs.arizona.edu : ALPHA, DEC
- brigantine.cs.arizona.edu : ALPHA, DEC

6 workstations are available at our research site for this experiment at this time. Host 'nexus' is Sparc10 machine that has 6 CPU's, and the others have a single CPU. The result data the size of which is 1MB have been generated assuming that 1000×1000 cells are simulated. Experiments have been performed as follows: First, a single worker PE was assigned to each host in the virtual machine and the execution time has been measured to figure out the real computation speed of the machine. And then the number of workers was increased from one to the maximum which can be handled by each host. As shown in Figure 5.6, 25 workers can be assigned to host 'nexus' and 20 workers to the others at the same time. Secondly, the parse tree of 6 worker nodes was generated and each node of the parse tree was distributed over the hosts respectively. Finally, parse trees of larger number of worker nodes were generated and distributed over the hosts according to their computation capabilities, and the total execution time of each case was measured. The first case was measured in serial and the last two cases were measured in parallel.

# node / host	1	2	3	4	5	10	15	20	25
nexus	7.5	7.6	7.6	7.6	7.8	12.8	21.1	26.5	33.7
leopard	7.4	13.5	18.5	25.2	33.2	71.8	106.7	137.9	N/A
zeta	11.5	23.3	31.4	43.6	54.8	106	176.5	264.3	N/A
cactus	12.1	24.6	35.9	49.7	66.4	115.2	194.5	296.7	N/A
clipper	6.0	11.2	17.2	22.6	27.1	54.3	90.2	115.2	N/A
brigantine	12.0	24.4	32.4	42.4	53.5	106.5	173.5	211.6	N/A

Figure 5.6: The Result of Sequential Running (Unit is seconds)

The experiments were performed during only night times to avoid multiuser interference. Since all the hosts described above are multiuser systems the performance of each host can be affected by other users.

Figure 5.6 shows the results of sequential running. The averages are taken on 50 executions for each number of nodes from 1 to 5, 20 executions for each number of nodes from 10 to 25. The execution times of the parse trees on the hosts with single CPU's are roughly proportional to the number of workers in the parse tree except for 'nexus'. Since 'nexus' has 6 CPU's, its computational power is much higher than those of other machines.

The results of parallel running over distributed machines are shown in Figure 5.7. For the parallel running case, the averages are taken on 20 executions for each parse

HOST	6		40		50	
	# node assigned	exe. time	# node assigned	exe. time	# node assigned	exe. time
nexus	1	7.5	25	33.7	25	33.7
leopard	1	7.4	4	25.2	5	33.2
zeta	1	11.5	2	23.3	3	31.4
cactus	1	12.1	2	24.9	3	35.9
clipper	1	6.0	5	27.1	10	54.3
brigantine	1	12.0	2	24.4	4	42.4
Total Exe. time	12.2		36.5		57.2	
Overall overheads	0.1		2.8		2.9	

Figure 5.7: The Result of Parallel Running (Unit is seconds)

tree. The columns indicated by numbers 6, 40, and 50 are the results for the parse trees of those numbers. Each column is divided into two subcolumns. The first subcolunm shows the number of nodes that assigned to each host and the second one shows the average execution time of the nodes assigned on each host. These execution times are taken from the previous results shown in Figure 5.6.

The first column shows the result of the parse tree of 6 worker nodes. A single worker node was assigned to each host respectively. It took 12.2 seconds to evaluate the parse tree of 6 worker nodes. The total execution time is slightly more than the execution time of the slowest machine, 'cactus', which yields 12.1 seconds to evaluate a single worker node. The time difference between these two is 0.1 seconds.

No one host in the virtual machine can handle a parse tree of over 30 nodes. Thus worker nodes of the parse tree should be distributed over the hosts when the parse tree of more than 30 nodes is evaluated. The distribution of the nodes in a parse tree was determined according to the results of sequential running as shown in Figure 5.6. 25 nodes were assigned to 'nexus', 4 to 'leopard', 5 to 'clipper', and 2 to other machines to evaluate the parse tree of 40 worker nodes. The total execution time of the parse tree was 36.5 seconds. As shown in Figure 5.7, the execution time of 25 worker nodes on 'nexus' was the slowest among those of all hosts when the parse tree of 40 worker nodes was evaluated. The result of this evaluation is 2.8 seconds more than that of the slowest machine. The parse tree of 50 worker nodes was also evaluated in parallel. 25 workers were assigned to 'nexus' again, 5 to 'leopard', 10 to 'clipper', 4 to 'brigantine', and 3 to others. The evaluation time was 57.2 seconds. This is 2.9 seconds more than that of the slowest machine, 'clipper', at this time.

The total execution time in parallel depends on the slowest machine because the worker nodes assigned to each host run in locally sequential. The total execution time was slightly more than that of the slowest host. This is because of the overheads. The last row of Figure 5.7 shows the overheads for each evaluation.

As discussed in Chapter 4, object PE's should be created and added to a container to do real computations. Creating objects, adding those object to a container, sending out ensemble methods, and gathering the results may take some time because executing these methods requires passing messages through the network. Our

heterogeneous network used for this experiment is Ethernet. Sending messages over Ethernet is serialized since only one message can be on the Ethernet bus at any time. This fact produces the network delay in passing messages to neighboring hosts.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

In this thesis, the parallel distributed container has been proposed and implemented by the use of C++, an object–oriented language, and PVM, a parallel distributed programming system. Object–oriented programming is a popular way to construct a software system these days and it has a concurrent nature even though languages are sequential.

Some important features of object–oriented language have been reviewed with C++ as the centering figure. They are the concept of class, inheritance, information hiding, data abstraction, late binding, an iterator and a general container class which are main features used to implement our parallel distributed container.

Class containers are well known concept in the object–oriented programming area. A container is an object that holds other objects and enables us to handle those objects conveniently. And yet, many container class libraries have been implemented in many object–oriented languages for general uses but there has been no attempt to realize a container in parallel under the distributed computing environment.

We have proposed to extend the serial version of a container that has been implemented by Dr. Zeigler to a parallel version and have realized it in the distributed heterogeneous computing environment. There were several considerations to achieve in designing and implementing this container such as transparency, heterogeneity, hierarchical construction, and multiple occurrences.

Transparency has been achieved by the use of information hiding in C++. All primitives used in a parallel container have been hidden in class definitions so that a user does not need to aware of the inside of methods. Heterogeneity has two folds: object heterogeneity and computing resource heterogeneity. The former has also been achieved by the use of inheritance in C++. This attribute also enables us to construct a hierarchical structure of class definitions. For the latter, PVM has been employed to utilize a collection of heterogeneous workstation clusters. Mutiple occurrences have also been realized by the use of the concept of object–oriented programming. An object can be added to multiple container objects at the same time and it can work for those bosses respectively.

We have tested this parallel container in the distributed heterogeneous workstation environment with the application of a parse tree in the context of watershed model simulation. As discussed in Section 5.3, large problems like the parse trees of 40 worker nodes or more could be solved by the use of this parallel container. As mentioned before, these parse trees could not be assigned to a single machine because of its limitation of the memory size. We also could speed up evaluating a parse tree

of 40 or 50 worker nodes by distributing workers over the hosts. The communication overhead was produced since the computing resources are located in a different place and connected by network. However, this overhead is trivial compared to the total execution time as the size of computation grows larger.

6.2 Future Work

After finishing our research for this thesis, we may be able to propose some possible future work as follows:

- This system is supposed to be combined with DEVS C++. DEVS has been implemented in C++ recently and is being used for the NSF grand challenge of watershed model simulation on the connection machine CM–5. Combining this system with DEVS C++ would enable us to run the watershed simulation in the heterogeneous computing environment and compare the performance of this system with that of CM–5.

- Dynamic load balancing between hosts is one of the most important issues of heterogeneous distributed computing. For the experiments of this thesis, it was assumed that the load of each host would not change during the execution. However, the load of the workstation may change dramatically because most of the workstations are multi–user systems. Any user can run his program on the host that is connected in the virtual machine and this may affect the total

execution time severely. Therefore, dynamic load balancing is required to get more stable performance.

- A dynamic message polling system should also be implemented for completely asynchronous message passing. Currently, Processing Elements in this system should wait until commands or designated results are arrived. This retards the advance of the master process to the next event. If a dynamic message polling system is added to this system, the master process can handle multiple jobs at the same time.

Appendix A

Parallel Container User's Guide

This appendix explains how to set up the environment for running the parallel container with PVM and how to make the applications with the parallel container classes. It also explains how to compile programs and how to run the applications. Most of the contents are summarized from [23]. You need to read [23] for more detailed information about PVM.

A.1 Setting Up Environment

A.1.1 Obtaining Source Codes

To use the parallel container, you should obtain and install the parallel container software. This software does not require special privileges to be installed. You can obtain the source codes of the parallel container via 'anonymous ftp' from /pub directory of 'prague.ece.arizona.edu'. Following steps show how to get and to unpack the source codes.

```
% ftp prague.ece.arizona.edu
ftp> cd pub
ftp> get pcont.tar.Z
ftp> bye
```

```
% uncompress pcont.tar.Z
% tar xvf pcont.tar
```

This will create a directory called `pvm3` in your $HOME directory.

A.1.2 Installing PVM

Once you have unpacked the directory called `pvm3` in your $HOME directory, then you are ready to build PVM.

First, copy the file `/pvm3/lib/Cshrc.stub` into your `.cshrc` file on the machines you wish to run PVM. The stub should be placed after your path is defined. Then type `make` in your pvm3 directory. This command will build PVM according to the machine type that you are using automatically.

A.1.3 Starting PVM

Starting PVM involves executing either `pvm` or `pvmd3`. Without arguments, these commands start up pvmd3 on the local host. The console(pvm) may be started and stopped multiple times on any of the hosts on which PVM is running, although typically a user starts only one console. The console allows interactive adding and deleting of hosts to the virtual machine as well as interactive starting and killing of PVM processes.

The PVM console prints the prompt

```
pvm>
```

and accepts commands from standard input.

You can also run `pvmd` without the PVM console. The 'hostfile' should be placed in your working directory before you invoke `pvmd`.

```
% pvmd hostfile &
```

This command will run the PVM daemon in the background. This will not work if `pw` is specified in the hostfile because UNIX will expect input(user password) but UNIX can not handle user input to background jobs.

The following steps get around this problem.

```
% pvmd hostfile
```

After the configuration is in place, PVM can be placed in the background by typing control–Z followed by `bg`. To shutdown PVM, type `halt` at a PVM console prompt.

A.1.4 Configuring Virtual Machine

Each user should have their own hostfile which describes their own personal virtual machine. Figure A.1 shows the format of the hostfile. The first host listed must be the computer on which the user will initially start PVM. Blank lines are ignored, and lines that begins with a # are comment lines.

Several options can be specified on each line after the hostname. The options are separated by white space.

- **lo= userid** allows the user to specify an alternate login name for this host.

- **pw** will cause PVM to prompt the user for a password on this host.

- **dx= location_of_pvmd** allows the user to specify a location other than default for this host.
- **ep= paths_to_user_executables** allows the user to specify a series of paths to search down to find the requested files to spawn on this host. Multiple paths are separated by a colon.

```
# configuration file for PVM

leopard.ece.arizona.edu
nexus.srnr.arizona.edu lo=ykcho dx=/opt5/home/ykcho/pvm3/lib/pvmd
cactus.ece.arizona.edu lo=guest dx=/usr/users/guest/pvm3/lib/pvmd
zeta.ece.arizona.edu lo=guest dx=/usr/users/guest/pvm3/lib/pvmd

# set default options for multiple hosts
* lo=ykcho dx=/home/helios2/ykcho/pvm3/lib/pvmd
cygnus.ece.arizona.edu
lyra.ece.arizona.edu
bootes.ece.arizona.edu

# hosts may be added later
&clipper.cs.arizona.edu lo=ykcho dx=~/pvm3/lib/pvmd
&brigantine.cs.arizona.edu lo=ykcho dx=/usr1/ykcho/pvm3/lib/pvmd
```

Figure A.1: An example of the hostfile

If the user wants to set any of the above options as defaults for a series of hosts, then the user can place these options on a single line with a for the hostname field. Hosts that the user doesn't want in the initial configuration but may add later can be specified in the hostfile by beginning those lines with an &.

A.2 Using Parallel Container

A.2.1 Writing Applications

As discussed earlier, the parallel container classes are based on the master/slave model. Therefore, a user should write down two kinds of programs for applications. One is for the master process and the other is for slave processes.

The master program will include following interface files:

- pvm3.h
- vars.h
- CM.h
- ENTITY.h
- CONTAINER.h
- BAG.h
- SET.h
- RELATION.h
- FUNCTION.h

To make the slave programs, you should include following user interface files:

- pvm3.h

- vars.h
- CM.h
- entityPE.h
- elementPE.h
- containerPE.h
- bagPE.h
- setPE.h
- pairPE.h
- relationPE.h
- functionPE.h

You can also use the sequential version of container classes in any application programs.

A.2.2 Compiling and Debugging Source Codes

Application programs that make PVM calls need to be linked with `libpvm3.a`. And all the parallel container classes are needed to be compiled and linked with application programs. Example makefiles are provided in `cont_master` and `cont_slave` directories. These make files show how to link your programs with PVM libraries

and the parallel container classes. After finishing all the application programs and makefile, you can compile applications with following command from UNIX prompt:

```
% aimk file_name
```

This command will compile all application programs and link them with PVM libraries, and put executables into the designated directory(`pvm3/bin`) as directed.

For debugging application programs, you can use any standard serial debugger such as dbx and gdb. You can also use diagnostic print statements for debugging. However, these diagnostic print statements sent to stdout or stderr from spawned processes do not appear on the user's screen. All these prints are sent to a single log file of the form `/tmp/pvml.<uid>` on the host where PVM was initially started. According to my experiences, using diagnostic print statements is better than any other methods for debugging parallel applications.

A.2.3 Running Applications

After finishing all compilations, you can run applications from a UNIX prompt on any of the machines in the configuration.. A whole series of applications may be run on the existing PVM configuration. It is not necessary to start a new PVM for each application. However, you need to reset PVM if an application crashes.

There are several testing programs available:

- In `cont_master`: testc.C, testb.C, tests.C, testr.C, testf.C

- In `cont_slave`: worker.C, cont_coord.C, bag_coord.C, set_coord.C, rel_coord.C, func_coord.C

Follow each step of following instructions for testing files:

```
In cont_master directory
 % aimk tst
 % cd ../cont_slave
 % aimk worker
 % aimk coord
 % cd ../cont_master
 % pvm (after pvm> prompt appears, type quit)
 % ctest
 % btest
 % stest
 % rtest
 % ftest
```

REFERENCES

[1] B. Zeigler, *Objects and Systems*, Springer-berlag, To be published, 1995.

[2] B. Zeigler, *Multifceted modeling and discrete event simulation*, Academic Press, 1984.

[3] B. Zeigler, *Object–Oriented Simulation with Hierarchical, Modular Models: Interligent Agents and Endomorphic Systems*, Academic Press, 1990.

[4] B. Stroustrup, *The C++ Programming Language*, Second Edition, Addison–Wesley, Reading, Masachusetts, 1991.

[5] B. Meyer, *Eiffel: The Language*, Prentice Hall, New York, 1992.

[6] G. Agha, *Concurrent Object–Oriented Programming*, CACM, September, 1990.

[7] Akinori Yonezawa, editor, *ABCL: An Object–Oriented Concurrent System – Theory, Language, Programming, Implementation and Application*, The MIT Press, 1990.

[8] T. Kofler. "Robust Iterators in ET++", *Structured Programming*, vol. 14, 1993, pp. 62–85.

[9] KE. Gorlen, "An Object–Oriented Class Library for C++ Programs", *USENIX Proceedings and Additional Papers C++ Workshop*, 1987.

[10] A. Goldberg and D. Robson, *Smalltalk–80: The Language and its Implementation*, Addison–Wesley, Reading, Massachusetts, 1989.

[11] J. Pallas and D. Ungar, "Multiprocessor Smalltalk: A Case Study of a Multiprocessor Based Programming Environment", *Proc. of SIGPLAN88*, 1988, pp. 268–277.

[12] D. Lea, "User's guide to GNU C++ Library". Technical report, Free Software Foundation, 1991.

[13] R. Lea and *et al.*, "COOL: system support for distributed programming", *Communications of the ACM*, vol. 36, no. 9, 1993, pp. 37–46.

[14] S. Schach. *Software Engineering*, Richard D. Irwin, Inc., and Aksen Associcates, Inc., 1993.

[15] N. Carriero and D. Gelernter, *How to write Parallel Programs: A First Course*, MIT Press, Cambridge, 1990.

[16] R. Butler and E. Lusk, "User's guide to the p4 programming system", Tech. Rep. Report ANL–92/17, Argonne National Laboratory, 1992.

[17] G. Schoinas, "Issues on the implementation of programming system for distributed applications", Tech. Rep., University of Crete, 1992.

[18] V. Sunderam, "PVM: a framework for parallel distributed computing", *Concurrency: Practice and Experience*, vol. 2, 1992, pp. 315–339.

[19] R. Harrison, "Portable tools and applications for parallel computers", *Int. J. Quantum Chem.*, vol. 40, 1991, pp. 847–863.

[20] C. Craig and *et al.*, "Parallel Programming Systems for Workstation Clusters"

[21] V. Sunderam and *et al.*, "The PVM Concurrent Computing System: Evolution, Experience, and Trends", *Parallel Computing*, vol. 20, no. 4, 1994, pp. 531–545.

[22] A. Beguelin and *et al.*, "A User's guide to PVM: Parallel Virtual Machine", Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, 1991.

[23] G. Geist and *et al.*, "User's guide and Reference Manual version 3.0", Technical Report ORNL/TM–12187, Oak Ridge National Laboratory, 1993.

[24] W. Bruce and *et al.*, "Design and Specification of Itera tors Using the Swapping Paradigm", OSU–CISRC–10/92–TR24, Ohio State University, 1992.

[25] M. Shaw and *et al.*, "Abstraction and verification in Alphard: defining and specifying iteration and generators", *CACM*, vol. 20, no. 8, 1977, pp. 553–564.

[26] B. Liskov and *et al.*, "Abstraction mechanisms in CLU", *CACM*, vol. 20, no. 8, 1977, pp. 564–576.

[27] M. Stephan and *et al.*, "pSather: Layered Extensions to an Object–Oriented Language for Efficient Parallel Computation", Technical Report TR–93–028, International Computer Science Institute, 1993.

[28] M. Stephan and *et al.*, "Sather Iters: Object–Oriented It eration Abstraction", Technical Report TR–93–045, International Computer Science Institute, 1993.

[29] P. America, "POOL: design and experience (Parllel Object–Oriented Language)", *OOPS Messenger*, vol. 2, no. 2, 1991, pp. 16–20.

[30] J. Yost, M. Marefat, and J. Kim, "Dynamic Simulation–Integrated, Intelligent Visualization: Methodology and Applications to Ecosystem Simulation", *Procedings of IFAC/IFIP Symposium on Analysis, Design, and Evaluation of Man–Machine Systems*, Massachusetts Institute of Technology, Cambridge, MA, June 1995.