THE DESIGN AND IMPLEMENTATION OF AN OBJECT-ORIENTED PARALLEL PROGRAMMING LANGUAGE

By

CHANG-HYUN JO

Bachelor of Economics
Sung Kyun Kwan University
Seoul, Korea
1984

Master of Science
Oklahoma State University
Stillwater, Oklahoma
1988

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF PHILOSOPHY
May, 1991

THE DESIGN AND IMPLEMENTATION OF AN OBJECT-ORIENTED PARALLEL PROGRAMMING LANGUAGE

Thesis Approved:

[illegible]
Thesis Advisor

J P Chandler

[illegible]

[illegible]

[illegible]
Dean of the Graduate College

PREFACE

The advent of commercial *parallel processing* machines in the hardware area and the emergence of new programming paradigms such as *object-oriented programming* in the software area have had a positive impact on the development of efficient and reliable software. The programming languages are necessary to satisfy sufficiently the requirements for parallel and distributed programming applications. It is also necessary that these languages support good software engineering methodologies. Object-oriented programming has emerged as a paradigm that supports practical software development with its ability to represent real-world problems. We are concerned in this dissertation with *high-level language support for distributed computing within the context of an object-oriented programming paradigm*, along with the dynamic interactions of objects. New language concepts and related language constructs are presented. These constructs are included in the definition of an object-oriented parallel programming language, and implementation of the language is outlined as well.

ACKNOWLEDGMENTS

First and foremost, my profound thanks go to my Ph.D. dissertation advisor, Dr. K. M. George. None of this could have been done without his consistent guidance, advice, kindness and humanity through the innumerable meetings and phone calls for many years.

I would like to thank the graduate advisor, Dr. John P. Chandler, for his sincere and consistent support during my graduate studies. I am also grateful to the other dissertation advisory committee members, Dr. George E. Hedrick and Dr. Keith A. Teague, for their encouragement, support, and critical review of my research. I would especially like to thank Dr. Keith A. Teague for arranging for me to use the iPSC/2 system in my research. I also wish to express my sincere appreciation to my M.S. thesis advisor Dr. Donald D. Fisher.

Thanks go to all of my friends, colleagues and office-mates, especially Brendan Machado who has provided me valuable help and assistance. My thanks also go to Michael Carter for helping me to use the iPSC/2 system easily, and to Anna Ventris for her kind help during my graduate studies. I would also like to convey my thanks to Dr. John P. Chandler and Michael Hearing who have proof-read multiple versions of this dissertation with care. In particular my thanks go to J. H. Baek and Dr. P. S. Jung who have enthusiastically participated in the informal AI study group seminars.

Finally and most importantly, my love and thanks go to my parents, Soon-Kyu and Boon-Sun Jo, who have provided consistent support, belief and love. With gratitude, I would like to thank my beautiful sons, Hyun-Soo and Jin-Soo, and my lovely wife, Ae-Kyung, for their love and patience throughout my graduate studies. I love you all forever.

TABLE OF CONTENTS

LIST OF FIGURES

Figures Page

CHAPTER I

INTRODUCTION

1.1 Prologue

Several commercial and experimental object-oriented programming languages have been designed and implemented. Some such as C++ [Stroustrup 82-89] and CLOS [CLOS 88] are extensions of existing conventional or functional programming languages, while others such as Smalltalk [Goldberg and Robson 83] and Eiffel [Meyer 88-89] are newly designed languages.

Parallel programming becomes a feasible idea because of the availability of new experimental and commercial parallel computers [August et al. 89] [Bakker et al. 87] [Baskett and Hennessy 86] [Bronnenberg et al. 86] [Cheng 89] [Dally 88] [DeBenedictis 88] [Dongarra 87] [Duncan 90] [Emrath 85,88] [Fox 87-89] [Gabriel 86] [Gehringer et al. 88] [Gottlieb et al. 83] [Haynes et al. 82] [Kuck et al. 86] [Kung 80,82] [LeBlanc et al. 88] [Odijk 87] [Padmanabhan 90] [Padua 79, Padua et al. 80] [Seitz et al. 88] [Test et al. 87] [Treleaven et al. 86] [Tsukakoshi et al. 87] [Veen 86] [Yew 88]. A number of papers have reported research about parallel programming and its language evolution [Ackerman 82] [Agerwala and Arvind 82] [Allen and Kennedy 82,85,87] [Andersen 89] [Ardo and Philipson 84] [Bal et al. 89] [Carriero and Gelernter 89] [Clapp and Mudge 89] [Cmelik et al. 89] [Fox et al. 88] [Gaudiot and Lee 87,89] [Gehani and Roome 86-90] [Goguen et al. 87] [Guarna 87,88] [Halstead 85] [Kim and Browne 88] [Kuck et al. 86] [Lamport 74-84] [Lee et al. 85] [Lee 88] [Lesser 74] [Li 86] [Luckham et al. 84] [Olsson 86] [Olson 85] [PCF 88] [Perrott 79,87] [Polychronopoulos 87,88] [Quinn et al. 88] [Ranka et al. 88] [Sabot 88] [Schutz 79] [Tsujino et al. 84] [Weihl 88,89] [Workshop 88]. Other research reports the work on concurrent/parallel programming by extending the existing programming languages and systems, such as logic programming languages [Clark and Gregory 86] [Clark 87] [Codish and Shapiro 87] [Conery and Kibler 85] [Corsini et al. 89] [Lin and Kumar 88] [Shapiro and Takeuchi 83, Shapiro 87,89] [Talia 90] [Tebra 87], functional programming languages [Goldberg and Hudak 88] [Hudak and Smith 86], symbolic programming languages [Halstead 85] [Fidge 88], and procedural languages [Guzzi 87] [Karp and Babb 88]. Besides the above, there are a number of concurrent and parallel programming language models [Agha 86-89] [Brinch-Hansen 73-87] [Chandy and Misra 88] [Dijkstra 68,75] [Hennessy 88,90] [Hoare 73-85] [Liskov 81,88, Liskov et al. 86,87, Liskov and Shrira 88] [Milner 80] [Olderog and Hoare 86].

Since the properties of object-oriented programming are well suited to the nature of real-world problems, and since parallel computers provide the potential of speed-up, combining these two areas is emerging as an important issue [Agha 89] [SIGPLAN 89] [Yonezawa and Tokoro 87] [Wegner 87-90]. Some researchers have already incorporated concurrency/parallelism into object-oriented programming languages [America et al. 86] [Bennett 87,90] [Corradi and Leonardi 87-90] [Gehani and Roome 88b] [Hur and Chon 87] [Rose and Steele 87] [Rosing et al. 88] [SIGPLAN 89] [Shibayama 89] [Tripathi and Berge 89] [Watanabe and Yonezawa 88] [Yonezawa and Tokoro 87]. Others have built object-oriented parallel programming languages and systems [Bershad et al. 88] [Black et al. 86] [Chase et al. 89] [Kafura 88, Kafura and Lee 89] [Koszarek 88] [Jul et al. 88] [Björnerstedt and Britts 88] [Yang et al. 89] [Zimmerman and Crichton 89]. The languages such as Concurrent C++ [Gehani and Roome 88b] and Emerald [Jul et al. 88], have sacrificed some properties of object-oriented programming (e.g., class in Emerald and inheritance in Concurrent C++) while achieving concurrency.

However our basic tenet is that all properties of object-oriented programming must be preserved when incorporating concurrency into the programming language. Therefore, we take an approach different from the previous approaches. We are interested in incorporating distribution and concurrency within object-oriented programming without sacrificing any of the object-oriented features. We are also interested in language concepts capable of capturing dynamic interactions of objects. With this objective, we have developed a programming language, Parallel-C++, to support parallelism within the object-oriented programming paradigm [Jo and George 89,91]. Parallel-C++ is based on C++ and preserves the properties of object-oriented programming while providing parallel programming. It also incorporates language concepts to support object migration. This research work is influenced by the work of several authors whose research extends from theoretical foundations to specific programming languages [Agha 86-89] [Brinch-Hansen 73-87] [Chandy and Misra 88] [Dijkstra 68] [Goldberg and Robson 83] [Grogono and Bennett 89] [Hoare 73-85] [Liskov 81-88] [Meyer 88-89] [Milner 80] [Nguyen 85] [Nygaard and Dahl 78, Nygaard 86] [Stroustrup 83-89] [Wirth 74-90]. One of the perceived benefits of the Parallel-C++ programming language is to provide a *model of dynamic behavior* in object-oriented systems. We introduce the new notions of *dynamic* and *static* objects, and a dynamic relationship between them called *ownership*. We show how these concepts can be incorporated into the programming language. We also illustrate the use of these concepts in practical situations.

This dissertation also includes the implementation scheme of a translator for the language Parallel-C++ [Jo et al. 91] on the Intel iPSC/2 hypercube multiprocessor computer [Intel 88]. The Parallel-C++ translator translates a Parallel-C++ source code into C++ programs supplemented with system primitives and subroutines on the iPSC/2. The translation scheme takes advantage of an existing C++ compiler [AT&T 85-89] for the iPSC/2.

The implementation scheme of a compiler and an interpreter for Parallel-C++ is also presented. The objective of compilation rather than translation is to generate more efficient object codes for the target machine which is here iPSC/2. The run-time storage management for distributed and dynamic objects is also described with some examples.

Throughout this dissertation, a *processor* denotes a physical computation unit, and a *process* denotes an abstract computation unit. An *object* can be mapped into a process in an abstract computation machine and can be mapped into a processor in a physical computation machine. Processes are called *concurrent* if their executions have the potential to overlap in time. *Parallel* computation means that parts of program segments (concurrent processes) actually execute at the same time by using a number of different processing elements.

The remainder of the dissertation is organized as follows: The rest of the sections in this chapter discuss parallel programming within conventional language and object-oriented programming in general, and surveys related work. The design issues for Parallel-C++ are described in Chapter 2. Chapter 3 defines the language syntax and semantics with several application examples to show the merits of the new concepts and to show how the concepts can be applied in practice. There are two implementation schemes, translation and compilation. Chapter 4 shows the implementation scheme for the translator. Chapter 5 gives a compiler-interpreter implementation scheme for Parallel-C++. Chapter 6 discusses contributions of this research and suggests possible topics for future research work with the conclusion.

1.2 Parallel Programming and Object-Oriented Programming

Parallel languages provide features that support the design and implementation of parallel algorithms. Programming languages provide two approaches to parallelism in the language constructs. The first approach is implicit parallelism. A tangible configuration of this is program restructuring, especially in the optimization step of compilation. A serial or parallel program can be restructured into a parallel code targeted for a specific parallel architecture. In this approach the user is not concerned with the parallelism; the optimizer parallelizes the program by restructuring [Allen and Kennedy 82-87] [Kuck et al. 86] [Midkiff and Padua 87]. In the second approach the user specifies the parallelism explicitly.

Explicit parallelism is user-programmed parallelism using the parallel constructs provided by the language. This approach can be accomplished by extending the existing languages [PCF 88] or by providing new parallel programming languages [Bal et al. 89]. Despite the advances, the use of parallel machines is still limited by the languages that are available.

Object-oriented programming is another evolution in programming languages. Many researchers believe that object-oriented programming provides a natural model to represent the real world [Stefik and Bobrow 86]. Several object-oriented programming languages have been designed and are being used for practical software development [Deutsch 89] [Goldberg and Robson 83] [Grogono 89] [Grogono and Bennett 89] [Kamin 88,90] [Koschmann and Evens 88] [Lieberherr and Holland 89] [Meyer 88-89] [Peterson 87] [Raj and Levy 89] [Stroustrup 83-89] [Wasserman 90] [Wegner 86-90] [Wolf 89]. While some languages add new concepts to object-oriented programming languages, others adapt object-oriented methods to new areas of application such as distributed computing [Yokoto and Tokoro 87] [Yonezawa and Tokoro 87].

1.2.1 Parallel Programming and Languages

Programming language design for distributed computing usually follows the following three approaches. The first approach is to parallelize existing sequential languages. The parallelizing compiler or translator detects program segments, to be possibly parallelized, and restructures sequential program to parallel programs. The work at Rice [Allen and Kennedy 82-87] and Illinois [Kuck et al. 86] [Padua and Wolfe 86] [Polychronopoulos and Banerjee 87, Polychronopoulos 87-89] are examples of this approach. The second approach is extending an existing language by providing language constructs to make parallel programing easy and efficient. This approach extends the syntax of conventional languages and uses dedicated compilers or translators. This type of approach is found in Distributed Smalltalk [Bennett 87,90], PCF Fortran [PCF 88] and others [Karp and Babb 88]. The third approach is to design new parallel languages. The work of Emerald [Black et al. 86], Concurrent Smalltalk [Yokoto and Tokoro 87] and others [Yonezawa and Tokoro 87] are examples of research using this approach.

1.2.2 Object-Oriented Programming Systems (OOPS)

Object-oriented programming (OOP) is based on many important language concepts such as abstract data types (ADT), encapsulation, dynamic binding, class, object, inheritance, polymorphism, and reusability [Olthoff 86] [Snyder 86]. It is also based on mathematical models such as universal algebra [Birkhoff 82] [Grätzer 79]. In OOP, the important concept is an object. An object encapsulates data structures and operations on those. This concept looks similar to data abstraction of abstract data types as occurs in Ada [Ada 79,83], CLU [Liskov and Zilles 75, Liskov et al. 77, Liskov and Snyder 79, Liskov and Guttag 86, Liskov 87], and Modula [Wirth 77]. Objects communicate with each other by passing messages using interface functions of the objects. A class is a template for its objects. The notion of class is similar to that of type. Objects which have the same type are generated from a class. Each object is distinct by having its own state. The concept of class was first introduced in Simula 67 [Franta 78] [Nygaard and Dahl 78]. This concept has been adopted in Smalltalk [Goldberg and Robson 83], Eiffel [Meyer 88-89], C++ [Stroustrup 83-89] and other object-oriented programming languages. Inheritance and dynamic binding facilities make object-oriented programming different from imperative programming. Dynamic binding makes typed systems truly generic. For example, in C++, virtual functions allow dynamic bindings. Inheritance allows objects to use data and functions of the objects defined by parent class. Reusability is another merit of object-oriented programming. Using inheritance and dynamic binding, the programmer reuses pre-written codes for new programs. For a large software system, it may reduce a substantial amount of code repetition, and hence it may reduce the cost of software development.

Another merit of object-oriented programming is that OOP represents much of the real world. In the real world, many entities may be regarded as an object. For example, those objects are humans, animals, cars, factories, more abstract systems and even atoms. They have their own data structures and interface functions. Objects communicate with each other by passing messages through interface functions. For example, atoms interact with each other, and they are combined to produce compound structures. A system keeps its own states even though structures may be similar, and may communicate with other systems by calling their interface functions. The distinct internal states of systems make them distinguishable from each other. The internal states of a system cannot be accessed by others directly, but can be altered by requests from others through the formal interfaces. Many object-oriented languages have been developed to support OOP paradigms. Some of those are Actor [Agha 86-89], Objective-C [Cox 83,86] [Cox and Schmucker 87], Smalltalk [Goldberg and Robson 83,89], Eiffel [Meyer 88-89], Trellis/Owl [O'Brien et al. 87], C++ [Stroustrup 82-89] and Self [Ungar and Smith 87].

In object-oriented programming, we are concerned with what has to be done in a module, while also being concerned with how it has to be done as in procedural programming. While there are some arguments that object-oriented programming is only an extendible typed language [Wirth 90], and real-world relations cannot be well represented only by inheritance, it is too early to appreciate the future impact of this on the real programming world. Nowadays OOP pervades the programming area and several application areas (such as database systems), but it still needs more development of theoretical background. However, the applications are growing much faster than the development of those theories. Object-oriented languages like Smalltalk are not only languages, but are also systems including their programming environments. Object-oriented programming is radically developed and widely used for graphical user interfaces [Myers et al. 90], data base systems [Date 90] [Kim et al. 87] [Maier et al. 86] [Smith and Zdonik 87] [Ullman 88], and operating systems design [Shapiro et al. 89].

1.2.2.1 Class and Object

An *object* is a unit system in the OOPS, usually being identified by a unique identifier, also having a type. An object consists of its own data members and the interface operations, called "methods", that operate on those data. These operations provide the interface to other objects to access and change the object's data. The data members may be altered only by the object's operations (for private members) or can be accessed only by other objects (for public members). Other objects request certain operations through these interface operations. In OOPS, this is called "message passing".

A *class* is a template for its objects. Several objects may be created from a class. Initially they all look the same. Different objects are differentiated by their own identifiers, and they have their own storage for their data members and may have different values. Even if the values for member data are the same, those objects are still different because they have their own address spaces.

1.2.2.2 Inheritance

Abstract data types (ADT) and objects (classes) originate from the same roots from the point of view of information hiding and encapsulation. The main difference between these two ideas is in *inheritance* [Snyder 86] [Stroustrup 89b]. OOPS uses class hierarchy to represent the static relationships among the objects. Those relationships can be any one of the conventional relationships with real-world entities, such as "is-a" or "has-a". This hierarchy can be represented by a tree (for single inheritance) or by a lattice (for multiple inheritance). Parent classes are called super-class and children are called sub-class. The object created from the sub-class may inherit the properties from the super-class. If a sub-class can inherit properties from more than one super-class, we call this multiple inheritance. These concepts are emphasized as important concepts to explain real-world problems in several research reports [Stroustrup 89b].

1.2.3 Programming Language Design and Implementation

The development of computer hardware leads us to the development of computer software which makes use of the hardware. A programming language is one of the important tools for developing computer software. Programming languages are not "the solution" to software problems, but because of their central role in software, they can help to simplify solutions [Fisher 76]. Adoption of an appropriate programming language may help to remove the barriers to solving software problems.

Some of the desirable characteristics of a programming language are the following [Fisher 76] [DoD 77]:

(1) The language should have a complete and unambiguous definition of syntax and semantics.

(2) The language should not be dependent on any particular object machine or any particular operating system.

(3) The language should be able to test and verify its correctness.

(4) A parallel language should provide the ability to create and terminate parallel processes through the parallel language constructs and/or the parallel system primitives.

(5) Synchronization ability is needed [Dinning 89]. Mutual exclusion to the system resources should be obeyed by concurrent processes.

(6) Exception handling should be provided when arithmetic overflow, exhaustion of free space, hardware errors, or any other run-time error occurs.

1.3 The Language Parallel-C++

In this dissertation, we present a new language, Parallel-C++, which allows parallel and distributed programming in an object-oriented programming environment [Jo and George 89,91]. In *distributed object-oriented parallel programming,* some of the objects in the system can be distributed on the different address spaces, and those objects can be executed in parallel. Parallel-C++ incorporates new language concepts such as static/dynamic objects and ownership.

In developing the concepts of dynamic and static objects, and ownership, we adopt ideas from object-oriented programming languages, especially C++ [Stroustrup 82-89]. The concepts of class, object, message and their syntax are the same as those in C++. C++ is an inherently sequential language. We extend the C++ language to support object-oriented parallel programming. The reason for choosing C++ as the base language for parallel extension is that C++ offers familiar syntax, static type checking, and its base language C has been widely used (which means many applications such as image processing are programmed by using it [Brown M89]). But many ideas for language design and compiler implementation are adopted from other object-oriented languages such as Smalltalk [Goldberg and Robson 83] and Eiffel [Meyer 88].

The *purpose* of our work for parallel programming within an object-oriented paradigm is to:

(1) Suggest a problem domain for object-oriented programming in the distributed computing environment.

(2) Survey current programming languages and review their distinguishing features for distributed programming and object-oriented programming.

(3) Design an object-oriented parallel programming language, Parallel-C++, to support distributed programming within an object-oriented paradigm.

(4) Explore different implementation schemes.

The approach of Parallel-C++ supports an abstraction of the whole system as opposed to abstraction of process migration, as in the case of languages and systems in the literature. The newly introduced language constructs "import" and "export" not only support object mobility, the associated concepts of dynamic object and ownership, but they also support even more general scenarios such as computation of an expression in different environments. This is because the object responding to a message of a dynamic object is not the same all the time. For example (assuming an aircraft is hijacked), the reply to a message from a pilot requesting permission to land at an airport depends on the airport. From that perspective, management of object migration becomes a consequence and not an end in itself. Moreover, the "*copy-and-delete*" semantics of "export" is different from the semantics followed by the other systems and introduce a different scope of objects.

1.4 Related Work

Besides the work mentioned in the previous sections, other related work is in the areas of load balancing and process migration. Process migration is abstracted as object mobility in the object-oriented systems. Several languages are concerned with providing support for load balancing and object mobility. The notions of process migration and object mobility, as well as their discussion, are not new. Since the 1970's, process migration and object migration have been thoroughly discussed in the literature for the several distributed systems [Artsy and Finkel 89] [Bennett 87,90] [Björnerstedt and Britts 88] [Douglis and Ousterhout 87] [Jazayeri 89] [Powell and Miller 83] [Zayas 87]. However, language support for object mobility based on object-oriented programming is a relatively new approach. The systems that incorporated this approach are Emerald [Black et al. 86] [Jul et al. 88], Sloop [Lucco 87], Amber [Chase et al. 89] and SOS [Shapiro et al. 89]. These and other similar systems, as well as languages, focus on object management and provide support for explicit management of object mobility. Other related work includes Distributed Smalltalk [Bennett 87,90], Argus [Liskov 88], Presto [Bershad et al. 88], Orca [Bal and Tanenbaum 88], and Concurrent C++ [Gehani and Roome 88b]. Especially, the works of [Beck 90] [Shapiro et al. 89b] [Yin et al. 90] focus on the research of parallel languages and systems based on C++. A comparison of the relevant characteristics of the related programming languages are summarized in Figure 1.1.

Object Model	Base	Characteristics	Object Support	Memory	Mechanism
Parallel-C++	Object-Oriented	Object Mobility Parallelism	Language	Distributed	Copy-and-Delete
Emerald	Object-Based	Object Mobility	Primitive	Shared	Call-by-Object-Reference
Distributed Smalltalk	Object-Oriented	Object Mobility	Primitive	Shared	Object Sharing
Amber	Object-Based	Object Mobility	Operating System	Shared	Remote Object Invocation
Sloop	Object-Oriented	Object Mobility	Language	Shared	Virtual Object Space
SOS	Object-Oriented	Object Mobility	Operating System	Both	Move or Copy
Argus	Object-Based	Location Independent	Language	Distributed	Remote Call-by-Value
ConcurrentC++	Class-Based	Parallel Processes	Language	Both	Transactions
Orca	Object-Based	Shared Data Object	Language	Both	Call-by-Sharing
Presto	Object-Oriented	Threads	System	Shared	Pre-defined Object Types

* Object-Oriented	= Object + Class + Inheritance	[Wegner 87]
Class-Based	= Object + Class	
Object-Based	= Object	

Figure 1.1. Comparison of Languages and Systems supporting Concurrency based on Object Model

In the following sections, a brief outline of related work is provided.

1.4.1 Emerald

Emerald is an object-based language and system designed for the construction of distributed programs [Black et al. 86] [Jul et al. 88]. Emerald provides explicit language support for *fine-grained object location and mobility*. Emerald objects consist of data objects and process objects. In Emerald's object model, local objects are private and remote objects are to be shared. Emerald supports concurrency between objects in a network, and supports concurrency of parallel operations within an object. A monitor-like construct is used for synchronization of concurrent operations. Emerald does not provide either classes or inheritance. Emerald uses three different implementation styles for objects, namely global objects, local objects, and direct objects. It provides five primitives for object location and migration: locate, move, fix, unfix, and refix. It also suggests various kinds of parameter passing modes related to objects, such as call-by-object-reference, call-by-visit, and call-by-move.

The authors of Emerald report that a prototype implementation targeted to a small network (DEC Micro-VAX II workstations) has been done. The Emerald implementation is targeted to shared memory architectures [Jul et al. 88]. These researchers report that Emerald is intended to run in a modest size network (e.g., within 100 nodes). The implementation consists of an Emerald compiler and an Emerald kernel. The creation of objects is accomplished by explicit constructors. The kernel supports code sharing. Remote objects are managed by an object table in each node. Invocation overhead has been reported, because the activation records for executing processes must also move when an object is moved.

1.4.2 Distributed Smalltalk

Distributed Smalltalk (DS) is an implementation of Smalltalk in a distributed system [Bennett 87,90]. DS and Emerald credit their origins to a common root. DS provides for object mobility, communication between remote users, direct access to remote objects, object sharing among users, and *distributed applications in the Smalltalk environment*. The presence of the class is required to move and use an object. DS also provides several design alternatives which could be useful to make implementation decisions of a distributed object system.

The authors report that Distributed Smalltalk has been implemented on a network of Sun-2 workstations. The characteristics of the implementation include distributed garbage collection, access control, and object mobility.

The work related to Distributed Smalltalk is Multiprocessor Smalltalk [Pallas 89]. Multiprocessor Smalltalk discusses *a Smalltalk implementation on a multiprocessor system*, and provides related performance analysis.

1.4.3 Amber

Amber is a programming system that allows a single application program to run on a homogeneous network, where each node is a shared-memory multiprocessor [Chase et al. 89]. Amber uses an existing programming language and operating system to provide support for concurrency and distribution of programs. The distribution model and *mobility primitives* are adopted from Emerald, and Amber's thread and synchronization models follow those of Presto. The Amber system consists of a C++ preprocessor and a run-time kernel. Amber programs are written in a subset of C++, enhanced with primitives for *thread* and *object* management. Objects and threads, which provide explicit support for concurrency, can be created dynamically. A collection of mobile objects can be distributed among nodes in a network, while interacting with each other through location-independent invocation. Computational load distribution is determined by the locations of data objects.

The implementation reported by the authors is on a DEC Fire-fly running the Topaz operating system. Amber programs are converted into a set of Topaz tasks, and they are distributed in the network, in which one task executes on each participating node. The global virtual memory is implemented by arranging the virtual address space which simplifies object migration.

1.4.4 Sloop

Sloop can be viewed as a parallel programming language and environment [Lucco 87]. It is based on an object-oriented model that supports explicit MIMD parallelism. Sloop allows explicit assignment of objects to physical processors by providing three operations, access, align, and copy. Sloop supports indivisible objects in which only one operation invocation executes at a time. Cooperating distributed objects can be located on different physical processors in a multiprocessor, and all distributed objects can simultaneously execute operations, while interacting with each other using an asynchronous access mechanism. Sloop hides details of the underlying multiprocessor by providing a *virtual object space* which contains a collection of objects that cooperate to solve a problem.

The authors report that Sloop implementations at AT&T Bell laboratories are running on a bus-based multiprocessor (the S/Net), on an experimental 64 processor hypercube processor, and on a heterogeneous network of workstations connected by an Ethernet. A Sloop program is transformed into a C program by the Sloop compiler. The Sloop run-time system, written in C++, uses object relocation heuristics and coroutine scheduling to perform mapping and load balancing.

1.4.5 SOS

SOS is a *Distributed Object-Oriented Operating System* [Shapiro et al. 89] [Shapiro et al. 89b]. While SOS uses standard operating system techniques, it also supports medium-sized objects, distributed or fragmented objects, and object migration. In SOS, to establish a remote service, a client object must acquire a proxy, which is a local interface object representing the service and migrating into the client's context at the time of need. Such a proxy can process the service locally or remotely.

SOS, written in C++, is prototyped on top of Unix (SunOS) [Shapiro et al. 89b]. The standard C++ tools such as inheritance and coercion methods, as well as a task library for coroutine-style programming, are used to provide interface between applications and the system.

SOS incorporates several similar concepts to Parallel-C++ relating to object mobility. However, there are still substantial differences between SOS and our work. First of all, the approach of SOS is operating system based, whereas our approach is high-level language based. Secondly, SOS is concerned with facilitating the implementation of compilers, whereas we are concerned with providing high-level language support for abstraction and developing a compiler for it. Thirdly, many ideas, such as Dynamic

Interface Environment Binding (DIEB), ownership, concurrent language support, new control flow and enhanced scope rules, are uniquely suggested in our work.

1.4.6 Argus

Argus is an object-based language and system which supports fault-tolerant distributed programming by providing location-independent invocation of distributed objects [Liskov and Scheifler 82] [Liskov 88]. There are two different entities, Argus *guardian* (which is an abstract object encapsulating resources) and CLU cluster (which represents local objects contained inside guardians) in Argus. A guardian resides in a node even though its resident location is changeable. A distributed program is composed of a number of guardians which can execute in parallel. Data objects storing resources in a guardian can be accessed by calling handler procedures. Argus uses a "pass-by-value" (not by-reference) parameter passing mode. Argus also provides *atomic* actions (like transactions in database systems) within programming languages like Avalon [Ditlefs et al. 88]. It does not provide the idea of object migration, but related issues are well discussed.

A prototype of Argus is implemented on a collection of Micro-VAX II workstations operating under Ultrix.

1.4.7 Concurrent C++

Concurrent C extends C by adding concurrent programming facilities [Cmelik et al. 89] [Gehani and Roome 86, 88, 90]. Concurrent C++ is obtained from Concurrent C by adding data abstraction facilities of C++ [Gehani and Roome 88b]. The communication model of Concurrent C is based on Ada rendezvous. A Concurrent C program consists of a set of *processes*, which are sequential programs independently executing in parallel. Concurrent C provides facilities for operations on processes such as processes creation, process termination, process synchronization, and priority specification. Concurrent C processes communicate with each other by means of (synchronous and asynchronous) transactions which can be thought of as services called by other processes. Concurrent C++ facilitates data abstraction of C++ within concurrent programming in Concurrent C. The language extension is made by allowing process variables in Concurrent C as class members in C++, and allowing process operations in Concurrent C as member functions in C++. Concurrent C++ processes allow multiple threads of control. However, Concurrent C++ does not allow inheritance by a derived process. Gehani and Roome report that data abstraction and parallel programming facilities seem to be orthogonal; however, the merger of Concurrent C and C++ raises several integration issues which might be valuable to review.

Concurrent C has been implemented on several types of systems: a Unix based single processor system, a set of VAX computers connected by an Ethernet network, and a shared memory multiprocessor system. The shared memory multiprocessor system is assumed to be an appropriate architecture for programming in Concurrent C.

1.4.8 Orca

Orca adopts a *shared data-object model* to facilitate distributed programming [Bal and Tanenbaum 88]. Shared data are encapsulated within passive objects which are instances of abstract data types. Parallel activities are realized by dynamic creation of multiple sequential processes. A process can pass its objects as shared parameters to its children. Processes communicate indirectly through shared data-objects, which play the role of a communication channel between processes because any changes to the object are visible to all processes. Such a scheme is similar to call-by-sharing in CLU [Liskov et al. 77]. Access to shared data is allowed through indivisible operations only, and such access is automatically synchronized.

Bal and Tanenbaum provide the following implementation models: implementation in a distributed system, with point-to-point message, and with reliable/unreliable multicast message. A prototype implementation has been done on top of the Amoeba distributed operating system. Orca programs are transformed into intermediate codes which can be translated into machine-dependent object code on a target system. The implementation is based on automatic object replication and object migration performed by a run-time system.

1.4.9 Presto

Support of object-oriented parallel programming in a multiprocessor environment is the objective of the Presto system [Bershad et al. 88]. C++ with a library and a run-time system are the primary components of the Presto system. Presto provides users with a set of pre-defined object types useful for writing parallel programs. In Presto, all objects execute in a single address space shared by all processors. Thread objects provide fine-grained control over an execution of a program, and synchronization objects provide concurrency control for threads executing simultaneously. Once created, a thread is capable of executing operations of an object in parallel with the starting thread.

Presto has been implemented as a *run-time library* written in C++ on Sequent Balance and Symmetry shared memory multiprocessor machines on top of the Dynix operating system. The Presto run-time system maps a user's threads onto physical processors, and provides access to a global shared memory. All mappings are transparent to the Presto users. A single scheduler object keeps track of all threads that are ready to run in each processor object. A user's programs are linked with the Presto library to obtain executable programs.

1.4.10 Other Related Work

Some operating systems for distributed systems, such as the Sprite operating system [Douglis and Ousterhout 87] and the DEMOS/MP operating system [Powell and Miller 83], use process migrations as a method by which executing processes transfer between processors. Other related work includes Loops [Kempf et al. 87], Linda [Gelernter 85] [Leler 90], Flavors [Moon 86], Concurrent Smalltalk [Yokoto and Tokoro

87], and CLOS [CLOS 88] [Keene 89] in the area of object-oriented programming languages and systems.

The next chapter is devoted to clarifying the problem domain in which we have to design new language constructs, and to explaining the concepts that are incorporated into Parallel-C++.

CHAPTER II

OBJECT-ORIENTED PARALLEL PROGRAMMING LANGUAGE

2.1 Problem Domain

Processes are a set of programs. Processes are called *sequential* when they execute sequentially one instruction at a time by following a given thread of control. Using sequential processes, we have been doing serial programming in various ways. A sequential process delivers the same result with given data [Brinch-Hansen 73]. Many specification methods and proof methods of correctness for sequential programming have been developed. They allow us to use sequential programming as a tool to express our behaviors.

But nowadays, we have met another genre in our computing. It is known as concurrent or parallel programming. Processes are called *concurrent* if their execution may be interleaved or overlapped in arbitrary order in a given time. Many of our actions in real life are done by concurrent behaviors. For example, we may have dinner while reading a newspaper, watching TV and talking to members of the family. While we drive a car, we are watching the road, listening to classical music, pushing the pedal and changing gears. In a factory, many control tasks can be done efficiently in parallel. In various ways, concurrent action is a natural form of human behavior. The programming that describes our lives must be able to express these kinds of parallelism. Parallel programming should be able to meet these needs in our lives. Concurrent programming can be achieved in various ways such as interleaving, overlapping, pipeling, time-sharing and multi-tasking execution of instructions. While *concurrency* gives the potential for *parallelism*, parallel processing achieves the actual simultaneous executions of parts of operations. Concurrent and parallel programming has been discussed as early as the sixties in the literature [Brinch-Hansen 73-77] [Dijkstra 68] [Hoare 74-78].

During the past decades, we have seen real, experimental and commercial concurrent processing machines which use multiple parallel processors. Those machines have demonstrated speed-up, efficiency and effectiveness in various research projects [Fox 87-89]. As a consequence, we need parallel languages to implement parallel algorithms efficiently. Some parallel languages have been investigated for achieving such a need in parallel computing.

2.2 Concurrent Language Features

A well-known representation of concurrent processes is a "parallel statements" construct [Dijkstra 68]. The "parallel statements" construct has several concurrent statements in the construct [Figure 2.1]. The "parallel statements" construct is well suited to structured programming, since a parallel statement has a single fork-point "parbegin" and a single join-point "parend".

begin S_0; parbegin S_1; S_2; ...; S_n parend; S_{n+1} end;

Figure 2.1. Parallel Statements

The parallel statements can be depicted by a precedence graph like the one shown in Figure 2.2. This language construct facilitates fine-grained parallelism among concurrent statements.

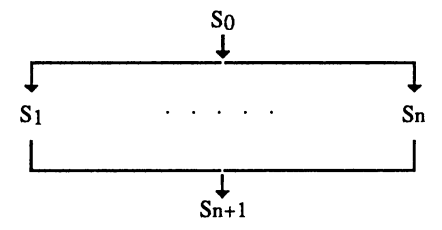
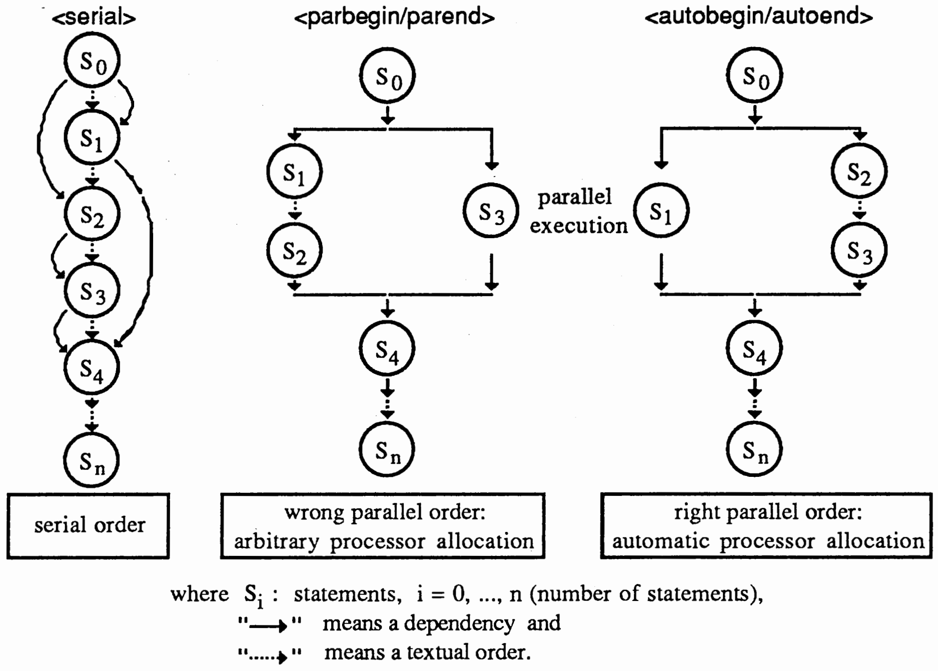


Figure 2.2. Precedence Graph of the Parallel Statements

When using the "parbegin/parend" construct, programmers should guarantee independence of statements in the construct if the compiler does not check for dependencies. To diminish this burden of programmers, another parallel block construct for automatic processor allocation, "autobegin/autoend", can be provided. The "autobegin/autoend" construct shifts the burden of enforcing parallelism from the programmer to the compiler. The compiler uses the restructuring algorithms to check for dependencies. Having determined the dependencies, the code is parallelized. If all statements are mutually independent, this is the same as a "parbegin/parend" statement. While processing this construct, an automatic processor allocator translates the program segment specified in the construct to a virtual execution graph, which is used to check interdependency among the statements. An automatic processor allocator congregates the dependent statements into a block at each dependency checking step. Finally, the automatic processor allocator maps the disjoint program segments to the dependent processors. This procedure is illustrated in Figure 2.3. Suppose we have a sequence of statements, S_0, ..., and S_n, to be executed. In Figure 2.3, the leftmost figure represents a sequential execution of the statements. In the sequential execution, the symbol " $\rightarrow$" represents a dependency of the statements, and the symbol "···>" represents a textual order of the statements. For example, "$S_0 \rightarrow S_2$" means that the execution of the statement S_2 depends on the execution of the statement S_0 (which means that S_2 cannot be executed before/while S_0

executes), and "$S_0 \cdots > S_1$" simply means that the statement S_0 is placed before the statement S_1 in a program.



<serial>
<parbegin/parend>
<autobegin/autoend>
S_0
S_1
S_2
S_3
S_4
S_n
parallel execution
serial order
wrong parallel order: arbitrary processor allocation
right parallel order: automatic processor allocation
where S_i : statements, i = 0, ..., n (number of statements),
"⟶" means a dependency and
"·····▸" means a textual order.

Figure 2.3. Automatic Processor Allocation Graph

Based on the information given in the leftmost figure which shows serial execution of the program segment, we may have two versions of parallel execution, using different parallel constructs such as "parbegin/parend" (middle figure) and "autobegin/autoend" (rightmost figure) which have different semantics. In the "parbegin/parend" figure, after the execution of the statement S_0, the statements, S_1, S_2, and S_3, are arbitrarily parallelized without checking interdependencies among the statements. As a consequence, this execution of the parallelized statements leads to a wrong result. Since the execution of the statement S_3 depends on the execution of the statement S_2, the statements S_2 and S_3 cannot be arbitrarily parallelized, and they must execute sequentially. In the "autobegin/autoend" figure, the statements S_2 and S_3 are serialized into a block, and they can execute with another block of the statement S_1 in parallel. This kind of automatic restructuring of a concurrent program is done by an automatic processor allocator in the parallelizing translator/compiler.

From the point of view of granularity of parallelism, we have only discussed the language features necessary to support fine-grained parallelism. We now introduce a language construct, "explicit process allocation", which may be useful for medium-grained parallelism.

We may explicitly assign processes to a number of parallel statements. In this case, parallel statements may be represented by several blocks of concurrent processes, and each process block consists of sequential statements in it. This language construct, the "explicit process allocation" (shown in detail in the next chapter), is initiated from the "parallel compound statement" [Dijkstra 68] and Ada "tasks" [Ada 79,83].

The statements assigned by an "explicit process allocation" construct are in a critical region. The statements among processes are mutually exclusive and the processes defined by the construct are completely independent of each other.

In general, we apply the *rule of disjointness* [Brinch-Hansen 73] to the parallel construct. The disjointness implies that a variable "v_i" changed by a statement "S_i" cannot be referenced by other statements. A compiler may provide a facility to check the interdependencies between concurrent process statements at compilation time. But this method does not guarantee to find dependent variables in the concurrent statements, with pointer variables, reference variable and the address of the variables being the typical examples of this case. We cannot find out their values at compilation time, because values are not assigned statically. Because they can be changed dynamically, those values can be known at run-time only. To prevent erroneous or unpredictable programs using such variables and common variables accessible by several concurrent processes, we need another language construct to exchange information of such variables between concurrent processes. Several language facilities for synchronization such as semaphores, critical regions and monitors, have been suggested in the literature. With a distributed memory facility without any local memory, message passing is the well-known method to communicate among concurrent processes. A prospective parallel language may include a synchronization construct "send/receive" to communicate and exchange data between concurrent processes. The "explicit process allocation" construct supports programming in medium-grained parallelism. Also, the parallel language may include language constructs like "parallel function calls" for medium and large-grained parallelism. We define the above language features in detail in the next chapter.

So far, we have discussed parallel programming in modular and structured programming only. In the next section, we discuss parallel programming with an object-oriented programming paradigm.

2.3 Distributed Object-Oriented Programming

2.3.1 Object-Oriented Programming in Distributed Computing

One of the barriers to parallelism in the conventional serial languages is access to common variable and global constructs. Local modularization, like objects in object-oriented programming, can be one of the solutions to achieving medium-grained and large-grained parallelism.

In distributed memory architecture, the cost of frequent communications among the distributed small computations is very high. To expect better efficiency and to reduce the frequency of communication, the other kind of computations may be considered. In this case, the modularization of the computation with local variables is necessary. This kind of module can be represented by an object in the object-oriented programming paradigm. An object includes local data and interface operations. Local data in an object can be modified only by its operations in the object. Objects communicate with each other by using these interface operations.

This section describes the idea which has been developed to allow object-oriented programming in a distributed computing environment. The major ideas and concepts in the design of the programming language Parallel-C++, which has incorporated object-oriented programming within parallel and distributed computing, are presented here.

2.3.2 Static and Dynamic Objects

The key concept in C++ is the *class* [Stroustrup 86]. Parallel-C++ adopts this concept to support data abstraction and information hiding. The class is a template from which several *objects* can be defined. Objects are instances of classes. An object consists of local states and methods which may modify its local states. There are two kinds of objects, *static objects* and *dynamic objects*, in our distributed object-oriented programming system:

(1) A *static object* (*free object*) is a stationary object. A static object may own any number of dynamic objects. A static object can be also viewed as a master object which controls other objects in a system.

(2) *Dynamic objects* can be moved from one static object to another. One or more dynamic objects can be owned by a static object. A dynamic object owned by a static object can be moved to another static object.

Parallel-C++ incorporates another important concept which is "*Distributed Object-Oriented Parallel Programming*". In distributed object-oriented parallel programming, objects can be distributed to have their own processors and hence they can be executed in parallel on their respective processors. Conceptually, a static object is bound to a processor and this binding does not change during the lifetime of the object. However, in the case of a dynamic object this binding varies. Several dynamic objects can be owned by a static object. Once an object is owned by any static object (or process), other objects (or processes) cannot access the methods and data structures of such an object. An object as one of the instances of the class can be accessed by only one object (or process) which owns that object at any time. A dynamic object can communicate with its owner by using the owner's methods.

Dynamic objects can be *export*ed and *import*ed by other objects. Permission to access objects owned by other processes can be gained by using "*export/import*" constructs only. The basic construct is shown in the language definition in the next chapter. Unlike in Modula [Wirth 77] and in Eiffel [Meyer 88], the terms "export" and "import" are used in relation to dynamic objects only. An object does not export itself, and the visibility of imported object is limited to the importing object. One process can export its objects to another process, while its counterpart can import that object. The exported object is queued into the import-list of the destination object (or process). Dynamic objects possess features for dynamic interaction with the enclosing environment.

One of the models for interaction among the concurrent computational objects for a distributed system is communication. Communication models provide interactions among independent objects while preserving encapsulation and information hiding in a computational object. "Export/import" constructs enable synchronous communication between the objects (or processes) by using dynamic objects. Once a dynamic object is exported, then the importing object can access this incoming object, and the exporting object (or process) cannot access this object anymore. Not only the logical address of this exported dynamic object has been changed, but the address space of this object has also been changed in the distributed computing system. This imported object responds to the messages with the new environment provided by the enclosing object (or process). This basic conceptual model is illustrated in Figure 2.4. (The corresponding language construct is shown later in the next chapter.)

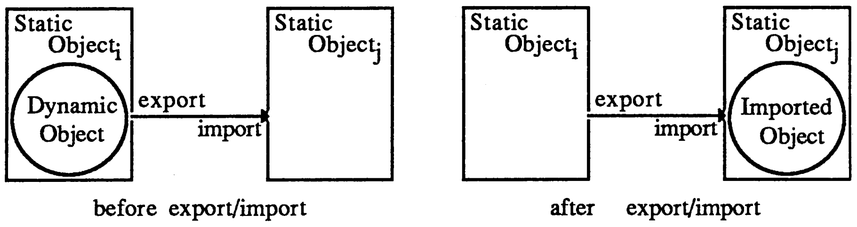


Figure 2.4. Object Migration Model

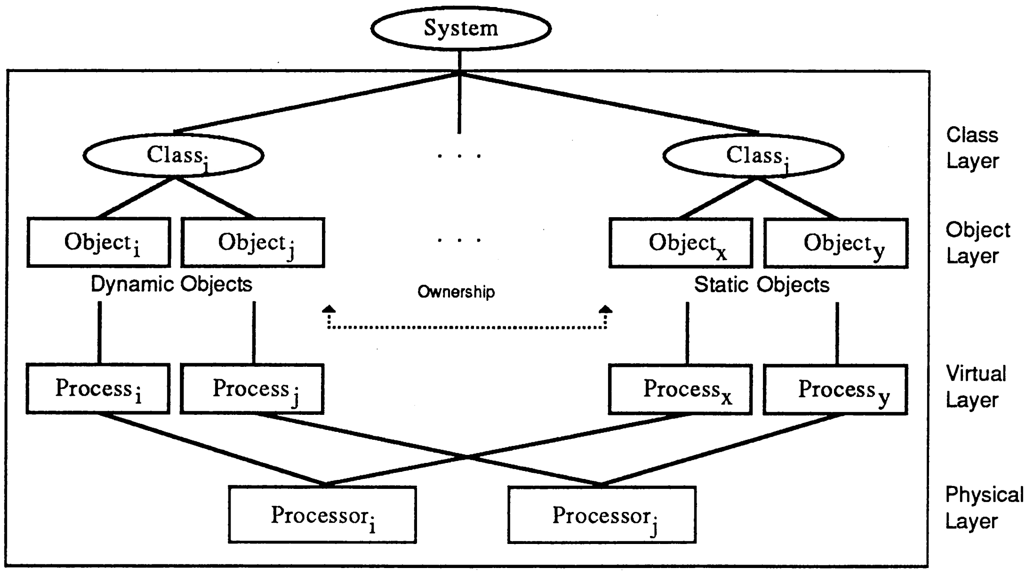


Figure 2.5. Parallel-C++ System View

2.4 Parallel-C++ System View

A system for our distributed object-oriented parallel programming language, Parallel-C++, can be implemented in various parallel computers. A system view for the implementation shows the general scheme of the implementation and mapping of the components in the system (Figure 2.5). A system is composed of class layer, object layer, virtual layer and physical layer. From the point of view of the object layer, a program consists of static objects and dynamic objects which are instantiated from their own classes in the class layer. Each object can be mapped into a process in the process layer. Processes reside on the physical processors.

2.5 Ownership

The relationship between the importing object and the corresponding dynamic object is called *ownership*. Ownership is a relation representing interactions between a static object and the corresponding dynamic object. When a dynamic object is exported to a certain static object, its ownership also changes. The definition of ownership is as follows.

2.5.1 Definition of Ownership

An object "s" *owns* an object "d" if and only if

(1) "s" alone can directly invoke methods of "d";

(2) "d" can directly invoke only methods of "s", and

(3) any communication between "d" and other objects must be directed through "s".

Then "s" is called an *owner* of "d".

An *ownership* between two disjoint sets of objects S and D, where S is the set of static objects and D is the set of dynamic objects, is a binary relationship (s_i, d_k), where $s_i \in S$ and $d_k \in D$. When a static object "s_i" exports a dynamic object "d_k" to another static object "s_j", ownership is changed from (s_i, d_k) to (s_j, d_k), where s_i , $s_j \in S$ and $d_k \in D$.

2.5.2 Properties of Ownership

If the static object "s_i" owns the dynamic object "d_k", then

(1) the static object "s_i" owns the communication control for dynamic object "d_k" exclusively, and

(2) the result of computation of dynamic object "d_k" is determined by the environment provided by static object "s_i".

Dynamic objects also follow static inheritance properties. A dynamic object looks up its own methods first. If the necessary methods cannot be found, then the dynamic object continues to look up the methods by tracing the inheritance hierarchy which has been statically defined and finds the inherited methods. The external environment of the dynamic object is provided by the static object. It can affect the evaluation of these methods. These properties are shown by distributed object processing and object migration.

(1) By distributed object processing, several concurrent objects are executable in parallel and they may exchange current states by means of communication among objects.

(2) By virtue of object migration, a dynamic object may react differently in different environments where it resides, while it keeps data integrity.

2.5.3 Example of Ownership

For example, in a real-time air-traffic control system, suppose that we have two airports "Dallas" and "OK_City", and we have two airplanes "American" and "United". "Dallas" and "OK_City" are instances of a class "airport", whereas "American" and "United" are instances of a class "plane". Ownership can be defined between the airports and planes. Ownership in this case indicates which airport is exclusively monitoring the plane. When these planes travel from one city to another, the ownership will be changed. Ownership shows this relation.

Let S = { Dallas, OK_City } and

D = { American, United }, then

the possible ownership relations are

R = { (Dallas, American), (Dallas, United),

(OK_City, American), (OK_City, United) }.

2.5.4 Characteristics of Ownership

(1) Ownership is different from inheritance. Inheritance hierarchy may or may not be the same as ownership hierarchy. Class hierarchy is static all the time. Ownership can be defined by any static objects and dynamic objects in the system. The owner of a dynamic object need not be an object instantiated from its super-class. The owner may be super, sub or a sibling in class hierarchy.

(2) Theoretically it is possible to have a hierarchy of objects related by ownership. In that case, all descendants defined by ownership also move when a dynamic object moves. (However, in our experimental language Parallel-C++, only one level of ownership is allowed.) Again the above descendants are not necessarily the objects instantiated from sub-class in class hierarchy.

(3) Reference to a dynamic object in a different address space (address in which other objects are) is only done by object migration. The "export/import" constructs support object migration (shown in the next chapter). Reference to an exported object is not allowed.

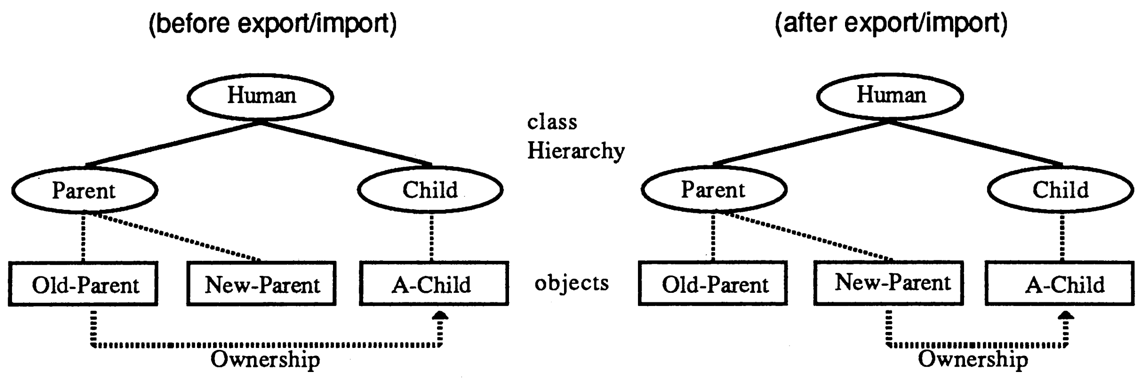
(4) Dynamic objects may respond differently to the methods according to the environments where they reside currently. We call this *Dynamic Interface Environment Binding* (DIEB). The DIEB facilitates a computation in different environments. A current owner provides a computation environment to a dynamic object.

(5) A global variable like a static variable cannot be defined, assigned or referenced in a dynamic object because the address space of the object is changeable at run-time.

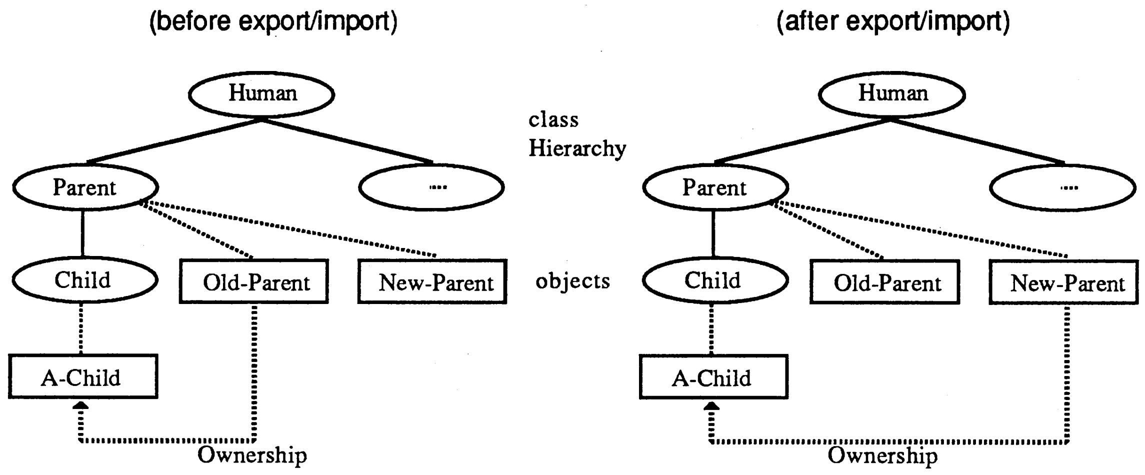
(6) When a dynamic object is exported, its state is also moved, and the address of the shared methods of its class can be found by tracing inheritance hierarchy, which is static in a system.

When an object moves to a new owner (importing object), its state is moved at the same time. This model preserves more of the properties of abstraction, encapsulation and information hiding than shared variables do. The information kept in an object is retained and the integrity of that object is preserved as it moves. This model represents situations that occur in many real-world situations. For example, in child adoption, the circumstance (environment of interaction) of a child has been changed, but his own characteristics and inclinations are preserved by the child initially. But new education can change his moral values and knowledge. Figure 2.6 depicts such an example that uses object migration. In Figure 2.6, classes are depicted by using ovals, and objects are depicted by using boxes. Super and sub classes are connected by lines. Thin dotted lines represent instantiation of objects from their respective classes. Thick dotted lines represent ownerships between dynamic objects and their owners. Figure 2.6<a> illustrates an ownership defined between the objects instantiated from sibling classes. Figure 2.6<b> illustrates an ownership defined between the object instantiated from a super-class and the object instantiated from a sub-class. The figures of left-hand side depict the situation before export/import operations, and the figures of right-hand side depict the situation after export/import operations.

Figure 2.7 shows various examples of ownership relations in detail. Also it illustrates the ownership relation when "export/import" operations are applied to the examples. The figures on the left, such as <a> and <b>, illustrate various ownership relations with the inheritance hierarchy. In the figures, arrows represent ownership relations. A dotted circle or a box represents an exported object instantiated from its class. A bold circle or a box represents an imported object from its class. We assume "P_i" is an exporting process and "P_j" is an importing process.



<a> Sibling Class Case



<b> Super-Sub Class Case

Figure 2.6. Ownership Hierarchy vs. Inheritance Hierarchy

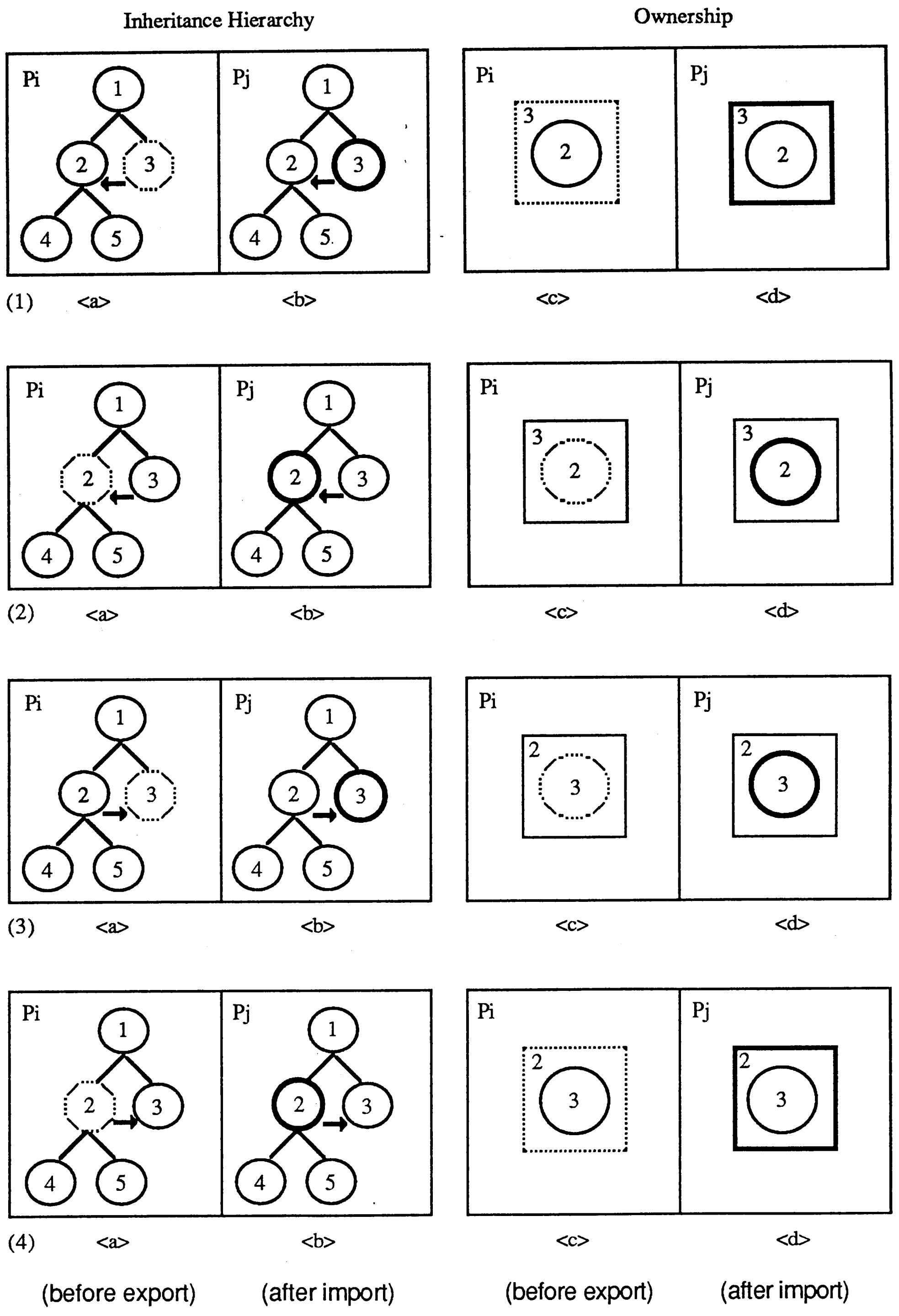


Figure 2.7. Various Ownership Relations with Inheritance Hierarchies

2.5.5 Some Questions and Answers

We may have several questions about distributed objects. Here, we list possible questions and suggest some solutions.

(1) Can any object be the owner of any other object?

Theoretically yes. But the owner should be defined in the syntax anyway. So not all objects are interchangeable dynamically between static and dynamic objects in a program. (Practically, for example, the owner is a static object enclosing dynamic objects as members in Parallel-C++. The objects used in an export construct can be assumed to be dynamic objects.)

(2) When (under what condition) does an object become an owner?

Theoretically, once a static object owns and gets communication control of dynamic objects, it becomes an owner. Syntactically, if a static object has dynamic objects like class objects as members, then ownership is declared. (Actually, there is no necessity for explicit syntax for static or dynamic objects in Parallel-C++, but once an object is declared in the same way as object-as-member, it is assumed that the enclosing object is static and the enclosed object is dynamic. Also a dynamic object can be found in an export construct.)

(3) When (how) does ownership change?

Once a dynamic object moves to another object, then ownership changes. If a dynamic object "d_k" moves from current owner "s_i" to a new owner "s_j", the ownership (s, d) may change thus:

$(s_i,d_k) \rightarrow (s_j,d_k)$.

Syntactically, this is done by "export/import" constructs. Dynamic interaction among distributed objects is done by ownership change. The computation of a dynamic object may be affected by the environment provided by its current owner. The external environment of a dynamic object is dynamically bound at run-time.

Inheritance hierarchy is static in Parallel-C++. Ownership change does not affect inheritance hierarchy at all. An object which moves to another environment is still an instance of class statically defined already. This conserves inheritance hierarchy all the time.

(4) How can communications be done among objects?

If a dynamic object is owned by an owner, all communication should go through such an owner. The owner provides a communication line from outside to the inner dynamic objects. This embedded communication provides a mechanism for information hiding and for Dynamic Interface Environment Binding (DIEB). Suppose we have different scenarios:

(4.a) Communication between dynamic objects and dynamic objects (Figure 2.8<a>):

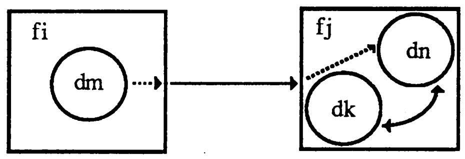
No direct communication between dynamic objects (for example, "d_m" and "d_n" in Figure 2.8<a>) is allowed, unless those dynamic objects reside at the same static object (for example, "d_k" and "d_n" in Figure 2.8<a>). The computation of a dynamic object is dependent on the enclosing environment provided by its owner. So the computation resulting from direct communication between dynamic objects may be different from the results expected in a different environment. Communication through the enclosing owner to dynamic objects ensures the results computed in the environment provided by the owner.

(4.b) Communication between dynamic objects and owner (Figure 2.8<b>):

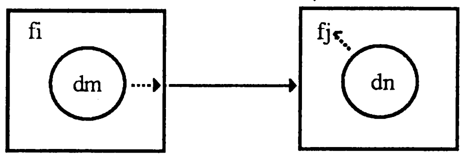
An owner can communicate with its dynamic objects using the methods of these dynamic objects defined already (for example, "f_j" communicates with "d_n" in Figure 2.8<b>). An owner cannot communicate directly with dynamic objects owned by other objects. Communication must go through the communication line provided by their owner (for example, "d_m" communicates with "f_j" through the communication line provided by "f_i" in Figure 2.8<b>).

(4.c) Communication between owner and owner (Figure 2.8<c>):

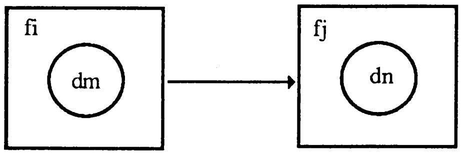
Communication between owners is done by calling each other's methods (for example, the communication between "f_i" and "f_j" has been done in conventional way in Figure 2.8<c>).



<a> Dynamic - Dynamic Communication



<b> Dynamic - Owner Communication



<c> Owner - Owner Communication

Figure 2.8. Various Communications among Distributed Objects

2.6 Contribution and Discussion

In the previous sections we have shown how object-oriented programming can be extended a level further by incorporating *semi-persistent dynamic objects*; semi-persistent because their lifetimes can be spanned by the lifetimes of more than one object. The totality of objects in the system consists of static objects and dynamic objects. A static object can be a master object which owns some dynamic objects which may be floating from one object (process) to another object (process) in the system. This situation occurs very frequently in the familiar real-world, examples being aircrafts (controlled by distributed airports) in an air-traffic-control-system, communication messages (passed by distributed nodes) in local area networks and process migrations (from one resource to another) in load balancing (resource sharing), etc.

Object mobility can be obtained as a consequence of the change in *ownership relation* between objects which defines a new *dynamic relation* among objects. An ownership relation between static and dynamic objects can be changed while conserving the static class hierarchy.

When the ownership of an object is changed due to export, the importing object may own this exported object. Ownership does not mean the overlay of their address spaces (e.g., object frames and activation records). Static and dynamic objects have their own address spaces. The ownership concept is orthogonal to the inheritance notion and hence does not interfere with the class hierarchy. To be truly object-oriented parallel programming, the conflict between concurrency and inheritance should be avoided. In Parallel-C++, the inheritance is static. The class hierarchy is constructed at compile time, and the system keeps its class-list, object-list, and class hierarchy at all times. When a method is called by an object, the system looks up the method table for the object. If the method looked up is not found at the current object, the system traces the class hierarchy already defined, and finds the desired method from the super-class. Even if the dynamic object migrates to another static object, the class hierarchy is still conserved and the inheritance can be traced. The instance variables in objects are not shared data.

A static object (or process) can export its objects to another static object (process), while its counterpart can import that object. The exported object is placed in the import-list of the destination object (or process). When objects are exported and imported among the processes, the environments of the objects will change. Object environment includes storage for the object frame, which consists of object name, class name, frame size, instance variables, pointer to the address of member functions, and other necessary information for the management of the object. If an object is exported, then the exporting object purges it from its environment. The importing object establishes an environment for it to communicate with the imported object. There are two kinds of environments for a dynamic object, *internal* environment and *external* environment. The internal environment of an object consists of its data structures and methods, including the ones it obtains via inheritance. The external environment is the environment enclosing the dynamic object. The external environment depends on the static object which owns the dynamic object. The external environment affects the communication between the static and dynamic objects. The internal environment of the dynamic object moves with it, but the external environment does not. The internal environment of an exported object is bound to the external environment which is provided by the importing object. By this *Dynamic Interface Environment Binding* (DIEB), an object behaves differently according to the environment in which it resides. Ownership guarantees encapsulation and protection of dynamic objects. The basic implementation scheme including the run-time memory management for this new concept is more specifically described in Chapter 4 and Chapter 5.

Parallel-C++ adds enhancement to the scope of the objects. An object in C++ may have either static or dynamic scope. In C++, an object can be instantiated from the beginning [Figure 2.9<a>] or in the middle of a program [Figure 2.9<b>] and it still lives until the end of program unless the destructor operation of the class is issued to kill it. Once the object's life ends, it cannot be restored again. In Parallel-C++, however, a dynamic object can be exported at any time from any process that owns it [Figure 2.9<c>], and the process which imports this object inherits the environment [Figure 2.9<d>], and the object can also be imported again with the current states. If the object is not exported, then its scope ends within the current owner. Moreover, the lifetime of a dynamic object extends over all the importing objects. In general, static objects have static scope.

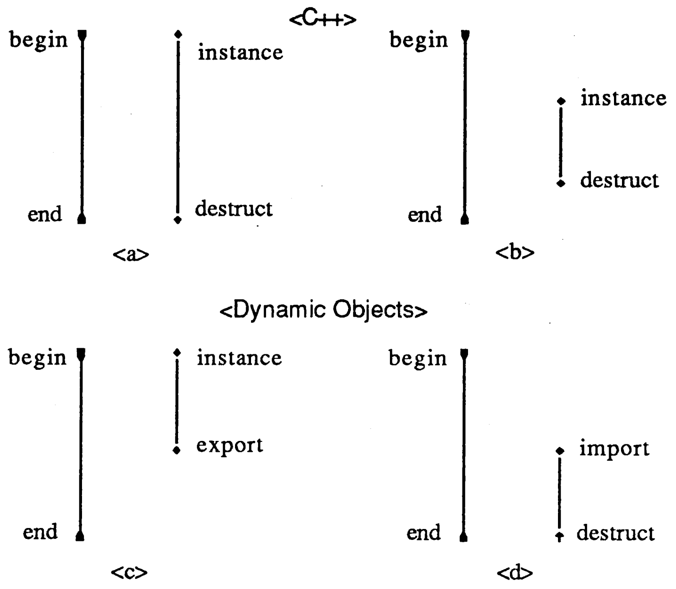


Figure 2.9. Comparison of the Scope of Objects

The language constructs "export/import" are different from Ada "rendezvous" [Ada 79,83] from the point of view of control. Figure 2.10 illustrates the differences in control flow between this construct and related constructs, such as "functions", "coroutines" and Ada "rendezvous". Programming using dynamic objects can simulate other control flow schemes provided by those related constructs.

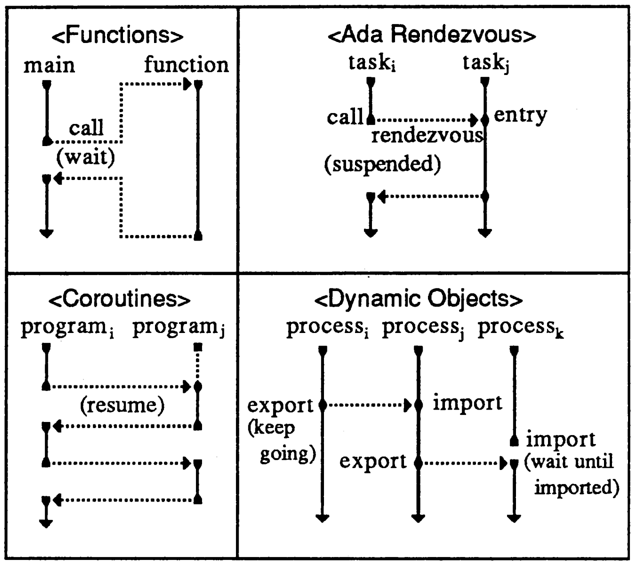


Figure 2.10. Comparison of Flow of Controls

Parallel-C++ language definitions and examples are given in the next chapter.

CHAPTER III

PARALLEL-C++ : LANGUAGE DEFINITION

3.1 Language Definition

A programming language has three main characteristics [Schmidt 86]:

(1) Syntax: the appearance and structure of its sentences,

(2) Semantics: the assignment of meanings to the sentences, and

(3) Pragmatics: the usability of the language.

In this chapter, we define the syntax and semantics of an object-oriented parallel programming language, Parallel-C++. We also show some examples of the usage of each new language construct. Implementation schemes for this language are shown in the following chapters. Language definitions through the syntax and semantics of the language should verify the correctness of the implementations.

3.1.1 Parallel-C++ Overview

The programming language C++ [Stroustrup 86] is a superset of the programming language C [Kernighan and Ritchie 78,88]. Parallel-C++ is an extension of C++ for allowing object-oriented programming in parallel and distributed programming environments [Jo and George 89,91]. Since the language definition for the serial programming part of Parallel-C++ is almost the same as that of the C and C++ languages, the language definition in this section does not provide a detailed description of the serial part of the language. Even though we do not provide all of the language features, we describe the language features which are new and which are different from existing languages. The language definitions that minimally show the salient features of the new language are described here.

Even though we mostly explain new language constructs by using existing language features, we try to avoid depending entirely on those languages that are sometimes too specific. The reason for our approach, which provides independent (or portable) modules of new language grammar, is to allow readers to import these ideas into their own models efficiently.

3.1.2 Scope Rules

A Parallel-C++ program consists of one or more translation/compilation unit(s) stored in one or more file(s). Files for translation/compilation should have the name with "*.c" for source programs and "*.h" for header files. The current implementation of Parallel-C++ expects header files to be defined before the translation/compilation of source program.

The scope rules for variable names, except dynamic objects, in a program usually follow the conventional scope of C++. But the scope of a dynamic object is local to its owner which is a static object. Variables declared in a dynamic object are local to that object. Export/Import operations may change the scope of a dynamic object. Once a dynamic object is exported, it is no longer available to its owner. The scope of an exported dynamic object is limited to the object importing such a dynamic object.

3.1.3 Keywords for Parallelism

In addition to the keywords defined in C++, Parallel-C++ has reserved some identifiers for use as keywords for object-oriented parallel programming. Those are the following.

keywords		Usage
parbegin	parend	// parallel statements
autobegin	autoend	// automatic processor allocation
parallel_commands	end_parallel_commands	// explicit process allocation
process	end_process	// explicit process definition
parallel_functions	end_parallel_functions	// parallel function calls
send	recv	// synchronization
export	import	// object migration

3.1.4 System Library

The functions, types and macros of the standard library are declared in system library files. Besides these, Parallel-C++ provides other header files, "msg.h", "token.h", "global.h" to support translation option. Because the Parallel-C++ translator automatically includes these files when a user performs a translation, so programmers need not explicitly include it. The header file "global.h" is included by the translator, compiler and interpreter programs. The header file "msg.h" defines functions and classes for message passing. Another header file "token.h" is for declaration of token codes used in translation. The "global.h" file declares global data structures and defines other necessary functions and classes. Other system files and header files, for the translation and compilation, are specifically described later in the sections describing implementation issues. Users can be concerned with only the system standard libraries or user-defined files which they use with translation/compilation.

3.2 Language Extension for Parallelism

To achieve explicit parallelism with an existing serial object-oriented language (such as C++), some extensions to that language are necessary. Those language extensions include the constructs for parallel statements, automatic process allocation, parallel function calls, synchronization, explicit process allocation and object migration. Here we define syntax and semantics for those new language constructs which we have designed and incorporated in Parallel-C++. In the syntax definition, we use "Gothic" letters for the keywords (which are mostly new constructs), and "*Italic*" letters for the non-terminal symbols. The symbol ":" is used as the meaning of "is defined as", and the symbol "|" is used as "or". The symbol "$_{opt}$" appears in the right-lower side of non-terminal means "optional". We do not go through all the symbols at the terminal symbol level, because those parts are conventionally well defined in the existing base languages, which are here C and C++. Those parts of the grammar are skipped and denoted using the "⋮ " symbol. An example usage of each language construct is also provided. We show here six kinds of typical extensions. We assume the newly extended grammars are derived from the non-terminal "statement".

```
a-program      :     . . .
⋮
statement      :     . . .
               |     parallel-statement
               |     automatic-processor-allocation-statement
               |     parallel-function-call-statement
               |     synchronization-statement
               |     explicit-process-allocation-statement
               |     object-migration-statement
⋮
```

The following sections of this chapter are dedicated to the definition of the language extensions of an object-oriented programming language while achieving parallelism.

3.3 Parallel Statements

A "parallel statements" construct allows parallel execution of the statements enclosed by it.

3.3.1 Syntax

parallel-statement	:	parbegin *compound-statement* parend
compound-statement	:	{ *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }
declaration-list	:	*declaration*
	|	*declaration declaration-list*
statement-list	:	*statement*
	|	*statement statement-list*

⋮

3.3.2 Usage

```
parbegin { statements } parend;
```

where	parbegin:	parallel statements begin
	parend:	parallel statements end
	statements:	optional parallel statements

3.3.3 Semantic Notes

(1) The statements in this construct are concurrently executable, and may execute in parallel at the run-time.

(2) Syntactically speaking, a "parallel statements" construct can be nested.

(3) When using a "parallel statements" construct, a user should guarantee that the execution of parallel statements in this construct are independent and mutually exclusive of each other to be parallelized successfully.

(4) The variables defined or used in the construct are strictly local.

(5) A parallelizing compiler binds the divided part of statements inside into one or more parallel blocks, as many as the number of processors that are allowed to use simultaneously in the system.

(6) Fine-grained parallelism is supported.

3.3.4 Example

Suppose we have a program segment like the following:

```
        parbegin
        {
S0:             a = b + 1;
S1:             c = d;
S2:             e = f * pi;
        }
        parend;
```

All three statements, S_0, S_2 and S_3, are independently executable, because there is no dependency among the variables used in the statements. However, the example given below illustrates the improper use of the construct.

```
        parbegin
        {
S0:             x = y + 1;      // related to S2
S1:             w = z;
S2:             y = x * pi;     // error: dependent on S0
        }
        parend;
```

In this example, S_0 and S_2 are dependent on each other. Because the variables "x" and "y" are used in both statements, the results may be different according to the order of their executions. The dependency checking by programmers is not an easy task unless the number of statements used in the construct is very small. In practice we need a parallelizing compilation facility to check program dependency automatically and to provide the capability to restructure programs. The next section shows a basic construct for doing such automatic checking and restructuring.

3.4 Automatic Processor Allocation

User may use an "automatic processor allocation" construct to diminish the burdens in checking the dependencies among the statements in a "parallel statements" construct. The automatic restructurer in compilation step restructures the statements in such a construct into parallelized blocks of statements after interdependency checking. At run-time the restructured blocks of statements are executed in parallel.

3.4.1 Syntax

```
automatic-processor-allocation-statement:
                              autobegin compound-statement autoend;

compound-statement  :         { declaration-list_opt statement-list_opt }

declaration-list    :         declaration
                    |         declaration declaration-list

statement-list      :         statement
                    |         statement statement-list

⋮
```

3.4.2 Usage

```
autobegin  {  statements  }  autoend;

     where  autobegin:    automatic restructuring begin
            autoend:      automatic restructuring end
            statements:   optional statements to be parallelized
```

3.4.3 Semantic Notes

(1) The statements in this construct are concurrently executable if no dependency is found, and may execute in parallel at run-time.

(2) If all statements to be parallelized in this construct are mutually independent, this has the same effect as a "parallel statements" construct. Even in this case, the blocks of restructured statements may not be the same as those of a "parallel statements" construct.

(3) The optimizer module in the parallelizing compiler draws a virtual graph during the checking of dependencies among the statements in the construct,

then restructures the program by blocking the dependent statements into several independent blocks.

(4) The run-time interpreter (Automatic Processor Allocator) executes the parallelized statements by running the independent blocks of codes on the parallel processors.

(5) Fine-grained parallelism is supported.

3.4.4 Example

For the following program segment:

```
S0:   .... ;
      autobegin
      {
S1:           a = x + 1;          // related to S4
S2:           b = x * pi;         // related to S3
S3:           c = b - 1;          // dependent on S2
S4:           z = a / b;          // dependent on S1 and S2
      }
      autoend;
      :
Sn:   .... ;
```

Thus program segment will be restructured like the following:

```
S0;
parbegin
{
      { S1; }             // Block1: on the processori
      { S2; S3; }         // Block2: on the processorj
}
parend;
S4;
:
Sn;
```

The parallelized program segment will be executed in the sequence of: S_0 first, $Block_1(S_1)$ and $Block_2(S_2$ and $S_3)$ in parallel, then S_4, and S_n finally.

The constructs, which have been described, are able to support fine-grained parallelism. Independent blocks of codes can be modularized in large size. The next construct is for parallel programming of large independent modules.

3.5 Parallel Function Calls

For the parallel execution of independent medium or large modules of program, the construct "parallel function calls" is provided. Independent sets of functions can be invoked in parallel by using this construct. Some research work has been reported in a similar area [Banerjee 86] [Li and Yew 88].

3.5.1 Syntax

```
parallel-function-call-statement :   parallel_functions
                                     {  function-statement  }
                                     end_parallel_functions;

function-statement      :    function-call_opt function-statement_opt

function-call           :    function-name ( parameter-list_opt );
                        |    return-var = function-name ( parameter-list_opt );

function-name           :    identifier

return-var              :    identifier

parameter-list          :    parameter
                        |    parameter, parameter-list

:
```

3.5.2 Usage

```
parallel_functions
{
        function-name_i  (parameter-list);
        :
        return-var = function-name_n (parameter-list);
}
end_parallel_functions;
```

where	parallel_functions:	parallel function call begin
	end_parallel_functions:	parallel function call end
	function-name$_i$:	calling function name
	parameter-list:	optional parameter list
	return-var:	return variable

3.5.3 Semantic Notes

(1) The functions in this construct may be concurrently invoked and executed in parallel at run-time.

(2) Only function call statements can appear in this construct.

(3) Access to global variables from a "parallel function calls" construct is not allowed unless the synchronization construct is used (shown in Section 3.6).

(4) Theoretically, the control flow has as many concurrent threads as the number of functions in this construct (However practically, it can have only as many parallel threads as the number of available processors to which parallel function calls can be mapped.).

(5) The maximum speed-up obtained is proportional to the number of processors assigned to this construct.

(6) The processes, which were allocated to some parallel functions and finished their execution, wait until other processes are finished. All processes used with this "parallel function calls" construct are synchronized (or joined) at the end of this construct. The "end_parallel_functions" statement acts as a barrier.

(7) The use of this construct in a recursive fashion is not permitted.

(8) Coarse-grained parallelism is supported.

3.5.4 Example(1)

Suppose we have the following program segment, with two functions, "func1" and "func2". In the "parallel function calls" construct, those two functions are concurrently invoked (in S_1 and S_2), executed in parallel, and synchronized at the end of the construct. After joining from concurrent threads, the next serial statement S_3 is executed.

```
            :
            int func1(int x)        { x++;              return x; }
            int func2(int y)        { y = y + 10;       return y; }
            :
            main()
            {
                    int arg = 0;  int sos = 0;  int sub1 = 0;  int sub2 = 0;
                    :

                    parallel_functions
                    {
                            :
S1:                         sub1    =    func1(arg);    // sub1 = 1;
S2:                         sub2    =    func2(arg);    // sub2 = 10;
                            :
                    }
                    end_parallel_functions;

S3:                 sos     =     sub1  +  sub2;        // sos: sum of subs = 11
                    :
            }
            :
```

3.5.5 Semantic Discussion(1)

In the example in Section 3.5.4, the parallel statements S_1 and S_2 can be executed either in the same processor or in different processors. In a distributed memory system, it is possible that the statements S_1 and S_2 execute in parallel on different processors with their own local memories. Then the variable "sos" in the statement S_3 in the main program may not have the correct answer "11" unless the values of variables, "sub1" and "sub2", in the main processor memory have been updated with new values somehow. At least the values of "sub1" and "sub2" in the statement S_3 would have to be updated when the parallel statements are joined just before the execution of statement S_3. The compiler or translator should generate the codes necessary to return the current values of these variables executed in different processor's memory to the main processor's memory. Even if the current values of variables in their local memories have been updated, the variables in the main processor's memory have not been updated yet. For example, the following pseudo-code segment is generated by the Parallel-C++ translator between the end of the parallel function calls construct and the statement S_3 in the restructured program. In distributed memory system without any shared memory, every communication between different address spaces should be done by message passing only. The constructs, "send" and "recv", used here for communications, are defined and explained in the next section.

```
:
if(my_node == process_for_S1)                   // return result from S1
        send(sub1, main_process, msg_type_1);
:
if(my_node == process_for_S2)                   // return result from S2
        send(sub2, main_process, msg_type_2);
:
if(my_node == main_process)                     // receive the results
{
        recv(process_for_S1, sub1, msg_type_1);
        recv(process_for_S2, sub2, msg_type_2);
}
:
```

3.5.6 Example(2)

Can we send a parameter "arg" and receive a return value in the same "arg"? This example exhibits more of the difficulties with parallel evaluations in the distributed system. The following example provides the motivation to examine this problem. In this example we have used the same variable "arg" in both parallel function calls by referencing its address. This causes non-deterministic results from their parallel execution.

```
:
void func1(int *x)      { (*x)++;                          }
void func2(int *y)      { (*y)= (*y) + 10;      }
:

main()
{
        int arg = 0;
        :

        parallel_functions
        {
                :               // possible arguments  →   possible results
S1:             func1(&arg);    // arg = {0 | 10}      →   {1 | 11}   ?
S2:             func2(&arg);    // arg = {0 | 1}       →   {10 | 11}  ?
                :
        }
        end_parallel_functions;

S3:     arg++;                  // arg = {2 | 11 | 12}  ?
}
```

3.5.7 Semantic Discussion(2)

In the example in Section 3.5.6, we passed parameters by address for parallel function calls. Since the sequence of execution of parallel function calls is non-deterministic, the function "func1(&arg)" may take the value of "arg" as either "0" (before the statement S_2 is executed) or "10" (after the execution of S_2) and the function "func2(&arg)" may take the value of "arg" as either "0" (before the statement S_1 is executed) or "1" (after the execution of S_1). To make it worse, after the execution of both functions, "arg" has the result of either "1" or "11" (by S_1), or either "10" or "11" (by S_2). Thus the execution of the statement S_3 results in "2", "11", or "12" for the value of "arg" according to its previous value. So we need to guarantee that the value of argument always be correct, here with the initial value of "0". The conditions to be satisfied by parameters of functions in a "parallel function calls" construct are the following:

(1) the same variable cannot be passed as argument for more than one function when we use "pass-by-address" in a parallel function calls construct. (If we need to, another copy of the variable should be used for it.),

(2) the same variable or address cannot be used for both arguments and return variables, and

(3) the same return variables for each parallel function statement cannot be used. Each return variable should be distinct.

We know they may not cover all the issues to be discussed, but those rules are the minimum restrictions to get integrity of data from parallel function evaluations.

3.6 Synchronization

For communication between parallelized blocks and objects, the synchronization constructs "send" and "recv" are provided.

3.6.1 Syntax

```
synchronization-statement   :   send-statement
                            |   recv-statement

send-statement   :  send (message-id, destination-process-id, message-type);
                 |  send (message-id, destination-process-id);

recv-statement   :  recv  (source-process-id, message-id, message-type);
                 |  recv  (source-process-id, message-id);

message-id                  :   identifier | constant

message-type                :   identifier | constant

source-process-id           :   identifier | constant

destination-process-id      :   identifier | constant

⋮
```

3.6.2 Usage

send (*message-id, destination-process-id, message-type*);

recv (*source-process-id, message-id, message-type*);

where	send():	message sending statement
	recv():	message receiving statement

message-id:	message identifier to be sent
message-type:	optional message type
source-process-id:	process identifier sending a message
destination-process-id:	process identifier receiving a message

3.6.3 Semantic Notes

(1) The "send" construct is for sending a message to the destination process.

(2) The "recv" construct is for receiving an incoming message from the source process.

(3) The communication is either synchronous or asynchronous.

(4) In the case of synchronous communication, the source process sends a message to the destination process by hand-shaking.

(5) In the case of asynchronous communication, a message sent from a source process is placed in the message-queue of the corresponding destination process. The destination process retrieves the message, with the expected message type, and which has been sent from the process defined by "source-process-id". The process that issued the "recv" command postpones its execution until receiving such a message, if no same-typed message has arrived yet at the message-queue.

(6) Process identifiers and optional message types are used to identify the proper messages.

3.6.4 Example

A usage example of this construct is shown with an example of the "explicit process allocation" construct in the next section.

3.7 Explicit Process Allocation

Parallel-C++ also provides a construct to support explicit process allocation by users. This construct is based on the "parallel compound" construct [Dijkstra 68], the "guard and parallel commands" of Hoare's CSP [Hoare 78] and Ada "tasks" [Ada 79,83]. This construct is useful when we explicitly allocate parallel processes to some portions of a program which can be executed concurrently.

In a distributed processor architecture, assuming the availability of sufficient processors, each process can be mapped into a physical processor during execution. This feature illustrates one of the differences between Parallel-C++ and other C++ based languages.

3.7.1 Syntax

```
explicit-process-allocation-statement  :
                                   parallel_commands (number-of-processes)
                                   { process-statements }
                                   end_parallel_commands;

process-statements       :         process-statement
                         |         process-statement process-statements

process-statement        :         process (process-id)
                                        compound-statement
                                   end_process;

compound-statement       :         { declaration-list_opt statement-list_opt }

number-of-processes      :         identifier | constant

process-id               :         identifier | constant

declaration-list         :         declaration
                         |         declaration declaration-list

statement-list           :         statement
                         |         statement statement-list

   :
```

3.7.2 Usage

```
parallel_commands (number-of-processes)
{
        process (process-id)    { statements; }  end_process;
        :
        process (process-id)    { statements; }  end_process;
}
end_parallel_commands;
```

where	number-of-processes:	total number of processes in the construct.
	process-id:	process identifier
	statements:	optional statements

3.7.3 Semantic Notes

(1) This construct allows users to allocate parallel processes explicitly to the concurrent portion of the programs.

(2) All the processes may execute concurrently and independently at run-time. They are capable of communicating with each other.

(3) Parallel_commands can be nested.

(4) "Number_of_processes" indicates the total number of processes in the construct.

(5) The process identifier is an identifier or an integer constant.

3.7.4 Example

In this example, two concurrent processes, "process(P0)" and "process(P1)", which perform miscellaneous calculations in parallel, are defined by using an explicit process allocation construct. During the calculations, they communicate with each other by using the synchronization constructs, "send" and "recv".

In this example, the "process(P0)" sends a value "0" of variable "x" to the "process(P1)", and it waits until the "process(P1)" sends a message. The "process(P1)" receives the value of "x" from the "process(P0)" through "z", evaluates an expression using "z", sends the current value of "z" to the "process(P0)", and proceeds to the next statement. The "process(P0)" receives the value of "z" through "y", and executes the remaining computations.

```
:
parallel_commands (2)
{
        process(P0)     {
                                x = 0;
                                send(x, P1);
                                recv(P1, y);
                                x = y + 1;
                                cout << x;                      // x = 2
                        }       end_process;
        process(P1)     {
                                recv(P0, z);
                                z++;
                                send(z, P0);
                                cout << z;                      // z = 1
                        }       end_process;
}
end_parallel_commands;
:
```

3.8 Object Migration

One of the valuable ideas incorporated in Parallel-C++ is distributed static/dynamic objects. Their salient features have been described in Chapter 2. Dynamic objects support distributed computing without losing important features of object-oriented programming, such as inheritance and information hiding. The constructs "export/import" support migrations of dynamic objects among static objects in a distributed computing system.

3.8.1 Syntax

object-migration-statement	:	*export-statement*
	|	*import-statement*
export-statement	:	export (*object-id*, *destination-object-id*);

import-statement	:	*import-object-variable* = import (*class-id*);
	|	*import-object-variable* = import (*class-id*, *source-object-id*);
source-object-id	:	*object-name* | *process-name*
destination-object-id	:	*object-name* | *process-name*
import-object-variable	:	*object-name*
class-id	:	*class-name*
object-id	:	*object-name*
class-name	:	*identifier*
object-name	:	*identifier*
process-name	:	process (*identifier*) | process (*constant*)

⋮

3.8.2 Usage

export (*object-id*, *destination-object-id*);

import-object-variable	=	import (*class-id*);
	|	import (*class-id*, *source-object-id*);

where	object-id:	exported object identifier
	class-id:	class identifier for the imported object
	import-object-variable:	object name for the imported object
	destination-object-id:	destination (importing) object identifier or process name
	source-object-id:	source (exporting) object identifier or process name

3.8.3 Semantic Notes

(1) The "export" construct exists to export an object to other objects (or processes). The exported object is placed in the import-list of the destination object (or process).

(2) The "import" construct is for importing any object of the given class and binding it to the import-object-variable. Giving class identifier and import-

object-variable allows a more generic form of anonymous object instantiation than giving a specific object.

(3) The object (or process) that issued the "export" statement can continue its execution even if no object (or process) imports the exported object.

(4) When the importing object (or process) issues the "import" statement, however, if no object (or process) exports such an object, then the importing object (or process) must postpone execution until a required object arrives at the import-list.

(5) If an object has been exported once, it cannot be exported again (when the "*copy-and-delete*" rule is applied); if such an object is needed again, it can be imported. (For example, returning a result from an object which has performed a certain calculation to the requesting/client object, which here means that the exporting object re-issues an object import.)

(6) If a "*copy-and-remain*" rule is applied after a copy of an object is exported out, the same object is still available in the current owner. This kind of scheme is useful for some applications such as message broadcasting and process copy. However, the current Parallel-C++ implementation adopts a "copy-and-delete" rule only.

(7) The necessary codes for export/import objects, such as object construction from a given class name, initialization and copying information for an importing object, are generated by the parallelizing compiler (or translator), and these are executed by the run-time interpreter.

(8) "Source-object-id" or "destination-object-id" may be either a static object identifier or a process name.

(9) "Class-id" is a class name of the imported object.

3.8.4 Example

Revisiting the child adoption example that we have mentioned in Chapter 2 (Figure 2.6 in Section 2.5.4), assume that a child has been adopted.

```
:
parallel_commands(2)
{
        process(Old_Parents)
        {
                child A_Child;                               // class "child"
                export(A_Child, process(New_Parents));       // adoption
        } ...
        process(New_Parents)
        {
                New_Child = import(child);                   // adopted
                New_Child.education();
        } ...
}
end_parallel_commands;
:
```

In the example code, a child object "A_Child" is exported by a process "Old_Parents", and it is imported by another process "New_Parents". An imported child has been initiated by the object "New_Child" in the importing process. Even if the circumstance of this child has been changed, his own characteristics are kept. (Even if the environment of the child object has been changed from "Old_Parents" to "New_Parents", "A_Child" object's own states are kept in "New_Child" object.) Only the importing process "New_Parents" (by assuming that such a process is a static object) can change the states of its imported object "New_Child" (here, by using the "education()" method). If the object "A_Child" is once exported, the exporting process "Old_Parents" cannot directly access the states of the exported object, unless it uses a communication provided by the current owner "New_Parents" (by assuming that such a process is an owner object). A discussion of issues related to export and import of dynamic objects is found in Chapter 2.

The next chapter describes the implementation scheme of a translator emphasizing the translation of the salient features of Parallel-C++.

CHAPTER IV

IMPLEMENTATION SCHEME I (TRANSLATOR)

This chapter describes the implementation of a translator for the language Parallel-C++ [Jo et al. 91] on the Intel iPSC/2 hypercube multiprocessor computer [Intel 88]. The translation scheme takes advantage of an existing C++ compiler and system calls for the iPSC/2. We focus on the implementation of two important language constructs.

(1) The language construct "*parallel_commands*" - which provides support for *explicit concurrent process allocation.* The concurrent processes can be mapped into parallel processors at run-time.

(2) The language constructs "*export/import*" - which provide support for *object migration* between objects and processes in the system.

We describe the implementation schemes in general and translation of the salient features of Parallel-C++ in particular. Relevant algorithms and a complete example are also given.

4.1 Overall Structure of the Parallel-C++ Translator

The Parallel-C++ translator translates a Parallel-C++ source code into a collection of C++ programs, system primitives and subroutines for the Intel iPSC/2 [Intel 88]. The overall structure of the translator is shown in Figure 4.1. Borrowing existing facilities from the C++ compiler [AT&T 85,86,89,89b] and system utilities, the Parallel-C++ environment provides a more sophisticated tool to implement problem-solving techniques (e.g., simulation) and in some cases a more efficient code. Even though the translation scheme takes advantage of existing facilities, the new language features incorporated into Parallel-C++ make the translator design a difficult task.

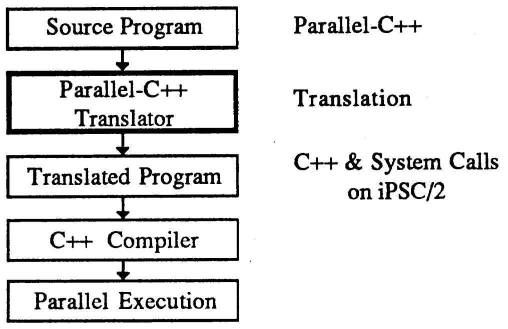


Figure 4.1. Structure of Parallel-C++ Translator

The Parallel-C++ translator extracts parallelism using the parallel constructs provided by the language and generates the necessary parallelized C++ codes by adding system primitives and library functions. The translator works as follows:

(1) reads an input Parallel-C++ source program,

(2) identifies parallel sections,

(3) performs pre-optimization (for parallel code generation),

(4) performs parallel code generation,

(5) performs post-optimization (for efficient and pretty code generation), and

(6) outputs parallelized C++ source program supplemented with the system calls and library functions on the target machine.

The Parallel-C++ translator can be viewed as a high-level language transformer. It translates Parallel-C++ code into source code for a C++ compiler for the iPSC/2. To facilitate parallel execution in the hypercube, the original source program written in Parallel-

C++ is transformed using several steps. One of the steps is to identify parallel sections in a program. Parallel sections are transformed into node programs for the iPSC/2 that can be executed concurrently.

4.2 Base machine for the Implementation: Intel iPSC/2

The Intel's Personal Super Computer (referred to as the iPSC/2) [Intel 88] is one of the parallel processing computer systems [Almasi and Gottlieb 89] [Dongarra 87] [Fox 88,89]. The iPSC/2 is a multiple-instruction, multiple-data (MIMD) stream machine that uses message-passing communication, using distributed memory in each node. The system consists of four main functional components - a cube, processing nodes, a system resource manager(SRM) and a network. The nodes are connected together to form a cube. Each node contains a 32-bit microcomputer using an Intel 80386 processor, an 80387 numeric coprocessor, an optional VX vector processor and an optional SX scalar processor with a local memory capacity ranging from one to sixteen megabyte(s). Currently the Parallel-C++ translator and compiler are dependent on the architecture of the iPSC/2 system. But the concepts of the parallel language constructs in Parallel-C++ can be adapted to implementation on other systems such as distributed memory parallel machines and supercomputers, with few modifications in the synchronization and other requirements of their compilers and translators.

4.3 The Current Implementation of the Translator

The current implementation of the Parallel-C++ translator is written in C++ on the iPSC/2. The translator consists of a main program and several header files (Figure 4.2). The main program "main.c" generates several parallelized C++ programs. A header file "token.h" defines tokens to be used for detecting parallel segments in a program, and another header file "global.h" defines global data structures and other necessary programs including function/class definitions used for string manipulation, nesting structure detection, fork-join concurrent constructs and Automatic Processor Allocation Tree (APA Tree).

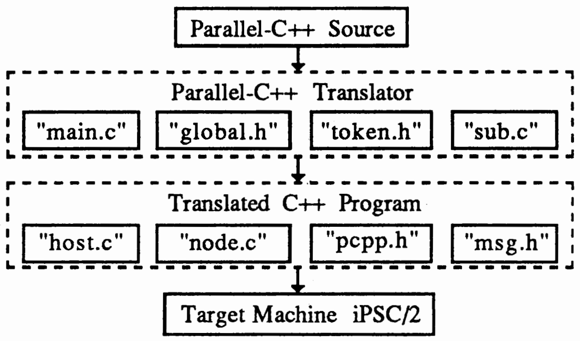


Figure 4.2. Current Implementation of Parallel-C++ Translator

A Parallel-C++ program is translated into several C++ source programs accompanied with header files. The translated program consists of "host.c", "node.c", "pcpp.h" and "msg.h". Two programs, "host.c" and "node.c", are needed for a system like the iPSC/2. Generally the "host" program acts as a supervisor, while the "node" program actually performs most of the calculation. The main part of the Parallel-C++ program is separated into these two parts as necessary. The "pcpp.h" is a header file generated from the Parallel-C++ source file to be used by those, "host.c" and "node.c", programs. It generally includes header files and function/class definitions in a source program. The "msg.h" is a header file provided by the translator for data structures and some definitions used at run-time.

The translation schemes used, and the corresponding algorithms, are described in the following sections.

4.4 Concurrent Processes

In Parallel-C++, the "parallel_commands" construct is provided to support explicit process allocation (discussed in Section 3.7). It provides support for the explicit generation of concurrent processes that can be executed in parallel on a multiprocessor machine. The "parallel_commands" construct combined with "object migration" construct (discussed in Section 3.8) provides a programming environment for distributed applications such as load balancing. The syntax and semantics of such language constructs are defined in the previous chapter. The usage of the construct is briefly shown again in Figure 4.3. Here we recall that all of the processes in the construct may execute concurrently and independently at run-time. These concurrent processes are capable of communicating with each other. The parameter "number of processes" indicates the intended total number of concurrent processes in the construct.

```
parallel_commands (number-of-processes)
{
        process (process-id)   { statements; }  end_process;
        :
        process (process-id)   { statements; }  end_process;
}
end_parallel_commands;
```

Figure 4.3. Explicit Process Allocation Construct

Figure 4.4 shows a simple illustrative example of the use of explicit process allocation in a program segment. The program consists of two processes (P_0 and P_1). Each process consists of a block of sequential statements (S_i and S_j in P_0, and S_m and S_n in P_1).

The two processes may concurrently execute and they can communicate with each other by sending messages. The "end_parallel_commands" statement acts as a synchronization point. The next section illustrates how this program segment can be processed by the Parallel-C++ translator.

```
:
parallel_commands (2)
{
        process(P0)     {
                                :
                                Si;
                                Sj;
                                :
                        } ...
        process(P1)     {
                                :
                                Sm;
                                Sn;
                                :
                        } ...
}
end_parallel_commands;
:
```

Figure 4.4. Explicit Process Allocation Example

4.4.1 Automatic Processor Allocation (APA) Tree

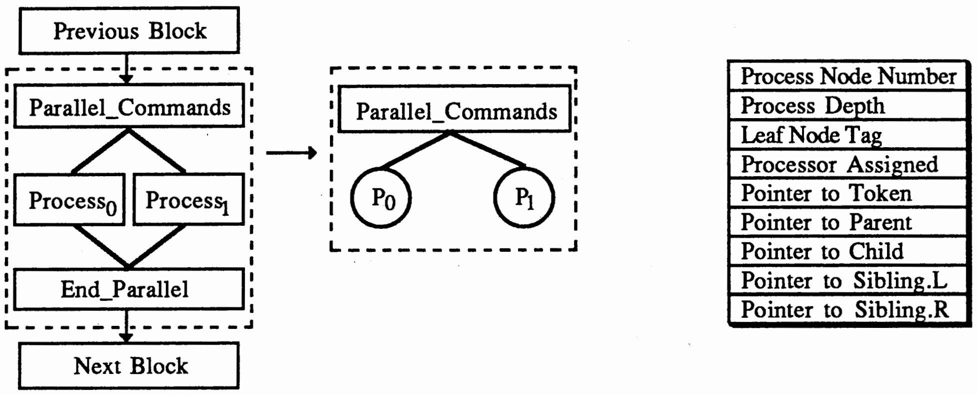
In order to allocate processors automatically to concurrent processes, the translator generates an *Automatic Processor Allocation tree* (APA tree) using the APA tree generation algorithm (Algorithm 1). If there are enough processors available on the target machine, each process in the explicit process allocation may be mapped using a one-to-one mapping into an individual physical processor.

```
Input:   a parallel_commands construct in a Parallel-C++ program.

Output:  an APA tree representing concurrent process nodes.

Method:  initially make a root to represent a parallel_commands construct;
         repeat while (not end_of_construct in a source)
         {
                 parse a source program to identify concurrent processes;
                 if (find concurrent_process)
                 {       push process_level into the process stack;
                         if (same process_level with current node)
                                 {       make a sibling node;        }
                         else if (lower process_level)
                                 {       parse parents node
                                                 to find a node of same process level;
                                         generate an appropriate node;               }
                         else if (higher process_level)
                                 {       generate a child node;      }
                 }
                 else if (find end_of_process)
                 {       pop process_level from the process stack;    }
                 keep current information in each token to use at the code generation step;
         }
```

Algorithm 1. APA Tree Generation

In an APA tree, each node represents a process in a "parallel_commands" construct. The Parallel-C++ translator generates an APA tree (as shown in Figure 4.5<a>) when the parallel_commands construct (in Figure 4.4) is parsed. Figure 4.5<b> shows the structure of a process node, P_i, in the APA tree.

In Figure 4.5<a>, two processes "P_0" and "P_1" are placed on two nodes. Those two nodes are leaf nodes which may execute in parallel. Each node may be assigned to a processor, and the necessary code to keep this information is generated and placed in the translated programs. For example, using the information kept in each process node, each concurrent process in a "parallel_commands" construct has been translated to codes like "if(mynode() == process_id) { ... }" using C++ code and iPSC/2 C routines (An example translation is shown later in Section 4.6.).



<a> A Simple APA Tree <b> A Process Node Structure

Figure 4.5. A Simple APA Tree and A Process Node Structure

As mentioned earlier, this example depicts a simple case. The "parallel_commands" construct can be more complicated, however. Such an example is considered in the next section.

4.4.2 Nested Structures

Parallel_commands constructs can be nested within another parallel_commands construct. Figure 4.6 shows a skeleton example of such a situation with a maximum nesting level of four.

```
parallel_commands(2)
{
        process(1a)                                              // level 1
        {       ...       Si;       ...                          // statement Si
                parallel_commands(2)
                {
                        process(2a)                              // level 2
                        {       ....       }
                        end_process;
                        process(2b)                              // level 2
                        {       ....       }
                        end_process;
                }
                end_parallel_commands;
        }
        end_process;

        process(1b)                                              // level 1
        {       :
                parallel_commands(1)
                {
                        process(2c)                              // level 2
                        {       :
                                parallel_commands(3)
                                {
                                        process(3a)              // level 3
                                        {       ....       }
                                        end_process;
                                        process(3b)              // level 3
                                        {       :
                                                parallel_commands(2)
                                                {
                                                        process(4a)     // level 4
                                                        {       ....    }
                                                        end_process;
                                                        process(4b)     // level 4
                                                        {       ....    }
                                                        end_process;
                                                }
                                                end_parallel_commands;
                                        }
                                        end_process;
                                        process(3c)              // level 3
                                        {       ....       }
                                        end_process;
                                }
                                end_parallel_commands;
                        }
                        end_process;
                }
                end_parallel_commands;
        }
        end_process;
}
end_parallel_commands;
:
```

Figure 4.6. Example Code with Nested Processes

In Figure 4.6, there are 10 nested processes in a program segment. Figure 4.7 shows the APA tree corresponding to the code segment. The "parallel_commands" construct has two processes in level 1, "1a" and "1b". Process "1a" in level 1 has two child processes, "2a" and "2b", and so on. The leaf nodes, "2a", "2b", "3a", "4a", "4b" and "3c", correspond to concurrent processes at the lowest level. Since they do not have any children, they can be executed in parallel. The number of leaf nodes represents the maximum level of parallel processing. To facilitate processor allocation, this tree is transformed into a binary APA tree form. Figure 4.8 shows the transformed binary APA tree. After transforming an APA tree into a binary APA tree, the APA routine allocates processors to process nodes in the tree using the process mapping algorithm (Algorithm 2).

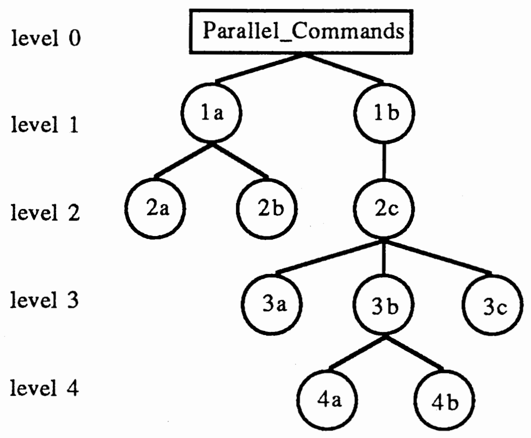


Figure 4.7. APA Tree with Nested Processes

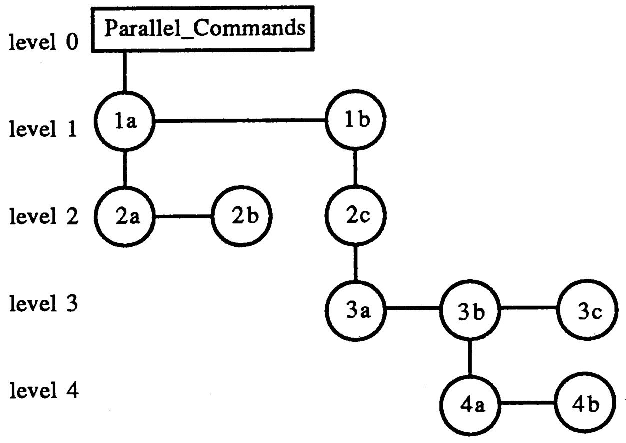


Figure 4.8. Transformed Binary APA Tree

```
Input:   an APA tree.

Output:  a transformed APA tree in which parallel processors are allocated to concurrent process
         nodes.

Method:  initially transform input APA tree to the binary APA tree;
         traverse the tree to find out concurrent leaf nodes;
         calculate the total number of leaf nodes (= necessary_processors);
         get the total number of available processors in the system (= available_processors);
         if (necessary_processors > available_processors)
                 call Process Node Boxing Algorithm (Algorithm 3);
         repeat traverse until (finished):
         {
                 if(leaf_node)
                 {
                         if (the node is boxed)
                                 assign the first processor to the leaf node;
                         else    assign the next available processor to the leaf node;
                 }
                 else    trace the processor identifiers assigned to its child nodes,
                         assign these processors to this internal node, and
                         keep this information for using at the code generation step;
         }
```

Algorithm 2. Process Mapping

The process mapping algorithm starts by checking leaf nodes in a depth-first search and determines the nodes that may run concurrently. In this particular example, the starred nodes ("*") in Figure 4.9 are those leaf nodes which may initially run concurrently at run-time. The APA routine assigns available processors to these leaf nodes. Once the processors are assigned to every leaf node, these numbers are traced from the leaf nodes to the parents. This processor assignment information is provided to the translator. The processor allocation is illustrated in Figure 4.9. In Figure 4.9, the number attached to the upper-right side of each leaf node represents the processor identifier assigned to each leaf node.

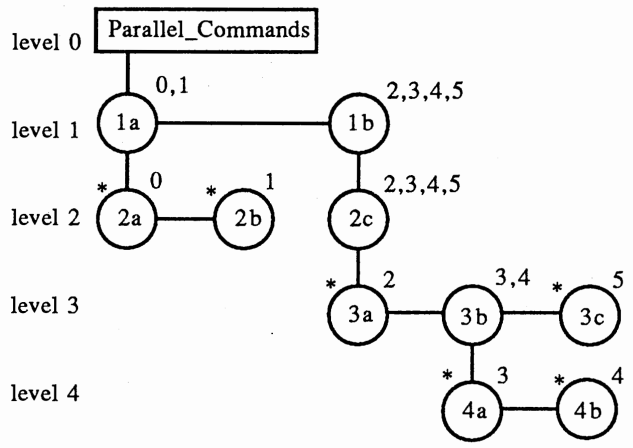


Figure 4.9. Process Allocation on the APA Tree

The Parallel-C++ translator uses the information given by the APA routine to generate appropriate parallel codes in the translated program. The parallel codes consist of a host program and a node program for the iPSC/2, and they ensure correct control flow within the "parallel_commands" construct. For example, the beginning parts of the parallel constructs given in the example code in Figure 4.6 are translated as follows:

```
"...
 if((mynode()==0) || (mynode()==1))                                    // process(1a)
        {
                :
                Si;                                                   // statement Si
                :
                if(mynode()==0)        {        ...        }          // process(2a)
                :
                if(mynode()==1)        {        ...        }          // process(2b)
                :
        }
... ".
```

4.4.3 Load Distribution with Limited Processors

In the preceding discussion, it was assumed that sufficient physical processors are available to be allocated to concurrent processes in "parallel_commands" constructs. However, this assumption is not possible except in the case of very small programs. Therefore, algorithms need to be developed to map processes to processors in a many-to-one fashion. Figure 4.10 depicts a process load distribution scheme with a limited number of processors using the preceding example of the previous section (Figure 4.6-4.9). It is assumed that the total number of available processors in the system is four. Using a depth-first search, the process node boxing algorithm (Algorithm 3) examines the first three leaf nodes until the number of processors (which is the number of assigned processors subtracted from the number of necessary processors) is equal to the total number of available processors. The boxed nodes are the first three leaf nodes to be checked in this particular example. The APA routine next assigns the first available processor to those nodes in a box, and assigns the remaining processors to the rest of the nodes (by using Algorithm 2). In this particular example, processor "0" is assigned to nodes "2a", "2b" and "3a", and processors "1", "2" and "3" are assigned to nodes "4a", "4b" and "3c", respectively. Those processors assigned to leaf nodes are traced to the upper-level parent nodes by the APA routine. The code generation routine uses this information later to generate the appropriate C++ codes and system calls on the target machine.

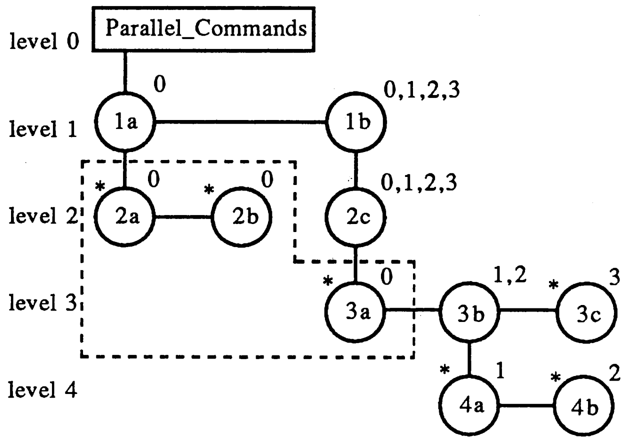


Figure 4.10. Load Distribution with Limited Processors

```
Input:   a transformed APA tree.

Output:  a transformed APA tree in which some concurrent process nodes are boxed.

Method:  necessary_processors = total number of leaf nodes;
         available_processors = total number of available processors;
         assigned_processors = 0;
         repeat traverse until (stop)
         {
           if (leaf_node)
           {
                assigned_processors = assigned_processors + 1;
                if ((necessary_processors - assigned_processors) >= (available_processors - 1))
                       {        this node is boxed;        }
                else stop;
           }
         }
```

Algorithm 3. Process Node Boxing

So far, in this chapter, the translation scheme of the "parallel_commands" construct has been described. Object migration is another important concept in Parallel-C++. The next section outlines this concept and the associated translation scheme.

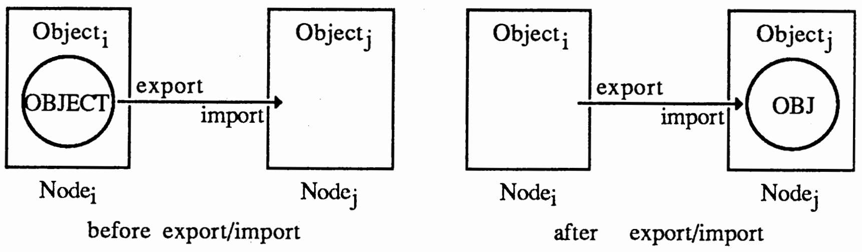
4.5 Object Migration

Parallel-C++ provides facilities to export and to import dynamic objects between static objects (or processes) in the system. This concept of object movement, in which objects travel from one object (or process) to another, is called *object migration*. As explained in the previous chapters, there are two kinds of objects in Parallel-C++: (1) *dynamic objects* - objects that migrate, and (2) *static objects* - stationary objects (or processes) that do not migrate. Using "*export/import*" constructs, dynamic objects can travel around among static objects (or processes). Such a static object is called the *owner* of the corresponding dynamic object. The corresponding language construct is explained in Section 3.8 and is also shown in Figure 4.11.

```
export (object-id, destination-object-id);

import-object-variable        =      import (class-id);
                              |      import (class-id, source-object-id);
```

Figure 4.11. Object Migration Constructs: Export/Import

We recall that the function of the "export" construct is to export a dynamic object to other objects (processes), and the function of the "import" construct is to import a dynamic object of the given class. The exported object is placed in the import-list of the destination object (process). The imported object is bound to the "import-object-variable". The process that issued the "export" statement continues its execution even if no object (process) imports such an exported object. When the importing object (process) issues the "import" statement, however, if no object (process) exports such an object, then the importing object (process) must postpone execution until the required object arrives at the import-list. These "export/import" constructs support object migration, which can represent practical real-world problems such as air-traffic control systems, dynamic load balancing problems [Cybenko 89], network simulations [Jo et al. 89], dynamic file migration [Gavish and Sheng 90], and real-time robot simulation [Cox and Gehani 87] [Cox 88] [Cox et al. 88], etc. in a natural way.



Source	Translation
export(OBJECT, process(OBJECT_J));	send(OBJECT) to Object$_j$; delete(OBJECT) from Object$_i$;

OBJ = import(CLASS);	OBJ = new CLASS;
	receive(OBJ);

Figure 4.12. Object Migration Translation Scheme

Export/import constructs in Parallel-C++ are translated using the scheme shown in Figure 4.12. The translated C++ code with iPSC/2 C routines for the example segment of code are shown in Figure 4.13. An importing object (or process) instantiates a new object of the same type to import a dynamic object from an exporting object (or process). The state of the new object is replaced by the state of the imported object. The exporting object (or process) sends the current state of the object by sending the contents of the memory region from the current object (or process) to the object region in the importing object (or process). There may be a possible waiting time for the importing object. Once the message containing the exported object arrives at the buffer of the importing object (or process), the exporting object (or process) continues execution of the statements following the "export" statement.

```
Source:         export (OBJECT, process(OBJECT_J));

Translation:    csend(PC_MSG_EXPORT, &OBJECT, sizeof(OBJECT), OBJECT_J,
                                                       PC_APPL_PID);

Source:         OBJ = import (CLASS);

Translation:    CLASS         OBJ;
                crecv(PC_MSG_IMPORT, &OBJ, sizeof(OBJ));
```

Figure 4.13. Translated Export/Import Constructs using C++ and iPSC/2 C Routines

4.6 Example

Figure 4.14 is an example of a simplified version of a practical program illustrating object migration. It combines the constructs discussed earlier, and provides a comprehensive example. The program simulates an air-traffic-control system. There are three airports, "OK-City", "Phoenix" and "Dallas", and three airplanes, "American", "United" and "Western", controlled by those airports, respectively. There are three concurrent processes, "Oklahoma", "Arizona" and "Texas", in which those airports reside. It is assumed that an airplane can be controlled by only one airport at a given time. Once control is transferred over to the destination airport by export/import, the original airport cannot directly control that airplane anymore. The state and information are hidden in an airplane object, and no other airport object can access them. When an airplane object is exported, the airport object importing such an airplane object takes over the state and information of the imported airplane. This program is translated (as shown in Figure 4.15) and executed by using the translation scheme outlined in this chapter. Figure 4.16 is the run-time output of this program.

```
// Parallel-C++ EXPORT/IMPORT Test Program.

#include <stream.h>
char* obj(int);
class plane {    int Name;   int Destine;
      public:
        plane(int n, int d)        {    Name = n;  Destine = d;
                                             cout<<"Airplane "<<obj(Name)<<" instantiated.\n"; }
        plane()                    {    cout<<"An airplane arrived.\n"; }
        void take_off(int Port)    {    printf("%s is taking-off from %s.\n",obj(Name),obj(Port)); }
        void landing(int Port)     {    printf("%s is landing at %s(%s).\n",
                                                         obj(Name),obj(Port),obj(Destine)); }
};

class airport   {    int Port;   int Wing;  int Destine;
      public:        airport(int a, int p, int d);
};

airport::airport(int a, int p, int d)  {    Port = a; Wing = p; Destine = d;
                                            cout << "Airport "<<obj(Port)<<" instantiated.\n";
                                            plane Flight(Wing, Destine);
                                            printf("%s controls %s.\n",obj(Port),obj(Wing));
                                            Flight.take_off(Port);
                                            export(Flight, process(Destine));
                                            x = import(plane);
                                            x.landing(Port);
}

char* obj(int i)   {       switch(i)  {    case  0 : return("OKLAHOMA");
                                           case  1 : return("ARIZONA");
                                           case  2 : return("TEXAS");
                                           case  3 : return("OK_CITY");
                                           case  4 : return("PHOENIX");
                                           case  5 : return("DALLAS");
                                           case  6 : return("AMERICAN");
                                           case  7 : return("UNITED");
                                           case  8 : return("WESTERN");
                                           default : return("NONE");
                                      }
}

main() {   int OKLAHOMA=0;     int ARIZONA=1;   int TEXAS  =2;
           int OK_CITY =3;     int PHOENIX=4;   int DALLAS =5;
           int AMERICAN=6;     int UNITED =7;   int WESTERN=8;
           parallel_commands(3)
           {        process(OKLAHOMA)  {  airport OK_City(OK_CITY, AMERICAN, ARIZONA);
                           } end_process;
                    process(ARIZONA)   {  airport Phoenix(PHOENIX,  UNITED,  TEXAS);
                           } end_process;
                    process(TEXAS)     {  airport Dallas(DALLAS,  WESTERN, OKLAHOMA);
                           } end_process;
           } end_parallel_commands;
}
```

Figure 4.14. An Example using Object Migration between Distributed Objects

```
/* <host.c>  Parallel-C++ (c) 1990 */

#include "pcpp.h"
main() {
        getcube("Parallel-C++","4","default",0);
        setpid(PC_HOST_PID);
        load("node",PC_ALL_NODES,PC_APPL_PID);
        message_class msg;
        csend(PC_MSG_TYPE_I, &msg, sizeof(msg), PC_ALL_NODES ,PC_APPL_PID);
        csend(PC_MSG_PAR_CO, &msg, sizeof(msg), PC_ALL_NODES, PC_APPL_PID);
        { // parallel_commands
            { // process    } // end_process
            :
        } // end_par_com
        crecv(PC_MSG_TYPE_L, &msg, sizeof(msg));
        killcube(PC_ALL_NODES, PC_ALL_PIDS);
        relcube("Parallel-C++");
} // end_of_host_main


/* <node.c>   Parallel-C++ (c) 1990 */

#include "pcpp.h"
main() {
          my_node = mynode();
          my_pid  = mypid();
          message_class  msg;
          crecv(PC_MSG_TYPE_I, &msg, sizeof(msg));
          if(my_node < NO_OF_NODES)  {
              int OKLAHOMA=0; int ARIZONA=1; int TEXAS  =2;
              int OK_CITY =3; int PHOENIX=4; int DALLAS =5;
              int AMERICAN=6; int UNITED =7; int WESTERN=8;
              crecv(PC_MSG_PAR_CO, &msg, sizeof(msg));
              { // parallel_commands
                  if((my_node==0))
                  { // process
                                        airport OK_City(OK_CITY, AMERICAN, ARIZONA);
                  } // end_process
                  if((my_node==1))
                  { // process
                                        airport Phoenix(PHOENIX, UNITED,  TEXAS);
                  } // end_process
                  if((my_node==2))
                  { // process
                                        airport Dallas(DALLAS,  WESTERN,  OKLAHOMA);
                  } // end_process
              } // end_par_com
          }
          if(my_node==0) { waitall(1,PC_ALL_PIDS); }
          if(my_node==0) {
                  csend(PC_MSG_TYPE_L,&msg,sizeof(msg),myhost(),PC_HOST_PID);}
} // end of node_main
```

Figure 4.15. Translated Program using C++ and iPSC/2 C Routines

```
/* <pcpp.h>   Parallel-C++ (c) 1990 */

#include <stdio.h>
#include <stream.h>
#include "msg.h"

char* obj(int);

class plane {   :

};

class airport {     int Port;   int Wing;     int Destine;
        public:
                        airport(int a, int p, int d);
};

airport::airport(int a, int p, int d)
{
            Port = a; Wing = p; Destine = d;
            cout << "Airport "<<obj(Port)<<" instantiated.\n";
            plane Flight(Wing, Destine);
            printf("%s controls %s.\n",obj(Port),obj(Wing));
            Flight.take_off(Port);
            csend(PC_MSG_EXPORT,&Flight,sizeof(Flight),Destine,PC_APPL_PID);
            plane x ;
            crecv(PC_MSG_IMPORT,&    x ,sizeof(          x ));
            x.landing(Port);
}

char* obj(int i)
{  :
}


/* "msg.h"   Parallel-C++ (c) 1990  */

//  msg type for Parallel-C++ programs
#define NO_OF_NODES     32       ...
class  message_class {      /* misc. class def. */
                                   :
};

// "fork.h"
#include <errno.h>
:
```

Figure 4.15. (Continued)

```
// Node(ID)                                           // Node(i) allocated

(0)          Airport OK_CITY instantiated.            // for Process(OKLAHOMA)
(2)          Airport DALLAS instantiated.             // for Process(TEXAS)
(1)          Airport PHOENIX instantiated.            // for Process(ARIZONA)

(0)          Airplane AMERICAN instantiated.
(2)          Airplane WESTERN instantiated.
(1)          Airplane UNITED instantiated.

(0)          OK_CITY controls AMERICAN.
(2)          DALLAS controls WESTERN.
(1)          PHOENIX controls UNITED.

(0)          AMERICAN is taking-off from OK_CITY.
(2)          WESTERN is taking-off from DALLAS.
(1)          UNITED is taking-off from PHOENIX.
                                                      // export/import occurred
(0)          An airplane arrived.
(2)          An airplane arrived.
(1)          An airplane arrived.

(0)          WESTERN is landing at OK_CITY(OKLAHOMA).
(2)          UNITED is landing at DALLAS(TEXAS).
(1)          AMERICAN is landing at PHOENIX(ARIZONA).
```

Figure 4.16. Output of the Example Program

4.7 Discussion of Translator Implementation

Besides the APA tree technique, "fork/join" constructs have been used experimentally to translate parallel constructs. For each concurrent process, a pair of "fork/join" constructs is generated and executed. The forked segments of code execute concurrently using child processes generated from a parent process at run-time. This scheme is useful to parallelize concurrent processes for a single processor system, or for a limited number of processors on the multiprocessor system. In a single processor system, the "fork/join" concurrency can be used to utilize concurrent processes. In a distributed processor system, for a maximum speed-up, the upper level processes in an APA tree may be assigned to parallel processors that are somewhat limited, and then "fork/join" parallelization can be used for the rest of the processes in the lower level.

Besides plainly generating a "fork/join" code for each process, automatic program partitioning using computation weights can be assigned to parallel processes. Researchers such as Sarkar [Sarkar 90] have investigated the "fork/join" parallelizing problems by instruction reordering. Another remaining problem of translation for explicit process allocation is load balancing among the involved processes.

The parallelizing translator of Parallel-C++ is relatively easy to implement by using existing facilities, because the translator can be built on the top of the C++ compiler and operating system in the target machine. Compatibility with existing C++ is not affected.

The current implementation relies much on static information, such as the number of available processors in the system, given at translation time. The translation using dynamic information given at run-time is more useful in real programming.

Translation techniques are not always good for fine-grained parallelism. Much time is spent on communication and on checking dependencies to satisfy data synchronization for concurrent execution of partially parallelized portions of codes. Programming using medium-grained or large-grained parallelism, such as parallel modules, parallel function calls and parallel objects, should be used to achieve concurrency with run-time data integrity.

For the structures of a translated program, linked-lists are used for each token or a line of code. Linked-lists, consisting of several objects or structures, make translation easier for dynamic insertion, deletion and appending statements than any other data structure does. Using a parser (instead of iteration structures of the main translation part, which is currently used), more sophisticated manipulation of translation for the nested program can be expected.

Figure 4.17 includes iPSC/2 C routines which have been used in the translation. Using the information kept in concurrent sections of codes, appropriate iPSC/2 C routines are generated in the translated program.

```
void    attachcube(char *cubename);         // attach to a cube and make it the current cube

void    crecv(long typesel, char *buf, long len);
                                            // send a message and wait for completion

void    csend(long type, char *buf, long len, long node, long pid);
                                            // send a message and wait for completion

void    getcube(char *cubename, char *cubetype, char *srmname, long keep);
                                            // allocate a cube

void    killcube(long node, long pid);      // terminate and clear out a process(es)

void    load(char *filename, long node, long pid);
                                            // load a node process

unsigned long    mclock();                  // return the time

long    myhost();                           // obtain the node id of the host machine

long    mynode();                           // obtain the node id of the calling process

long    mypid();                            // obtain process id of the calling process

long    numnodes();                         // obtain the number of nodes in the cube

void    relcube(char *cubename);            // release a cube

void    setpid(long pid);                   // sets process id of a host process

void    waitall(long node, long pid);       // wait for all the specified processes to complete
```

Figure 4.17. iPSC/2 C Routines used in Translation

Based on the experience of parallelizing translation, we have investigated basic implementation schemes of a parallelizing compilation for this particular language and system. In the next chapter, we describe the implementation schemes for the Parallel-C++ compiler-interpreter in general.

CHAPTER V

IMPLEMENTATION SCHEME II (COMPILER-INTERPRETER)

This chapter describes the design of an implementation scheme for a Parallel-C++ compiler-interpreter targeted to an abstract machine which is assumed to be a multiprocessing computer like the iPSC/2 hypercube system. One of the objectives of this project is to study the issues that arise in the compilation of distributed programming languages. In this work, attention is focused on the storage management scheme.

5.1 Overall Structure

The Parallel-C++ compiler-interpreter implementation consists of two phases, compilation and interpretation. These two phases are illustrated in Figure 5.1.

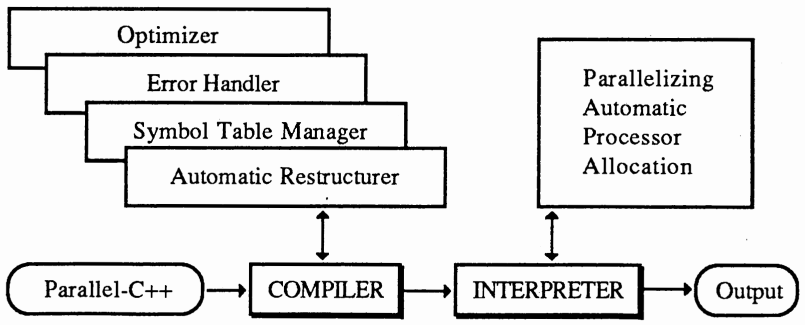
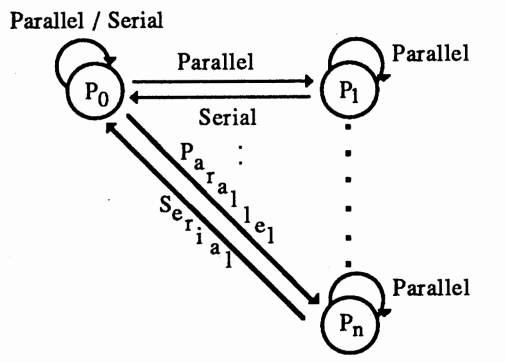


Figure 5.1. Parallel-C++ Compiler-Interpreter Structure

The Parallel-C++ compilation is further subdivided into four phases, lexical analysis, syntax analysis, code generation and automatic restructuring. The components, which are part of the compilation, consists of lexical analyzer, syntax analyzer, code generator, automatic restructurer, symbol table manager, error handler and optimizer. The lexical analyzer reads a Parallel-C++ source program and produces a sequence of tokens, while constructing a symbol table by calling the symbol table manager. If the lexical analyzer encounters lexical errors, then the error handler is invoked. The specification of the parser is provided by the Parallel-C++ grammar. Intermediate code is generated while parsing by calling the code generator. The code generator produces intermediate code based on the semantic rules defined by the grammar. The symbol table manager also helps at this stage to handle temporary variables which can be generated while parsing. The syntax analyzer also performs the task of error handling such as identifying, locating, and reporting syntactic errors, by calling the error handler. When performing a compilation, the compiler performs automatic restructuring. The automatic restructurer restructures some parts of the intermediate code to be parallelized by using the information explicitly given in the program. For example, with a user-defined parallel construct "autobegin/autoend", the automatic restructuring routine checks interdependencies of the statements in the construct. This routine divides the statements into independently and concurrently executable blocks of statements. Each block consists of instructions that must be executed sequentially. The automatic restructurer constructs a virtual execution graph to check the interdependency of the segments, and congregates the dependent statements into blocks at each dependency-checking step. The intermediate code generated by the compiler is targeted to a MIMD type abstract machine.

The interpreter executes the target program (which is here the intermediate code generated by the compiler). The tasks of "parallel execution" and "automatic processor allocation" are done by the interpreter. The concurrent blocks of statements, which are automatically parallelized and restructured at the compilation step, can be mapped into respective processors assigned by the automatic processor allocation routine. Such parallelized statements can be concurrently executed by the interpreter at run-time. The interpreter mainly executes the serial program segments on the main processor, and activates the necessary number of processors when it meets parallel sections. To do this, the interpreter uses a flag to denote one of the two possible modes, serial and parallel. At a given time during interpretation, the interpreter can be in one of two possible modes. In serial mode, the interpreter executes an object code segment one instruction at a time. When the interpreter meets a parallel construct, it switches to parallel mode. The actions associated with mode switching include run-time actions such as automatic processor allocation. An initial message passing may occur when it is needed (e.g., to transfer initial data). When the mode switching is complete, all of the processors allocated for the parallel constructs concurrently execute their portions of the program. After parallel execution, when the interpreter meets the end of the parallel construct (the join point), the execution mode switches to the serial mode. At this point, all concurrent processes must have terminated and control is transferred to the main processor. The main processor takes care of execution of the serial part of code until it meets another parallel construct again. The transition diagram of this control flow is shown in Figure 5.2. In the present scheme, the parallel constructs at higher level are handled along with the lower level.



P_i : Processors, i = 0, ..., n (No. of Processors)
Execution modes: Serial / Parallel

Figure 5.2. Control Flow Transition Diagram

Current implementation of the compiler-interpreter is targeted to the iPSC/2, and consists of several programs, implemented in C++. Typical programs among them are "host.c", "node.c", "global.h", "lex.yy.c" and "y.tab.c". The programs "lex.yy.c" and "y.tab.c" are the lexical analyzer and syntax analyzer, respectively. Those two programs are generated by using Unix system tools, LEX [Lesk and Schmidt 75] and YACC [Johnson 78] [Sullivan 86]. The syntax analyzer, "y.tab.c", indirectly performs target code generation, code restructuring, code optimization, symbol table management, and error/information message generation by calling necessary modules such as the code generator and the lexical analyzer. The lexical analyzer, "lex.yy.c", includes all lexical definitions and symbol-table management functions. The program "global.h" includes all global function definitions, variable declarations, class definitions, other general definitions and sub-programs, such as code generator, message generation functions, and other miscellaneous functions. The three programs, "lex.yy.c", "y.tab.c" and "global.h", are included in the main program, "host.c", and the interpreter, "node.c". They are compiled and linked together to make the executable code, "host" and "node". The main program "host.c" calls the syntax analyzer module, from which the lexical analyzer and code generator are invoked. After finishing code generation, "host.c" sends initial messages to the interpreter, "node.c", which executes the target code generated by the compiler. The main program, "host.c", is executed on the host processor, and copies of the interpreter, "node.c", are executed on node processors. The interpreter executes the serial part of the target code on the main processor, and the parallel part of the code on the parallel processors, designated by the automatic processor allocator while switching parallel/serial mode.

5.2 Abstract Machine Model

Since we are interested in compilers for a distributed memory machine, an abstract machine model based on Intel iPSC/2 is used. Such an abstract machine is hypothetically a multiprocessor system with local memory, and is capable of parallel processing. A schematic view of an abstract machine model is shown in Figure 5.3. The abstract machine is assumed to have a general-purpose register architecture, and consists of a host processor and several node processors. The interconnection network is assumed to be the same as the iPSC/2 [Intel 88]. The operating system is assumed to be a Unix-like operating system that is capable of file management, input/output management, system supervisor, device interfaces, interprocess communication, central processing unit (CPU) scheduling, process management, multi-tasking management, user interface, memory management, communication network control, and etc. The host and the nodes are inter-connected by a communication network. The host processor has a system control unit, a network control unit, an input/output device control unit, and input/output devices. A node is an independent processor which is equipped with a CPU, a local memory, and network/bus interface units.

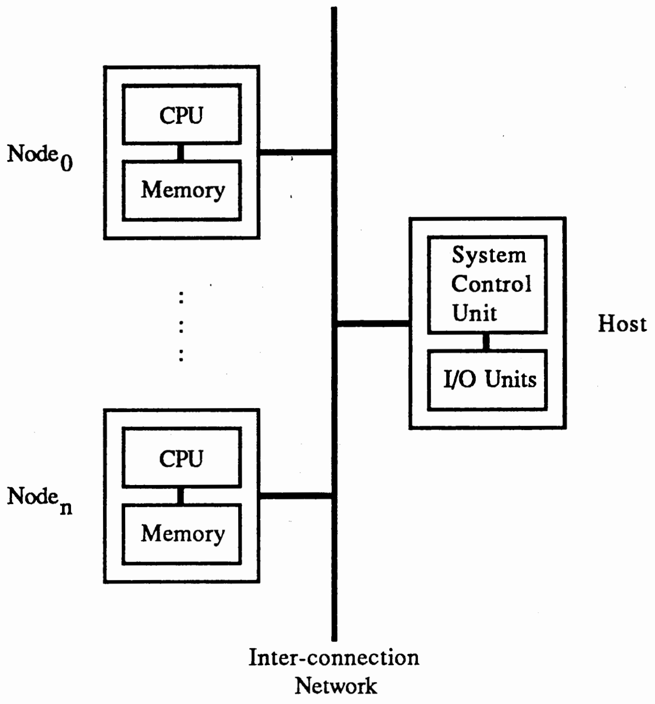


Figure 5.3. Abstract Machine Model Architecture

5.3 Instruction Set for Conceptual Architecture

The intermediate code is assumed to be the instruction set of the virtual machine. The basic form of an instruction is a quadruple [Aho et al. 86]. This intermediate code is machine-independent, and can be subsequently translated to the machine code for the target machine. (The current implementation allows the interpreter to execute the intermediate code directly, which is generated by the compiler, on the iPSC/2.) Each instruction has an operation and three operands. The format of a quadruple is "(operation-code, operand-1, operand-2, operand-3)". For example the quadruple "(+, y, z, x)" has the same meaning as the operation "x <- y + z". The instruction for the general-purpose register architecture has explicit operands, either registers or memory locations. Therefore, the variable names shown in the example quadruple, such as "x", "y", and "z", are supposed to appear, in the actual code, as symbolic addresses of those variables to be located in memory, or as registers to be allocated to those variables. Storage for objects is allocated in the heap area of the memory. The code for object communication includes identifiers of the objects. At run-time, those object identifiers are converted into actual addresses of the storages for the objects. For example, an export statement is translated to the code "(operation-export, exported-object, object/process flag, destination-object)". The "operation-export" tells that this operation is for exporting an object. The "exported-object" operand is the identifier of the object to be exported. The "object/process flag" tells the destination of this export operation, either object or process. The "destination-object" is the identifier of the destination object (or process) for this export operation. At run-time, when an object is to be passed to another processor (for example, object export occurred), the address and length of the storage for the object are calculated. Then the stream of bytes is passed by value to the destination object in a distributed memory model (current implementation model). However, the address of the object (or the object identifier itself) can be passed by reference to the destination in a shared memory model.

5.4 Run-Time Storage Management

The cost associated to parallel processing is primarily in run-time storage management for a distributed memory system. In this section we describe the run-time storage management for Parallel-C++.

5.4.1 Distributed Storage Management

Run-time data structures include class-lists, object-lists, import-lists, object frames, stacks and heaps. All processors keep their own run-time data structures in their local memory. At run-time the general data storage management for the program is done using the stack model [Gries 71]. Each processor has its own stack in its local memory. In our scheme for the distributed memory system, when the parallel processors are invoked, the main processor distributes copies of the current environment to all of the parallel processors involved. Each parallel processor keeps its own activation records until the task allocated to it is completed. When the interpretation switches to the serial mode from the parallel mode, the area occupied by the current environment is freed.

The current values of all of the variables in the activation records and the symbol-table in the main processor are sent to the local memories of the processors involved when the processing mode switches to the parallel mode. The symbol table serves as a message to transfer current values of variables between processors. When the main processor sends this information, it also sends the current system time. Distributed processors involved in parallel computing receive a message of the symbol-table with current system time. After concurrently performing their portion of the calculation as defined in the parallelized target code, those distributed processors update time-tags of the variables involved in the calculations and send a message of the updated symbol-table with such information. After receiving such messages from the parallel processors, by using the time-tag information, the main processor updates the values of the variables which had been updated by the previous distributed computing. The interpreter manipulates all of those operations which have been described here, and controls all involved processors, using the information specified in the target code which has been parallelized and restructured by the compiler. This time-tag scheme brings in another communication overhead to the system. However, it is one of the solutions to solve data integrity problems occurring in distributed memory systems.

Besides this kind of complexity involved in data updating, there is also communication overhead through message-passing. This kind of communication overhead causes a major burden, reducing the efficiency of computing in a distributed memory system. This scheme, due to the characteristics of a distributed memory computer, limits the use of shared variables among the parallel code segments and the use of non-local variables in the parallel function calls. Because the use of shared variables needs lock-step synchronization and message passing, the communication overhead for the frequent use of shared variables may adversely affect the efficiency of distributed computing. Therefore in our system, if necessary, access to shared variables is allowed only by using a synchronization construct. It helps a parallelizing compiler to detect easily the accessing of variables in different address spaces, and to produce communication code for it efficiently. In a shared memory system, much of this kind of burden can be eliminated by passing (or using) pointers (or addresses) to the distributed objects and variables in a global virtual address space.

5.4.2 Distributed Object Management

For the distributed object management, the heap model is used [Goldberg and Robson 83]. The traditional stack model is not adequate for the dynamic data storage management for the objects because the lifetime of an object is not fully scoped within a function or an object. All of the classes are grouped into the class-list. Each processor keeps a copy of class-list in its local memory. An object instantiated from a class is represented by a contiguous region of memory, like a C++ object [Stroustrup 89b]. Each object has its own data storage to keep its current state of member data. All of the objects instantiated from each class are placed in the object-list. Each processor keeps its own object-list to record all objects which are either instantiated in the processor or imported from other processors. Each object has an object frame, stored in the heap storage, to keep the necessary information for the object. Each entry in the object-list has a pointer to its object frame in the heap, and also has a pointer to its class in the class-list. An object frame includes an object identifier, a pointer to its class in the class-list, an integer variable to keep its object frame size, a static/dynamic object tag, a pointer to its owner (for the dynamic object frame) or a pointer to the dynamic object-list (for the static object frame), a pointer to the imported object-list (for the static object frame), data members (An object of derived class also includes/concatenates data members defined in the base class.), a pointer to the member function area, and a tag of access level - private or public (Figure 5.4<a>). Each object can use this information to perform its operations properly.

Object Identifier
Pointer to Class
Object Frame Size
Static/Dynamic Tag
Pointer to Owner/Dynamic Objects
Pointer to Imported-Objects
Data Members
Pointer to Member Functions
Access Level

<a> Object Frame

Source	Destination	Object Frame

<b> Dynamic Object Message Format

Figure 5.4. The Structure of Object Frame and Dynamic Object Message Format

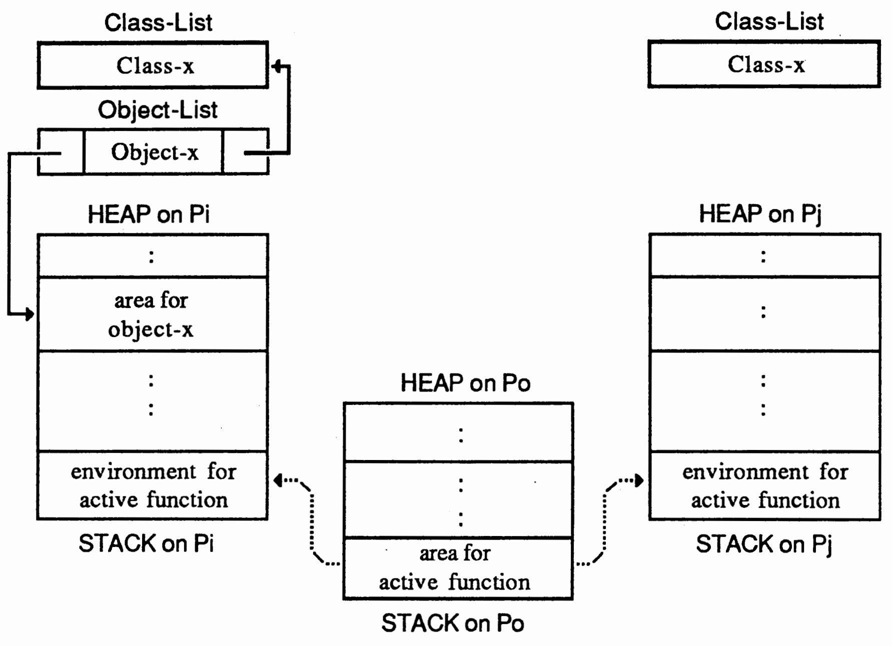
5.4.3 Dynamic Object Management

In Parallel-C++, dynamic objects can be exported and imported among static objects (or processes). We remind the reader that there are two kinds of environments for a dynamic object: (1) the *internal* environment is made up of the data structures and methods of an object; and (2) the *external* environment is the environment enclosing the dynamic object. The internal environment of the dynamic object moves with it, but the external environment does not. The internal environment of an exported object is bound to the external environment which is provided by an importing object. We call this *Dynamic Interface Environment Binding* (DIEB). By this DIEB, an object behaves differently according to the environment in which it resides.

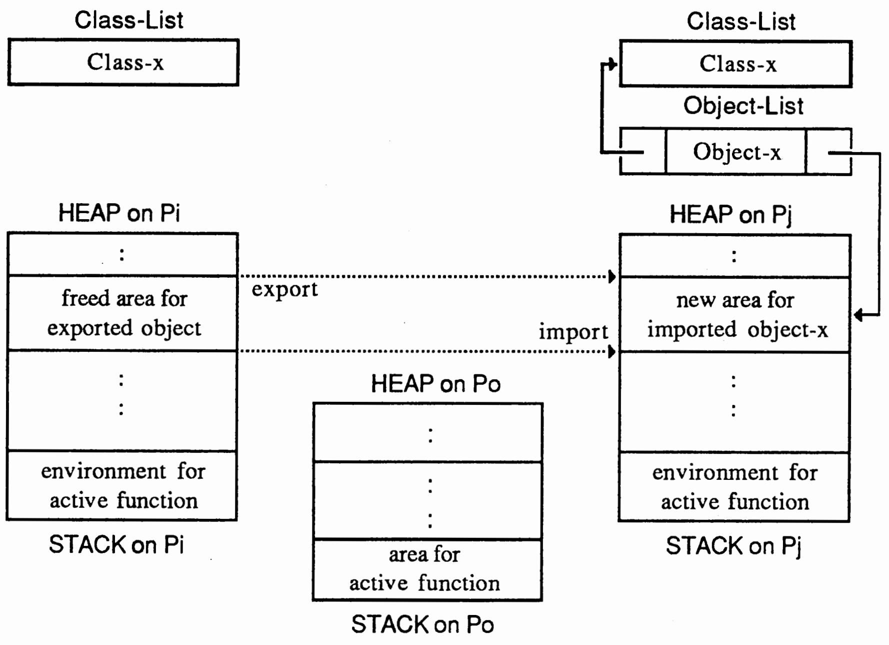
When an object (or process) exports a dynamic object, it also exports the internal environment of the dynamic object by sending a dynamic object message (Figure 5.4<b>) to the destination object (or process). This affects heap storage as follows; (1) the internal environment of the dynamic object is deleted from the exporting object, (2) the internal environment for the imported object is established in the importing object, (3) the heap segment for the exported object is freed from the exporting object (copy-and-delete semantics). Also, when an object meets a destructor, the area occupied by the object is freed (by following the conventional scope rule).

Establishing the new environment consists of several steps. The instantiated (or imported) object is included in the object-list. The pointer to its class in the class-list is modified. In a distributed memory environment, the physical address of the dynamic object in the importing object (or process) becomes different from the previous address in the exporting object (or process). At the same time its object frame is allocated in the heap, and the necessary pointers and values are updated (e.g., updating the value of pointer to the owner of the imported dynamic object).

Figure 5.5<a> and 5.5<b> show snapshots of the run-time storage for distributed and dynamic objects on parallel processors. In this example figure, we have three processors, "P_0", "P_i" and "P_j". Especially "P_0" is a main processor, and "P_i" and "P_j" are distributed processors which are executable concurrently. We assume that each processor has its own local storage. The object "object-x" is instantiated from the class "class-x" and its environment is established in the processor "P_i". After "object-x" is imported by the processor "P_j", the new environment for this imported object is established in the processor "P_j" and the old environment for the exported object is freed from the processor "P_i". The run-time storage, stacks and heaps for dynamic objects, are managed by the respective processors in which those objects reside at that time. The stack and heap areas are assumed to be at the opposite ends of a portion of memory.



<a> Run-Time Storage before Export/Import



<b> Run-Time Storage after Export/Import

Figure 5.5. Run-Time Storage with Dynamic Objects

5.5 Example

As a practical example to illustrate the run-time storage management scheme, especially for the management of distributed and dynamic objects, we consider the following scenario. Several airplane objects are concurrently controlled by several airport objects. Only one airport object can control a plane object at a time. This situation occurs often in several distributed systems. A skeletal implementation code of the scenario is shown in Figure 5.6. This example code is developed using the "parallel_commands" construct and the "export/import" constructs explained in the previous chapter. In this example, the main process allocates two concurrent processes, "Oklahoma" and "Arizona". These processes can be mapped to parallel processors and can be concurrently executed at run-time. In these processes, the airport objects, "OK-City" and "Phoenix", are instantiated from the airport class. We assume that airport "OK-City" sends a plane "American" to the airport "Phoenix". In this example, the airport objects are static objects and the plane objects are dynamic objects.

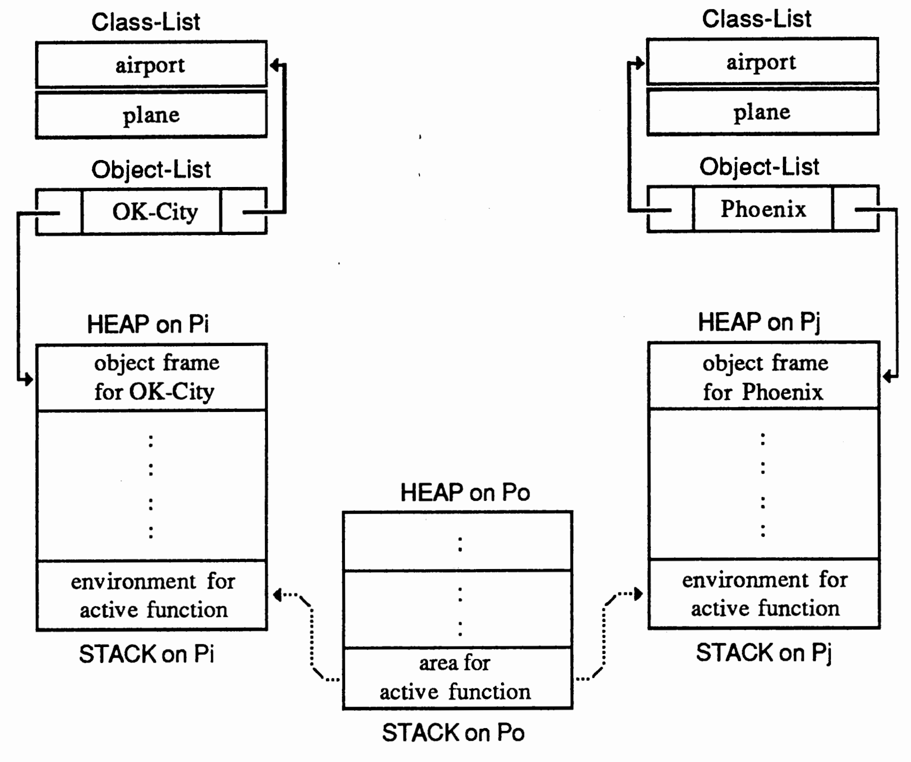
The objects distributed in different processors can have the same name. But those objects are physically different and have their own data structures and methods. The "importing-object-variable" (for example, "Flight" used in the "import" construct in Figure 5.6) can be used as an anonymous object instantiation for the given class (for example "plane" in the "import" construct in Figure 5.6). The given class helps a generic instantiation of its object. The initial status of the imported object "Flight" is essentially the same as the exported object "American". The run-time interpreter takes care of this kind of initialization details according to the state of the imported object.

```
/* Example Scenario:   Airport "OK_City" (on "Pi", "OKLAHOMA") sends a plane
                                  "American" to "Phoenix" (on "Pj", "ARIZONA").  */

class plane {    int  enviro;                                                    // For line(i)
                 :                                                               // refer to
                 public:                                                         // Figure 5.7<i>
                        void takeoff()   { ... enviro = owner →weather; ... }
                        void landing()   { ... enviro = owner →weather; ... }
};

class airport {  int  weather;
                 :
                 public:
                 control_takeoff(plane Flight) {  Flight.takeoff();
                                                  export(Flight, Airport_id); }   // line(c)
                                                                 // Airport_id = {Phoenix, ...}

                 control_landing(plane Flight)  {  Flight = import(plane);        // line(c')
                                                   Flight.landing(); }
                                                                 // Flight = {American, ... }
};

main() {  :
            parallel_commands(No.of.Airports)
          {
                 process(OKLAHOMA)
                 {         :
                        airport OK_City;                                         // line(a)
                        for(;;)
                        {  switch(real.time.input)
                           {
                                  case(create):   plane American;               // line(b)
                                  case(takeoff):  OK_City.control_takeoff(American);
                                  case(landing):  OK_City.control_landing(...);
                           }
                        }
                 } ...
                 process(ARIZONA)
                 {         :
                        airport Phoenix;                                         // line(a')
                        for(;;)
                        {  switch(real.time.input)
                           {
                                  case(create):   plane ...;                    // line(b')
                                  case(takeoff):  Phoenix.control_takeoff(...);
                                  case(landing):  Phoenix.control_landing(Airplane_id);
                           }
                        }
          :      } ...
          }
           end_parallel_commands;                                                // line(d)
}
```

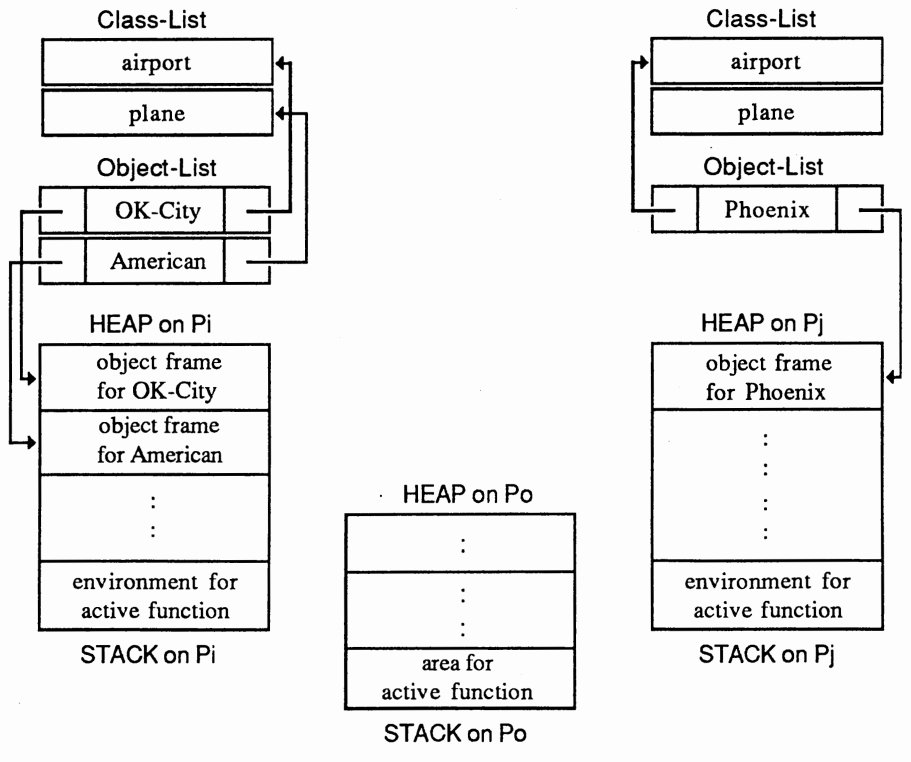
Figure 5.6. Example of Dynamic Object Migration between Static Objects

Consistent with the characteristics of the target machine - distributed memory, access to global variables and the usage of the pointer variable in the member functions of dynamic objects are not allowed. Necessary information should be passed by value. When no free area can be allocated to the object due to memory fragmentation, heap compaction is done by moving all of the object frames to the lower-end of the heap area, and the related object pointers should be updated. While some languages like C++ need explicit deallocation of objects, other object-oriented systems, such as Smalltalk or Lisp-based systems, use garbage collection to deallocate objects [Atkins and Nackman 88]. Actually the effects of the "export/import" constructs play the role of implicit deallocation and allocation of dynamic objects.

As an example, Figure 5.7 (a, b, c and d) provides snap-shots of the run-time memory organization for the execution of the dynamic object migration example in Figure 5.6. (Figure 5.7<i> matches the line number "i" in Figure 5.6.) In this particular example, we have three processors, "P_0", "P_i" and "P_j". The main processor "P_0" allocates two parallel processors "P_i" and "P_j". A process in the example program is mapped into a processor in the figure. The airport object "OK-City" (on "P_i") sends a plane object "American/Flight" to the airport object "Phoenix" (on "P_j"). This example shows the run-time storage management for Parallel-C++ with emphasis on object level.

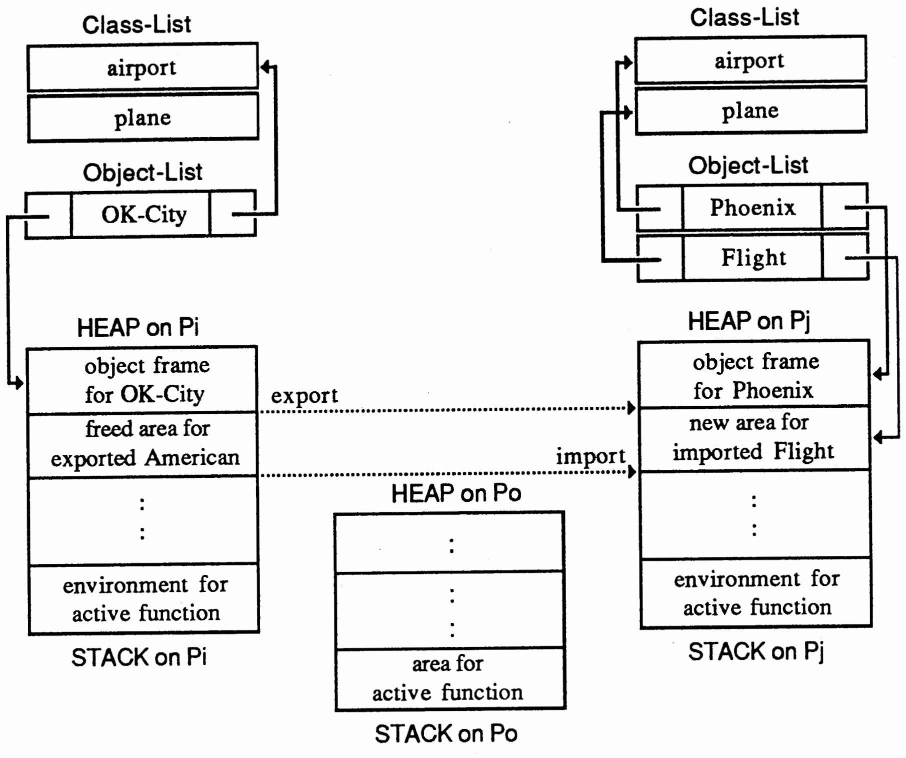


<a>

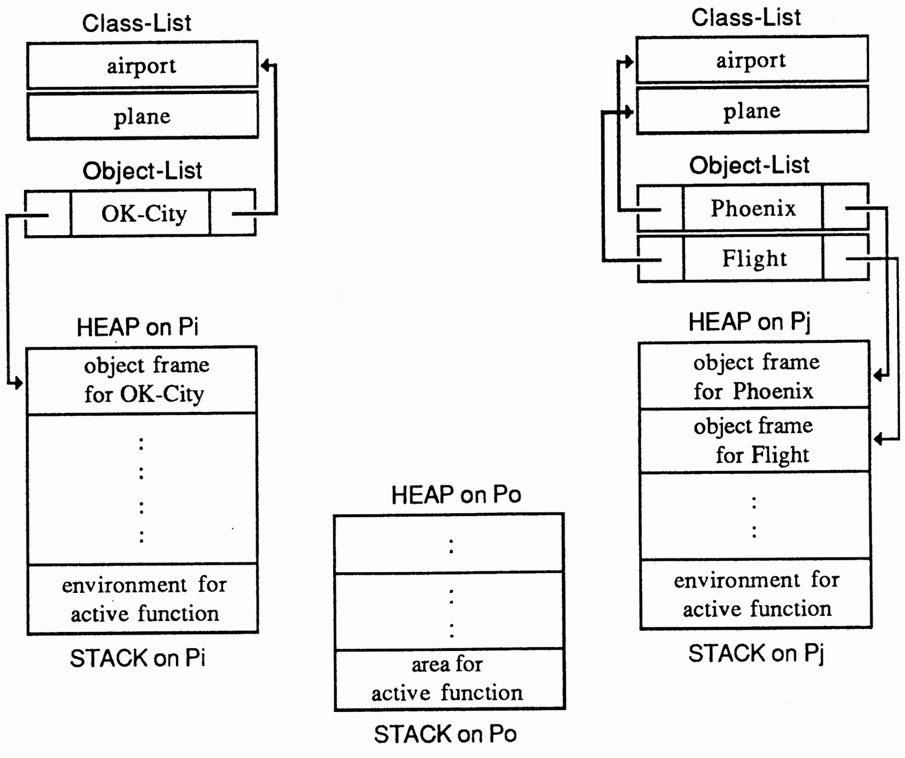


<b>

Figure 5.7. Run-Time Storage with Distributed and Dynamic Objects



<c>



<d>

Figure 5.7. (Continued)

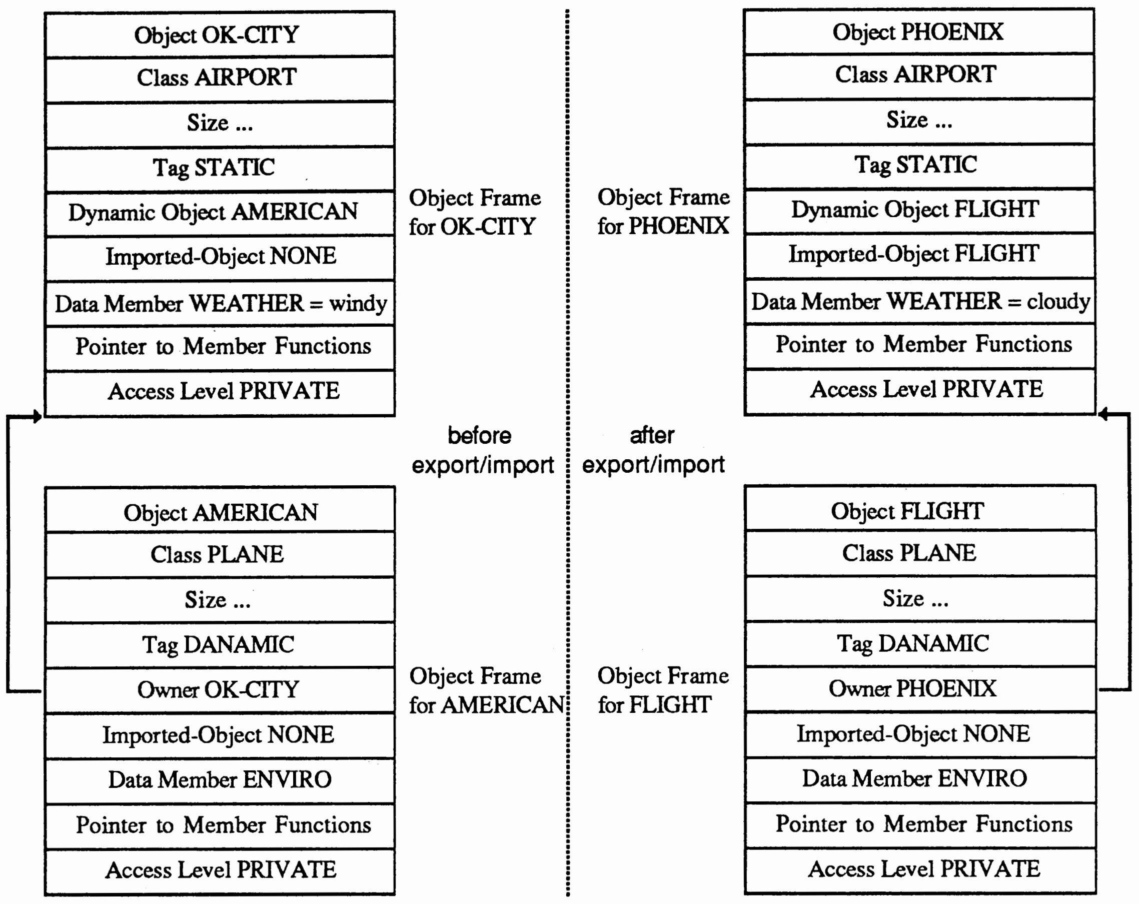
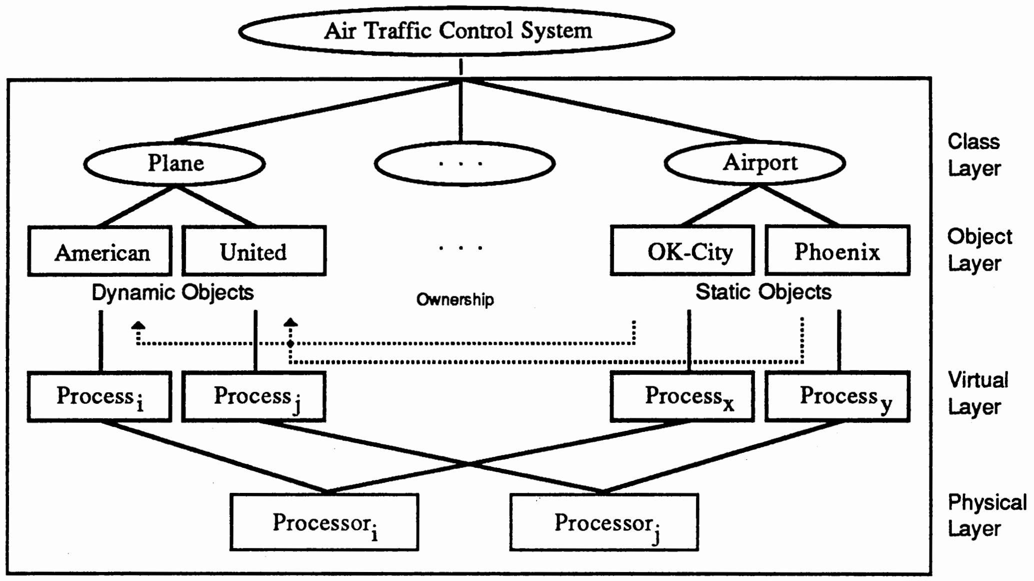


Figure 5.8. Distributed Object Frames

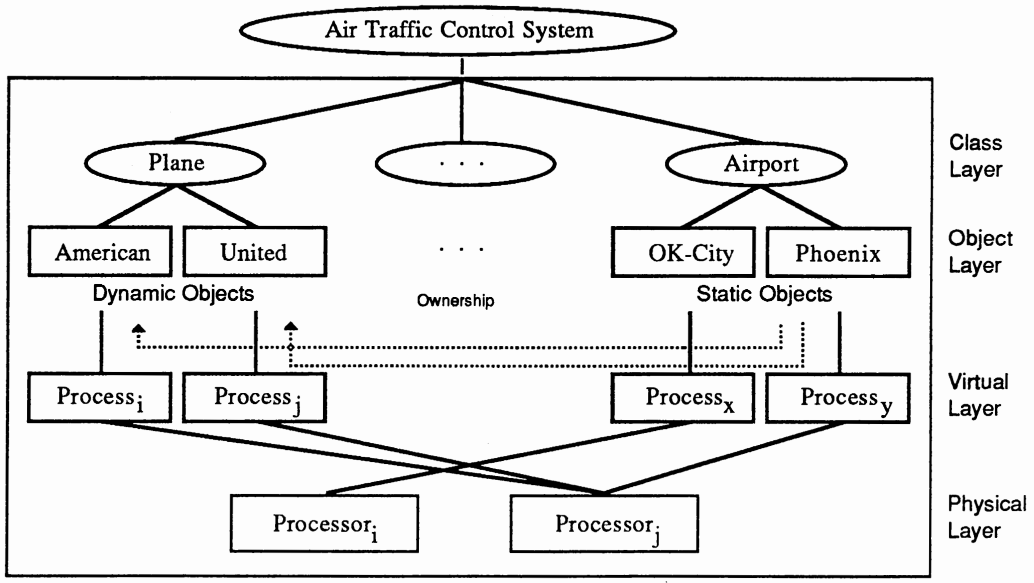
The planes, taking off and landing, may need weather information of the airports. The member functions, "takeoff" and "landing", of the plane objects use the pointer, "owner", to get current weather information from the member data, "weather", of the airport objects. The owner's address can be taken from the variable, "pointer to the owner", in the object's frame which has been established in the memory where the importing object resides. When the importing object establishes the imported object's frame, the importing object sets its value of "this" - which is a pointer to the importing object itself - to the "owner" pointer in the imported object's frame. The exported plane can get local weather information from the various environments of the importing airports. The computation in the environment provided by the exporting object can be different from the computation in the environment provided by the importing object, while changing the ownership. This is one of the typical examples for Dynamic Interface Environment Binding. For example, in Figure 5.8, the plane "American" (in the exporting process "P_i") takes off with the "windy" weather condition of the airport "OK-City", and the plane "Flight" (which is actually the same object as "American" - only its symbolic name is changed in the importing process "P_j" after export/import) is landing with the "cloudy" weather condition of the airport "Phoenix". Figure 5.8 shows the object frames (magnified from the object frames shown in Figure 5.7) of the distributed objects in several processors while changing ownership by "export/import" operations.

5.6 A System View

We show a specific system view for this particular example. (Refer to Section 2.4 for a Parallel-C++ system view in general.) Conceptually we may view a Parallel-C++ system on the distributed processor as an environment consisting of a set of four layers - class layer, object layer, virtual layer and physical layer. Each layer is mapped to the layer below. Figure 5.9 illustrates the four layers using the example in the previous section. The environment "Air_Traffic_Control_System" has several classes, such as "Airport" and "Plane". The objects "OK_City" and "Phoenix" are the instances of class "Airport", and the objects "American" and "United" are the instances of class "Plane". We remind the reader that the class layer and object layer have instantiation relation, the object layer and virtual layer have virtual implementation relation, and the virtual layer and physical layer have physical mapping relation. Figure 5.9<a> shows the system view before "export" and "import" operations (Line-c and line-c' in Figure 5.6), and Figure 5.9<b> shows the change in system view after such operations. Dotted line with arrow shows the ownership (for example, "OK_City" owns "American" in Figure 5.9<a>).



<a> System View before Export/Import



<b> System View after Export/Import

Figure 5.9. System View

5.7 Discussion of Compiler Implementation

In the distributed memory, it is assumed that the storage for a dynamic object should be allocated in the local memory of the processor in which the importing object resides. The high cost for frequent communication between distributed objects can deteriorate the benefit of the distributed dynamic object-oriented programming. Because of the characteristics of distributed memory systems (without any shared memory), global variables or pointer variables cannot be used, and pass-by-reference mode cannot be used for the parameter passing, especially with the dynamic objects. Those restrictions are cumbersome barriers for users to make sophisticated programs. The development of a better scheme and a more sophisticate run-time system is needed to solve that problem. Fine-grained parallelism does not seem to be a promising scheme for this kind of parallel model. Better efficiency can be expected when medium/coarse-grained parallelism is pursued. Without using objects in simple conventional parallel programming, optimization cost (to do interdependency checking or restructuring, for example) is not negligible. The cost for fine-grained parallelism, such as parallel programming in the statement level, outweighs its benefits. Efficiency should be properly measured and carefully considered.

CHAPTER VI

CONCLUSION AND SUGGESTIONS FOR FUTURE WORK

6.1 Epilogue

This dissertation presents the programming language design concepts, language definition, semantics and implementation schemes for a distributed object-oriented parallel programming language, Parallel-C++. The language incorporates novel concepts that support object-oriented programming within a distributed computing environment. Parallel-C++ is a high-level language that supports object-oriented parallel and distributed programming.

We have also presented an implementation scheme for a Parallel-C++ translator. The translation techniques described in this dissertation have been used in the implementation of the prototype Parallel-C++ translator which is currently running on the Intel iPSC/2.

Furthermore, based on the experience gained from implementing the translator, the implementation scheme for a compiler has also been defined. Run-time storage management is well illustrated and exemplified.

Parallel-C++ is an extension of the C++ language. Language constructs are added to support explicit parallelism within the object-oriented programming paradigm. Parallel-C++ preserves the properties of object-oriented programming, such as information hiding and inheritance, while providing support for parallel and distributed computing. In Parallel-C++ there are two kinds of objects: (1) *static objects* that reside at a certain process at a certain time; and (2) *dynamic objects* that float from one static object (or process) to another. This concept can support *object migration*. However, the concept of static objects is relative to the concept of dynamic objects. Dynamic objects can be exported/imported between static objects in a distributed system. Parallel-C++ also supports explicit parallelism and implicit parallelism. Compared with the other notions, it provides a very readable and simple code for object distribution and movement in application programs. Cost of object movement in the distributed memory is high compared to that in a shared memory system. The discussion of the object mobility in the distributed memory system provides a new approach. Most systems for object mobility available in the literature have addressed the issue of object mobility in the context of a single address space and a shared memory. Programming using distributed and dynamic objects in Parallel-C++ provides a new control flow and shows a different scope rule. Major new ideas and concepts of the programming language design presented in this dissertation are:

(1) Distributed dynamic objects and static objects.

(2) Ownership which is a dynamic relationship between objects.

(3) Dynamic Interface Environment Binding (DIEB) which permits computation of objects in different environments.

(4) A new approach to distributed object-oriented parallel programming.

(5) Implementation schemes for dynamic objects in a distributed computing environment.

The parallelizing translator implementation described in this dissertation has been targeted to an Intel iPSC/2 hypercube multiprocessing system. The implementation schemes for a compiler and run-time interpreter are also provided. The major implementation schemes suggested are:

(1) A translator dynamically parallelizing programs for explicit parallelism.

(2) Automatic processor allocation for distributed programs.

(3) Support for object movements on the physically distributed address space.

(4) Distributed memory allocation for each local processor.

(5) Implicit deallocation of the freed area for the exported objects and implicit allocation of the environment for the imported objects.

(6) A compiler implementation scheme restructuring programs for implicit/explicit parallelism in the distributed memory computing environment.

6.2 Contribution and Further Research

The main contribution of distributed and dynamic objects is to provide new control flow of programs. This is illustrated with other control flows provided by existing programming languages (as shown in Section 2.6). Programming using dynamic objects can also simulate other control flow schemes.

Dynamic objects also add enhancement to the scope rules (as shown in Section 2.6). Dynamic objects are semi-persistent, because their lifetimes are spanned over other static objects importing such objects.

One of the main characteristics of programming language is expressiveness. Parallel language constructs suggested in Parallel-C++ help programmers write simple and readable parallel programs. A programmer may define concurrency explicitly in his program. The parallelizing translator or compiler may detect and restructure the program in which concurrency is implicitly or explicitly defined.

Another distinguishing characteristic of this work is that the current implementation shows the implementation scheme of distributed objects based on a distributed memory computing system, while most other work has been done using models based on shared memory systems. The implementation of distributed objects and object migration in the shared memory system is much simpler than that in the distributed memory system. It is also possible to think of a hybrid model which has distributed memory with shared resources such as printers and disks.

Besides the refinements to the work suggested in this dissertation, the valuable issues for future work also include development of formal/mathematical models [Manes and Arbib 86], and formal semantics [America et al. 86] [Hennessy 88,90] [Schmidt 86] of distributed dynamic objects. The development of proof methods, specification methods [Meyer 85], and debugging methods [McDowell and Helmbold 89] [Utter and Pancake 89] for proving correctness of program using dynamic objects are other important problems to be addressed.

BIBLIOGRAPHY

[Ackerman 82] Ackerman, William B. Data Flow Languages. IEEE Computer, (February 1982), 15-25.

[Ada 79] Preliminary ADA Reference Manual, and Rationale for the Design of the ADA Programming Language, SIGPLAN Notices, 14(6), (Part A and Part B), (June 1979).

[Ada 83] Reference Manual for the Ada Programming Language. ANSI/MIL-STD-1815A-1983, United States Department of Defense, (American National Standards Institute, Inc.), (Feb. 17, 1983), [Reprinted in Horowitz, E. (Ed.) Programming Languages: A Grand Tour (2nd Ed.), Computer Science Press (1985), 417-751].

[Agha 86] Agha, Gul. An Overview of Actor Languages. OOP Workshop, (June 9-13, 1986), SIGPLAN Notices (October 1986), 58-67.

[Agha 86b] Agha, Gul. Actors: A Model of Concurrent Computation in Distributed Systems. The MIT Press, (1986).

[Agha 89] Agha, Gul. Foundation issues in concurrent computing. Proc. of ACM SIGPLAN Workshop on Object-Based Concurrent Programming. SIGPLAN Notices 24(4), (April 1989), 60-65.

[Agerwala and Arvind 82] Agerwala, T. and Arvind. Data flow systems. Guest Editors' Introduction. IEEE Computer, (Special Issues for Data Flow), (Feb. 1982), 10-13.

[Aho et al. 86] Aho, A. V., Sethi, R. and Ullman, J. D. Compilers - Principles, Techniques, and Tools. Addison-Wesley Publishing Co. Reading, MA, (1986).

[Allen and Kennedy 82] Allen, John R. and Kennedy, Ken. A program to convert Fortran to parallel form. Rice MASC TR82-6, Department of Mathematical Sciences, Rice University, Houston, TX 77001, (March 1, 1982).

[Allen and Kennedy 85] Allen, J. R. and Kennedy, D. A parallel programming Environment. IEEE Software, (July 1985), 21-29.

[Allen and Kennedy 87] Allen, Randy and Kennedy, Ken. Automatic translation of FORTRAN programs to vector form. ACM ToPLaS, 9(4), 491-542, (October 1987).

[Almasi and Gottlieb 89] Almasi, G. S. and Gottlieb, A. Highly Parallel Computing. The Benjamin/Cummings Publishing Co. Inc., Redwood City, CA 94065, (1989).

[America et al. 86] America, P., de Bakker, J., Kok, J. N. and Rutten, J. Operational semantics of a parallel object-oriented language. ACM PoPL Proc., (1986), 194-208.

[Andersen 89] Andersen, B. Hypercube experiments with Joyce. SIGPLAN Notices, 24(8), (August 1989), 13-22.

[Ardo and Philipson 84] Ardo, Anders and Philipson, Lars. Implementation of a Pascal based parallel language for a multiprocessor computer. Software-Practice and Experience, 14(7), (July 1984), 643-657.

[Artsy and Finkel 89] Artsy, Y. and Finkel, R. Designing a process migration facility: The Charlotte experience. IEEE Computer, 22(9), (Sept. 1989), 47-56.

[Atkins and Nackman 88] Atkins, M. C. and Nackman, L. R. The active deallocation of objects in object-oriented systems. Software-Practice and Experience, 18(11), (November 1988), 1073-1089.

[AT&T 85] UNIX System V AT&T C++ Translator Release Notes. #307-175, AT&T, (1985).

[AT&T 86] AT&T C++ Translator Release 1.2 Addendum to the Release Notes. #307-005, AT&T,(1986).

[AT&T 89] UNIX System V AT&T C++ Language System, Release 2.0. Selected Readings. Select Code 307-144. AT&T, (1989).

[AT&T 89b] UNIX System V AT&T C++ Language System, Release 2.0. Product Reference Manual. Select Code 307-146. AT&T, (1989).

[August et al. 89] August, M.C., Brost, G.M., Hsiung, C.C. and Schiffleger, A.J. Cray X-MP: The birth of a supercomputer. Computer 22(1), (Jan. 1989), 45-52.

[Bakker et al. 87] Bakker, J. W., Nijman, A. J. and Treleaven, P. C. (Eds.) Proceedings of the conference on Parallel Architectures and Languages Europe(PARLE). Eindhoven, The Netherlands, June 1987, Vol.1 and 2, Lecture Notes in Computer Science, No.258 and 259, Springer-Verlag, Germany, (1987).

[Bal and Tanenbaum 88] Bal, H.E. and Tanenbaum, A.S. Distributed programming with shared data. IEEE Computer Society 1988 International Conference on Computer Languages, Miami Beach, Florida, (Oct. 9-13, 1988), 82-91.

[Bal et al. 89] Bal, H.E., Steiner, J.G. and Tanenbaum, A.S. Programming languages for distributed computing systems. ACM Computing Surveys, 21(3), (September 1989), 261-322.

[Banerjee 86] Banerjee, Utpal. A direct parallelization of call statements - A review. CSRD Rpt. No.576, Univ. of Illinois at Urbana-Champaign, (April 1986).

[Baskett and Hennessy 86] Baskett, F. and Hennessy, J. L. Small shared-memory multiprocessors. Science, V.231, (Feb. 28, 1986), 963-967.

[Beck 90] Beck, Bob. Shared-memory parallel programming in C++. IEEE Software, 7(4), (July 1990), 38-48.

[Bennett 87] Bennett, J. K. The design and implementation of Distributed Smalltalk. OOPSLA '87 Proceedings, SIGPLAN Notices 22(12), (Dec. 1987), 318-330.

[Bennett 90] Bennett, J. K. Experience with Distributed Smalltalk. Software-Practice and Experience, 20(2), (Feb. 1990), 157-180.

[Bershad et al. 88] Bershad, B. N., Lazowska, E. D. and Levy, H. M. PRESTO: a system for object-oriented parallel programming. Software-Practice and Experience, 18(8), (August 1988), 713-732.

[Birkhoff 82] Birkhoff, G. Some applications of universal algebra. Csákány, G., Fried, E. and Schmidt, E. T. (Eds.) Colloquium on Universal Algebra, Esztergom, Hungary, (June 27-July 1, 1977), North-Holland (1982), 107-128.

[Björnerstedt and Britts 88] Björnerstedt, A. and Britts, S. AVANCE: An Object Management System. OOPSLA '88 Proceedings, 206-221, (1988).

[Black et al. 86] Black, A., Hutchinson, N., Jul, E. and Levy, H. Object structure in the Emerald system. OOPSLA '86 Proc. SIGPLAN Notices 21(11), (Nov. 1986), 78-86.

[Brinch-Hansen 73] Brinch-Hansen, Per. Concurrent programming concepts, ACM Computing Survey 5(4), (Dec. 1973), 223-245.

[Brinch-Hansen 74] Brinch-Hansen, Per. A programming methodology for operating system design. Proc. of the Information Processing (IFIP '74), North-Holland Pub. (1974), 394-397.

[Brinch-Hansen 75] Brinch-Hansen, Per. The programming language Concurrent Pascal. IEEE Trans. on Software Eng. SE-1(2), (June 1975), 199-207.

[Brinch-Hansen 75b] Brinch-Hansen, Per. The purpose of Concurrent Pascal. Proc. of the International Conference on Reliable Software, SIGPLAN Notices 10(6), 305-309, (June 1975).

[Brinch-Hansen 87] Brinch-Hansen, P. Joyce - a programming language for distributed systems. Software-Practice and Experience 17(1), 29-50, (Jan. 1987).

[Brinch-Hansen 87b] Brinch-Hansen, P. A Joyce implementation. Software-Practice and Experience, 17(4), 267-276, (April 1987).

[Bronnenberg et al. 86] Bronnenberg, W.J.H.J., Janssens, M.D., Odijk, E.A.M. and van Twist, R.A.H. The architecture of DOOM, Proc. of Future Parallel Computers, An Advanced Course, Pisa, Italy, (June 9-20, 1986), Lecture Notes in Computer Science, No.272, Springer-Verlag, (1987), 227-269.

[Brown M89] Brown, M. L. A hypercube image processing language. C3P-753,754, California Institute of Technology, (April 26, 1989).

[Carriero and Gelernter 89] Carriero, N. and Gelernter, D. How to write parallel programs: a guide to the perplexed. ACM Computing Surveys 21(3), (Sept. 1989), 323- 357.

[Chandy and Misra 88] Chandy, K. Mani and Misra, Jayadev. Parallel Program Design: A Foundation. Addison-Wesley, (1988).

[Chase et al. 89] Chase, J.S., Amador, F.G., Lazowska, E.D., Levy, H.M. and Littlefield, R.J. The Amber System: Parallel programming on a network of multiprocessors. Proc. of the 12th ACM Symposium on Operating Systems Principles, ACM/SIGOPS Operating Systems Review 23(5), (1989), 147-158.

[Cheng 89] Cheng, Hui. Vector pipelining, chaining, and speed on the IBM 3090 and Cray X-MP. IEEE Computer, 22(9), (Sept. 1989), 31-46.

[Clapp and Mudge 89] Clapp, Russell M. and Mudge, Trevor. Ada on a hypercube. Ada Letters, 9(2), (March/April 1989), 118-128.

[Clark and Gregory 86] Clark, Keith and Gregory, Steve. PARLOG: Parallel programming in Logic. ACM Trans. on Programming Languages and Systems, 8(1), (January 1986), 1-49.

[Clark 87] Clark, Keith. PARLOG: The language and its applications. Proc. of PARLE(1987), V.2, Lecture Notes in Computer Science,#259,Springer-Verlag, 30-53.

[CLOS 88] Common Lisp Object System Specification. X3J13 Document 88-002R, (by Bobrow, D.G., DeMichiel, L.G., Gabriel, R.P., Keene, S.E., Kiczales, G. and Moon, D.A.), SIGPLAN Notices, V.23, Special Issue, (Sept. 1988).

[Cmelik et al. 89] Cmelik, R. F., Gehani, N. H. and Roome, W. D. Experience with multiple processor versions of Concurrent C. IEEE Trans. on Software Engineering,15(3), (March 1989), 335-344.

[Codish and Shapiro 87] Codish, M. and Shapiro, E. Compiling OR-parallelism into AND-parallelism. New Generation Computing, V.5, (1987), 45-61.

[Conery and Kibler 85] Conery, J. S. and Kibler, D. F. AND parallelism and nondeterminism in logic programs. New Generation Computing, V.3, (1985), 43-70.

[Corradi and Leonardi 87] Corradi, Antonio and Leonardi, Letizia. How to embed concurrency within an object environment: Parallel Objects. Proc. of the International Conference on Parallel Processing and Applications. (Italy, Sept. 1987), 79-84, (Ed. by E. Chiricozzi and A. D'amico, North-Holland. 1988).

[Corradi and Leonardi 87b] Corradi, A. and Leonardi, L. An environment based on Parallel Objects: PO. Proc. of the IEEE Phoenix Conference on Computers and Communications, (1987), 253-257.

[Corradi and Leonardi 88] Corradi, A. and Leonardi, L. The specification of concurrency: an object-based approach. Proc. of the 7th International Phoenix Conference on Computers and Communications, (1988), 246-250.

[Corradi and Leonardi 89] Corradi, A. and Leonardi, L. PO: An object model to express parallelism. Proc. of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming. SIGPLAN Notices, 24(4), (April 1989), 152-155.

[Corradi and Leonardi 90] Corradi, A. and Leonardi, L. A concurrent prototype in Smalltalk-80. COMPEURO '90, Tel-Aviv, Israel, (1990).

[Corsini et al. 89] Corsini, P., Frosini, G. and Speranza, G. The parallel interpretation of logic programs in distributed architectures. The Computer Journal, 32(1), (1989), 29-35.

[Cox 83] Cox, Brad J. The object-oriented pre-compiler: Programming Smalltalk 80 methods in C language. SIGPLAN Notices, 18(1), 15-22, January 1983.

[Cox 86] Cox, Brad J. Object-Oriented Programming: An Evolutionary Approach. Addison-Wesley, Reading, MA. 1986.

[Cox and Schmucker 87] Cox, B. J. and Schmucker, K. J. Producer: A tool for translating Smalltalk-80 to Objective-C. OOPSLA '87 Proc. SIGPLAN Notices 22(12), (Dec. 1987), 423-429.

[Cox and Gehani 87] Cox, I. J. and Gehani, N. H. Concurrent C and Robotics. IEEE Intl. Conf. on Robotics and Automation, (1987),1463-1468.

[Cox 88] Cox, I. J. C++ language support for guaranteed initialization, safe termination and error recovery in robotics. IEEE Intl. Conf. on Robotics and Automation, (1988), 641-643.

[Cox et al. 88] Cox, Ingemar J., Kapilow, D.A., Kropfl, W.J. and Shopiro, J.E. Real-time software for robotics. AT&T Technical Journal, 67(2), (March/April 1988), 61-70.

[Cybenko 89] Cybenko, G. Dynamic load balancing for distributed memory multiprocessors. Journal of Parallel and Distributed Computing. 7, 279-301, (1989).

[Dally 88] Dally, W. J. Fine-grain message-passing concurrent computers. Proc. of the 3rd Conference on Hypercube Concurrent Computers and Applications, Vol.1, 2-12, Pasadena, CA, (Jan. 19-20, 1988).

[Date 90] Date, C. J. An Introduction to Database Systems. (V.1, 5th Ed.), Addison-Wesley Pub., (1990).

[DeBenedictis 88] Debenedictis, E. P. Multiprocessor architectures are converging. Proc. of the 3rd Conference on Hypercube Concurrent Computers and Applications, Vol.1, Pasadena, CA, (Jan. 19-20, 1988), 13-20.

[Deutsch 89] Deutsch, L. P. The past, present, and future of Smalltalk. ECOOP '89 Proceedings, Cambridge Univ. Press, (1989), 73-87.

[Dijkstra 68] Dijkstra, E. W. Cooperating sequential processes. F Genuys(Ed.), Programming Languages, Academic Press, London, pp.43-112, 1968.

[Dijkstra 75] Dijkstra, E. W. Guarded commands, nondeterminacy and formal derivation of programs. Comm. of the ACM, 18(8), (August 1975), 453-457.

[Dinning 89] Dinning, Anne. A survey of synchronization methods for parallel computers. IEEE Computer, 22(7), (July 1989), 66-77.

[Ditlefs et al. 88] Detlefs, D.L., Herlihy, M.P. and Wing, J. M. Inheritance of synchronization and recovery properties in Avalon/C++. IEEE Computer, (December 1988), 57-69.

[DoD 77] Department of Defence Requirements for high order computer programming languages, Revised "IRONMAN", (July 1977), SIGPLAN Notices 12(12), (Dec.1977), 39-54.

[Dongarra 87] Dongarra, J. J. (Editor). Experimental Parallel Computing Architectures. North-Holland, (Publisher) Elsevier Science Publishers B. V., P.O.Box 1991, 1000 BZ, Amsterdam, The Netherlands, (Distributor) Elsevier Science Publishing Company, Inc., 52 Vanderbilt Avenue, New York, N.Y. (1987).

[Douglis and Ousterhout 87] Douglis, F. and Ousterhout, J. Process migration in the Sprite operating system. Proc. of the 7th International Conference on Distributed Computing Systems, (1987), 18-25.

[Duncan 90] Duncan, R. A survey of parallel computer architectures. IEEE Computer, 23(2), (Feb. 1990), 5-16.

[Emrath 85] Emrath, Perry. Xylem: An operating system for the Cedar multiprocessor. IEEE Software, (July 1985), 30-37.

[Emrath et al. 88] Emrath, P., Padua, D. and Yew P.-C. Cedar architecture and its software. CSRD Rpt. No.796. Univ. of Illinois, Urbana, (June 1988).

[Fidge 88] Fidge, C. J. A Lisp implementation of the model for 'Communicating Sequential Processes'. Software-Practice and Experience, 18(10), (Oct. 1988), 923-943.

[Fisher 76] Fisher, D. A. A common programming language for the Department of Defense -- Background and technical requirements. Paper P-1191, Institute for Defense Analyses, Science and Technology Division, 400 Army-Navy Drive, Arlington, Virginia 22202, (June 1976).

[Fox 87] Fox, Geoffrey C. Domain decomposition in distributed and shared memory environments. Tech. Rpt. C3P-392, California Institute of Tech. (June 8, 1987).

[Fox 87b] Fox, G. C. The hypercube as a supercomputer. Proc. of the 2nd International Conference on Supercomputing (ICS '87), Vol.1, (1987), 186-194.

[Fox et al. 88] Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J. and Walker, D. Solving Problems on Concurrent Processors. Vol.1, Prentice-Hall, (1988).

[Fox 88] Fox, Geoffrey. (Ed.) Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications, Vol.1, Architecture, Software, Computer Systems and General Issues. Pasadena, California, (January 19-20, 1988).

[Fox 89] Fox, Geoffrey C. 1989 - The first year of the parallel supercomputer. Tech Rpt. C3P-769, California Institute of Technology, (June 6,1989).

[Franta 78] Franta, W. R. SIMULA language summary. ACM SIGPLAN History of Programming Languages Conference, SIGPLAN Notices 13(8), (Aug. 1978), 243-244.

[Gabriel 86] Gabriel, R. P. Massively parallel computers: The Connection Machine and NON-VON. Science, V.231, (Feb. 28, 1986), 975-978.

[Gaudiot and Lee 87] Gaudiot, J. L. and Lee, L. T. Multiprocessor systems programming in a high-level data-flow language. Proc. of PARLE(1987), V.1, Lecture Notes in Comp. Sci., #258, Springer-Verlag, 134-151.

[Gaudiot and Lee 89] Gaudiot, J. L. and Lee, L. T. Occamflow: A methodology for programming multiprocessor systems. Journal of Parallel and Distributed Computing 7, (1989), 96-124.

[Gavish and Sheng 90] Gavish, B. and Sheng, O.R.L. Dynamic file migration in distributed computer systems. CACM 33(2), (Feb. 1990), 177-189.

[Gehani and Roome 86] Gehani, Narain H. and Roome, William D. Concurrent C. Software-Practice and Experience, 16(9), pp.821-884, September 1986.

[Gehani and Roome 88] Gehani, N. H. and Roome, W. D. Rendezvous facilities: Concurrent C and the Ada Language. IEEE Trans. on Software Engineering, 14(11), November 1988.

[Gehani and Roome 88b] Gehani, N. H. and Roome, W. D. Concurrent C++: Concurrent programming with class(es). Software-Practice and Experience, 18(12), (December 1988), 1157-1177.

[Gehani and Roome 90] Gehani, N. H. and Roome, W. D. Concurrent C: a language for programming multiprocessor systems. Byte (December 1990), 327-334.

[Gehringer et al. 88] Gehringer, E.F., Abullarade, J. and Gulyn, M.H. A survey of commercial parallel processors. Computer Architecture News 16(4), (Sept. 1988), 75-107.

[Gelernter 85] Gelernter, David. Generative communication in Linda. ACM Trans. on Programming Languages and Systems, 7(1), (Jan. 1985), 80-112.

[Goguen et al. 87] Goguen, J., Kirchner, C., Meseguer, J. and Winkler, T. OBJ as a language for concurrent programming. Proc. of the 2nd International Conference on Supercomputing (ICS '87), Vol.1, (1987), 196-198.

[Goldberg and Robson 83] Goldberg, Adele and Robson, Dave. Smalltalk-80: The Language and Its Implementation. Addison-Wesley Pub. Co., Reading, MA, (1983).

[Goldberg and Robson 89] Goldberg, Adele and Robson, Dave. Smalltalk-80: The Language. Addison-Wesley Pub. Co., Reading, MA, (1989).

[Goldberg and Hudak 88] Goldberg, B. and Hudak, Paul. Implementing functional programs on a hypercube multiprocessor. Proc. of the 3rd Conference on Hypercube Concurrent Computers and Applications, Vol.1, Pasadena, CA, (Jan. 19-20, 1988), 489-504.

[Gottlieb et al. 83] Gottlieb, A., Grishman, R., Kruskal, C.P., McAuliffe, K.P., Rudolph, L. and Snir, M. The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer. IEEE Trans. on Computers, C-32(2), (Feb. 1983), 175-189.

[Grätzer 79] Grätzer, George. Universal Algebra. Springer-Verlag, New York, (1979).

[Gries 71] Gries, David. Compiler Construction for Digital Computers. John Wiley & Sons, (1971).

[Grogono 89] Grogono, P. Design criteria for a simple object-oriented language. OOP-89-5, Dept. of Computer Science, Concordia University, Montreal, Quebec, H3G 1M8, Canada, (1989).

[Grogono and Bennett 89] Grogono, P. and Bennett, A. A theory for object-oriented languages. (Extended Abstract), OOP-89-1, Dept. of Computer Science, Concordia University, Montreal, Quebec, H3G 1M8, Canada, (1989).

[Guarna 87] Guarna, Vincent A. Jr. VPC - a proposal for a vector parallel C programming language. CSRD Rpt. No.666, Univ. of Illinois at Urbana-Champaign, June 16, 1987.

[Guarna 88] Guarna, Vincent A. Jr. A technique for analyzing pointer and structure references in parallel restructuring compilers. CSRD Rpt. No.721, Univ. of Illinois at Urbana-Champaign, (January 1988).

[Guzzi 87] Guzzi, Mark D. Cedar Fortran Programmer's Manual *Draft*, CSRD Document No.601, Univ. of Illinois at Urbana-Champaign, April 13, 1987.

[Halstead 85] Halstead, Robert H. Jr. Multilisp: A language for concurrent symbolic computation. ACM Trans. on Programming and Systems, 7(4), (October 1985), 501-538.

[Haynes et al. 82] Haynes, L.S., Lau, R.L., Siewiorek, D.P. and Mizell, D.W. A survey of highly parallel computing. IEEE Computer, (January 1982), 9-24.

[Hennessy 88] Hennessy, Matthew. Algebraic Theory of Processes. The MIT Press, (1988).

[Hennessy 90] Hennessy, Matthew. The Semantics of Programming Languages: An Elementary Introduction using Structured Operational Semantics, John Wiley and Sons, Pubs., (1990).

[Hoare 73] Hoare, C.A.R. Hints on programming language design. STAN-CS-73-403, Computer Science Department, Stanford University, (Oct. 1973).

[Hoare 74] Hoare, C.A.R. Monitors: An operating system structuring concept. Comm. of the ACM, 17(10), (Oct. 1974), 549-557.

[Hoare 78] Hoare, C.A.R. Communicating sequential processes, Comm. of the ACM, 21(8), (Aug.1978), 666-677.

[Hoare 85] Hoare, C.A.R. Communicating Sequential Processes. Prentice Hall International, UK, (1985).

[Hudak and Smith 86] Hudak, P. and Smith, L. Para-functional programming: A paradigm for programming multiprocessor systems. ACM PoPL '86 Proc., (1986), 243-254.

[Hur and Chon 87] Hur, Jin H. and Chon, Kilnam. Overview of a parallel object oriented language CLIX. CS-TR-87-25, Computer Science Department, Korea Advanced Institute of Science and Technology, Seoul, Republic of Korea, (1987), and to appear in Proc. of European Conf. on Object-Oriented Programming, Paris, (June 1987).

[Intel 88] Intel Corporation. iPSC/2 User's Guide and C Programmer's Reference Manual. (March 1988).

[Jazayeri 89] Jazayeri, Mehdi. Objects for distributed systems. Proc. of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, SIGPLAN Notices 24(4), (April 1989), 117-119.

[Jo et al. 89] Jo, Chang-Hyun, Fisher, Donald D. and George, K. M. Abstraction and Specification of Local Area Networks. Proceedings of the Eighth Annual International Phoenix Conference on Computers and Communications, (March 22-24, 1989), Scottsdale, Arizona, IEEE Computer Society Press, (1989), 337-342.

[Jo and George 89] Jo, Chang-Hyun and George, K. M. Distributed object-oriented programming with dynamic objects. OSU-CS-TR-89-14, Department of Computer Science, Oklahoma State University (1989).

[Jo and George 91] Jo, Chang-Hyun and George, K. M. Language concepts using dynamic and distributed objects. Proceedings of the ACM 1991 Computer Science Conference (ACM/CSC '91), (March 5-7, 1991) San Antonio, Texas, ACM Press (1991), 211-220.

[Jo et al. 91] Jo, Chang-Hyun, George, K. M. and Teague, K. A. Parallelizing translator for an object-oriented parallel programming language. Proceedings of the Tenth Annual International Phoenix Conference on Computers and Communications (IPCCC '91), (March 27-30, 1991), Scottsdale, Arizona, IEEE Computer Society Press, (1991), 265-271.

[Johnson 78] Johnson, Stephen C. Yacc: Yet Another Compiler-Compiler. Computing Sci. Tech. Rpt. No.32, Bell Laboratories, Murray Hill, New Jersey 07974, (1975 and July 31, 1978).

[Jul et al. 88] Jul, E., Levy, H., Hutchinson, N. and Black, A. Fine-grained mobility in the Emerald system. ACM Transactions on Computer Systems, 6(1), (Feb. 1988), 109-133.

[Kafura 88] Kafura, Dennis. Concurrent object-oriented real-time systems research. TR 88-47, Dept. of Comp. Sci., Virginia Tech, Blacksburg, VA 24061, 1988.

[Kafura and Lee 89] Kafura, D. G. and Lee, Hae Keung. Inheritance in Actor based concurrent object-oriented languages. ECOOP '89 Proceedings, Cambridge Univ. Press, (1989), 132-145.

[Kamin 88] Kamin, Samuel. Inheritance in SMALLTALK-80: A Denotational Definition. Proc. of PoPL,(1988), 80-87.

[Kamin 90] Kamin, Samuel N. Programming Languages: An Interpreter-Based Approach. Addison-Wesley, (1990).

[Karp and Babb 88] Karp, Alan H. and Babb, Robert G. II. A comparison of 12 parallel Fortran dialects. IEEE Software, (September 1988), 52-67.

[Keene 89] Keene, Sonya E. Object-Oriented Programming in COMMOM LISP: A Programmer's Guide to CLOS. Addison-Wesley Pub.Co. (1989).

[Kempf et al. 87] Kempf, J., Harris, W., D'Souza, R. and Snyder, A. Experience with CommonLoops. OOPSLA '87 Proc., SIGPLAN Notices 22(12), (Dec. 1987), 214-226.

[Kernighan and Ritchie 78,88] Kernighan, Brian W. and Ritchie, Dennis M. The C Programming Language. Prentice-Hall, Englewood Cliffs, NJ 07632, (1978, 2nd Ed. 1988).

[Kim et al. 87] Kim, Won, Banerjee, J. and Chou, Hong-Tai., Garza, J.F. and Woelk, D. Composite object support in an object-oriented database system. OOPSLA '87 Proc. SIGPLAN Notices 22(12), (Dec. 1987), 118-125.

[Kim and Browne 88] Kim, S.J. and Browne, J.C. A general approach to mapping of parallel computations upon multiprocessor architectures. Proc. of the 1988 International Conference on Parallel Processing, Vol.3, The Penn St. Univ. Press, (Aug.15-19, 1988), 1-8.

[Koschmann and Evens 88] Koschmann, T. and Evens, M. W. Bridging the gap between object-oriented and logic programming. IEEE Software, (July 1988), 36-42.

[Koszarek 88] Koszarek, J. L. Hardware support for distributed objects in a hypercube. Proc. of the 3rd Conference on Hypercube Concurrent Computers and Applications, Vol.1, Pasadena, CA, (Jan. 19-20, 1988), 26-32.

[Kuck et al. 86] Kuck, D.J., Davidson, E.S., Lawrie, D.H. and Sameh A.H. Parallel supercomputing today and the Cedar approach. Science, 231, (Feb. 1986), 967-974.

[Kung 80] Kung, H. T. The structure of parallel algorithms. Advances in Computers, Vol.19, (1980), 65-112.

[Kung 82] Kung, H. T. Why systolic architectures? IEEE Computer, (Jan. 1982),37-46.

[Lamport 74] Lamport, Leslie. The parallel execution of DO loops. Comm. of the ACM 17(2), (Feb. 1974), 83-93.

[Lamport 74b] Lamport, Leslie. A new solution of Dijkstra's concurrent programming problem. Comm. of the ACM, 17(8), (August 1974), 453-455.

[Lamport and Schneider 84] Lamport, L. and Schneider, F.B. The "Hoare Logic" of CSP, and all that. ACM Trans. on Prog. Lang. and Sys., 6(2), (April 1984), 281-296.

[LeBlanc et al. 88] LeBlanc, T.J., Scott, M.L. and Brown, C.M. Large-scale parallel programming: Experience with the BBN Butterfly Parallel Processor. ACM/SIGPLAN PPEALS, (July 19-21, 1988), SIGPLAN Notices, 23(9), (September 1988), 161-172.

[Lee et al. 85] Lee, Gyungho, Kruskal, C. P. and Kuck, D. J. An empirical study of automatic restructuring of nonnumerical programs for parallel processors. IEEE Trans. on Computers, C-34(10), (Oct. 1985), 927-933.

[Lee 88] Lee, Gyungho. Automatic restructuring of conditional cyclic loops. Proc. of the 1988 International Conference on Parallel Processing, (August 15-19), Vol.2, The Penn State Univ. Press, University Park, Pennsylvania, (1988), 42-45.

[Leler 90] Leler, Wm. Linda meets Unix. IEEE Computer, 23(2), 43-54, (Feb. 1990).

[Lesk and Schmidt 75] Lesk, M. E. and Schmidt, E. Lex - A lexical analyzer generator, Computing Sci. Tech. Rpt. No.39, Bell Laboratories, Murray Hill, New Jersey 07974, (July 21, 1975 and October 1975).

[Lesser 74] Lesser, Victor R. The design of an emulator for a parallel machine language. Proc. of ACM SIGPLAN-SIGMICRO Interface Meeting, (May 30-June 1, 1973), SIGPLAN Notices 9(8), (August 1974).

[Li 86] Li, Kuo-Cheng. A note on the Vector C language. SIGPLAN Notices, 21(1), (January 1986), 49-57.

[Li and Yew 88] Li, Zhiyuan and Yew, Pen-Chung. Program parallelization with interprocedural analysis. CSRD Rpt. No.775, Univ. of Illinois at Urbana-Champaign, (June 1988).

[Li and Yew 88b] Li, Zhiyuan and Yew, Pen-Chung. Efficient interprocedural analysis for program parallelization and restructuring. CSRD Rpt. No.804 Univ. of Illinois at Urbana-Champaign, (July 1988).

[Li and Yew 88c] Li, Zhiyuan and Yew, Pen-Chung. Interprocedural analysis for parallel computing. Proc. of the 1988 International Conference on Parallel Processing, Vol.2, (August 15-19, 1988), The Penn State Univ. Press, University Park, Pennsylvania, (1988), 221-228.

[Lieberherr and Holland 89] Lieberherr, K. J. and Holland, I. M. Assuring good style for object-oriented programs. IEEE Software, (Sept. 1989), 38-48.

[Lin and Kumar 88] Lin, Y. J. and Kumar, V. An execution model for exploiting AND-parallelism in logic programs. New Generation Computing, V.6, (1988), 393-425.

[Liskov and Zilles 75] Liskov, B. and Zilles, S. N. Specification techniques for data abstractions. IEEE Trans. on Software Engineering, SE-1(1), (March 1975), 7-19.

[Liskov et al. 77] Liskov, B., Snyder, A., Atkinson, R. and Schaffert, C. Abstraction mechanisms in CLU. CACM, 20(8), (Aug. 1977), 564-576, [Reprinted in Horowitz, E. (Ed.) Programming Languages: A Grand Tour (2nd Ed.), Computer Science Press (1985), 226-238].

[Liskov and Snyder 79] Liskov, B. and Snyder, A. Exception handling in CLU. IEEE Trans. on Software Eng. (Nov. 1979), [Reprinted in Horowitz, E. (Ed.) Programming Languages: A Grand Tour (2nd Ed.), Computer Science Press (1985), 239-251].

[Liskov 81] Liskov, Barbara. On linguistic support for distributed programs. Proc. of the Symposium on Reliability in Distributed Software and Database Systems, Pittsburgh, PA, (July 21-22, 1981), 53-60.

[Liskov and Scheifler 83] Liskov, B. and Scheifler, R. Guardians and Actions: Linguistic support for robust, distributed programs. ACM ToPLaS, 5(3), (July 1983), 381-404.

[Liskov and Guttag 86] Liskov, B. and Guttag, J. Abstraction and Specification in Program Development. The MIT Press, McGraw-Hill Book Co. (1986).

[Liskov et al. 86] Liskov, B., Herlihy, M. and Gilbert, L. Limitations of synchronous communication with static process structure in languages for distributed computing. ACM PoPL '86 Proc., (1986), 150-159.

[Liskov 87] Liskov, Barbara. Data abstraction and hierarchy. OOPSLA '87 Addendum to the Proc., Orlando, FL, (October 4-8, 1987), Special Issue of SIGPLAN Notices, 23(5), (May 1988), 17-34.

[Liskov et al. 87] Liskov, B., Curtis, D., Johnson, P. and Scheifler, R. Implementation of Argus. Proc. of the 11th ACM Symposium on Operating Systems Principles. ACM/SIGOPS 21(5), (1987), 111-122.

[Liskov 88] Liskov, B. Distributed programming in Argus. CACM, 31(3), (March 1988), 300-312.

[Liskov and Shrira 88] Liskov, Barbara and Shrira, Liuba. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. Proc. of the SIGPLAN '88 Conference on Programming Language Design and Implementation, Atlanta, Georgia, (June 22-24, 1988), SIGPLAN Notices 23(7), (July 1988), 260-267.

[Lucco 87] Lucco, S. E. Parallel programming in a virtual object space. OOPSLA '87, SIGPLAN Notices 22(12), (Dec. 1987), 26-34.

[Luckham et al. 84] Luckham, D. C., Von Henke, F. W., Larsen, H. J. and Stevenson, D. R. Adam: An Ada-based language for multiprocessing. Software-Practice and Experience, 14(7), (July 1984), 605-642.

[Maier et al. 86] Maier, D., Stein, J., Otis, A., and Purdy, A. Development of an object-oriented DBMS. OOPSLA '86 Proc., SIGPLAN Notices, 21(11), (Nov. 1986), 472-482.

[Manes and Arbib 86] Manes, Ernest G. and Arbib, Michael A. Algebraic Approaches to Program Semantics. Springer-Verlag, (1986).

[McDowell and Helmbold 89] McDowell, C.E. and Helmbold, D.P. Debugging Concurrent Programs. ACM Computing Surveys, 21(4), (Dec. 1989), 593-622.

[Meyer 85] Meyer, B. On formalism in specifications. IEEE Software, (January 1985), 6-26.

[Meyer 88] Meyer, Bertrand. Object-Oriented Software Construction. Prentice Hall International (UK), (1988).

[Meyer 88b] Meyer, Bertrand. Eiffel: A language and environment for software engineering. The Journal of Systems and Software (Elsevier Science Pub. Co., Inc.), Vol.8, (1988), 199-246.

[Meyer 89] Meyer, B. From structured programming to object-oriented design: The road to Eiffel. Structured Programming,10(1), Springer-Verlag, (1989), 19-39.

[Midkiff and Padua 87] Midkiff, S. P. and Padua, D. A. Compiler algorithms for synchronization, IEEE Trans. on Computers, C-36(12), (December 1987), 1485-1495.

[Milner 80] Milner, R. A Calculus of Communicating Systems, LNCS 92, Springer-Verlag, (1980).

[Moon 86] Moon, David A. Object-Oriented Programming with Flavors. OOPSLA '86 Proceedings. (September 1986).

[Myers et al. 90] Myers, B. A., Giuse, D. A., Dannenberg, R. B., Zanden, B. V., Kosbie, D. S., Pervin, E., Mickish, A., and Marchal, P. Garnet: comprehensive support for graphical, highly interactive user interfaces. IEEE Computer 23(11), (November 1990), 71-85.

[Nguyen 85] Nguyen, Van Long. A Theory of Processes. (Ph.D. Thesis), TR 85-691, Dept. of Computer Science, Cornell University, (June 1985).

[Nygaard and Dahl 78] Nygaard, K. and Dahl, O. The development of the SIMULA languages. ACM SIGPLAN Notices, 13(8), (Aug. 1978), 245-272.

[Nygaard 86] Nygaard, K. Basic concepts in object-oriented programming. OOP Workshop, SIGPLAN Notices 21(10), (Oct. 1986), 128-132.

[O'Brien et al. 87] O'Brien, P.D., Halbert, D.C. and Kilian, M.F. The Trellis programming environment. OOPSLA '87, SIGPLAN Notices, 22(12), (Dec. 1987), 91-102.

[Odijk 87] Odijk, E. A. M. The DOOM system and its applications: A survey of Esprit 415 subproject A, Philips Research Laboratories. Proc. of PARLE(1987), V.1, Lecture Notes in Comp. Sci., #258, Springer-Verlag, 461-479.

[Olderog and Hoare 86] Olderog, E.-R. and Hoare, C.A.R. Specification-oriented semantics for communicating processes. Acta Informatica 23, Springer-Verlag, (1986), 9-66.

[Olsson 86] Olsson, R. A. Issues in distributed programming languages: The evolution of SR. Ph.D. Dissertation, The Univ. of Arizona, (1986).

[Olson 85] Olson, Robert A. Parallel processing in a message-based operating system. IEEE Software, (July 1985), 39-49.

[Olthoff 86] Olthoff, W. G. Augmentation of object-oriented programming be concepts of abstract data type theory: The ModPascal Experience. OOPSLA '86 Proc., SIGPLAN Notices, 21(11), (Nov. 1986), 429-443.

[Padmanabhan 90] Padmanabhan, K. Cube structures for multiprocessors. CACM 33(1), (Jan. 1990), 43-52.

[Padua 79] Padua, D. A. Multiprocessors: Discussions of some theoretical and practical problems. Ph.D. Thesis, Rep. UIUCDCS-R-79-990, Dept. of Comp. Sci., Univ. of Illinois at Urbana-Champaign, (1979).

[Padua et al. 80] Padua, D.A., Kuck, D.J. and Lawrie, D.H. High-speed multiprocessors and compilation techniques. IEEE Trans. on Computers, C-29(9), (Sept.1980), 763-776.

[Padua and Wolfe 86] Padua, David A. and Wolfe, Michael J. Advanced compiler optimizations for supercomputers. Comm. of the ACM, 29(12), (December 1986), 1184-1201.

[Pallas 89] Pallas, Joseph Ira. Multiprocessor Smalltalk: Implementation, Performance, and Analysis. Ph.D. Thesis, Dept. of Computer Science, Stanford University, (STAN-CS-90-1315), (December 1989).

[PCF 88] The Parallel Computing Forum. PCF Fortran: Language Definition. Version 1. Parallel Computing Forum, Kuck & Associates, 1906 Fox Drive, Champaign, IL 61820, (August 16, 1988).

[Perrott 79] Perrott, R. H. A language for array and vector processors. ACM Trans. on Programming Languages and Systems, 1(2), (October 1979), 177-195.

[Perrott 87] Perrott, R. H. Supercomputer Languages: a case study. Proc. of the 2nd International Conference on Supercomputing (ICS '87), Vol.3, (1987), 20-26.

[Peterson 87] Peterson, Gerald E. Tutorial: Object-Oriented Computing. Vol.1: Concepts, Vol.2: Implementations, IEEE (1987).

[Polychronopoulos and Banerjee 87] Polychronopoulos, C. D. and Banerjee, Utpal. Processor allocation for horizontal and vertical parallelism and related speedup bounds. IEEE Trans. on Computers, C-36(4), (April 1987), 410-420.

[Polychronopoulos 87] Polychronopoulos, Constantine. Automatic restructuring of Fortran programs for parallel execution. CSRD Rpt. No.665, Univ. of Illinois at Urbana-Champaign, (May 1987).

[Polychronopoulos 88] Polychronopoulos, Constantine D. Parallel Programming and Compilers. Kluwer Academic Publishers, Norwell, MA 02061, (1988).

[Polychronopoulos 88b] Polychronopoulos, Constantine. Toward auto-scheduling compilers. CSRD Rpt. No.789, Univ. of Illinois at Urbana-Champaign, (May 1988).

[Polychronopoulos 88c] Polychronopoulos, Constantine D. Compiler Optimizations for enhancing parallelism and their impact on architecture design. CSRD Rpt.No.781, Univ. of Illinois at Urbana-Champaign, (August 1988).

[Polychronopoulos 89] Polychronopoulos, Constantine D. Compile-time reduction of interprocessor communication for parallel computers. CSRD Rpt. No.642, Univ. of Illinois at Urbana-Champaign, (January 1989).

[Powell and Miller 83] Powell, M. L. and Miller, B. P. Process migration in DEMOS/MP. Proc. of the 9th ACM Symposium on Operating Systems Principles, (Oct. 1983), 110-119.

[Quinn et al. 88] Quinn, M. J., Hatcher, P. J. and Jourdenais, K. C. Compiling C* programs for a hypercube multicomputer. ACM/SIGPLAN PPEALS, (July 19-21, 1988), SIGPLAN Notices 23(9), (September 1988), 57-65.

[Raj and Levy 89] Raj, R. K. and Levy, H. M. A compositional model for software reuse. ECOOP '89 Proceedings, Cambridge Univ. Press, (1989), 3-24.

[Ranka et al. 88] Ranka, S., Won, Youngju and Sahni, S. Programming a hypercube multicomputer. IEEE Software, pp.69-77, September 1988.

[Rose and Steele 87] Rose, J. R. and Steele, G. L. Jr. C*: An extended C language for data parallel programming. Proceedings of the International Conference on Supercomputing (ICS '87), Vol.2, (1987), 2-16.

[Rosing et al. 88] Rosing, M., Schnabel, R.B. and Weaver, R. Dino: Summary and examples. Proc. of the 3rd Conference on Hypercube Concurrent Computers and Applications, Vol.1, 472-481, Pasadena, CA, Jan. 19-20, 1988.

[Sabot 88] Sabot, Gary. The Paralation Model. Architecture-Independent Parallel Programming. The MIT Press. (1988).

[Sarkar 90] Sarkar, Vivek. Instruction reordering for fork-join parallelism. Proc. of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation. (New York, June 1990), SIGPLAN Notices, 25(6), (June 1990), 322-336.

[Schmidt 86] Schmidt, David A. Denotational Semantics: A Methodology for Language Development. Allyn and Bacon, Inc. 7 Wells Ave., Newton, Massachusetts 02159, (1986).

[Schutz 79] Schutz, H. A. On the design of a language for programming real-time concurrent processes. IEEE Trans. on Soft. Eng., SE-5(3), 248-255, (May 1979).

[Seitz et al. 88] Seitz, C.L., Athas, W.C. and Flaig, C.M. The architecture and programming of the Ametek Series 2010 multicomputer. Proc. of the 3rd Conference on Hypercube Concurrent Computers and Applications, Vol.1, Pasadena, CA, (Jan. 19-20, 1988), 33-36.

[Shapiro and Takeuchi 83] Shapiro, Ehud and Takeuchi, Akikazu. Object-oriented programming in Concurrent Prolog. New Generation Computing (Ohmsha, Ltd. and Springer-Verlag), Vol.1, (1983), 25-48.

[Shapiro 87] Shapiro, Ehud. (Ed.) Concurrent Prolog. Vol. 1 & 2. MIT Press (1987).

[Shapiro 89] Shapiro, Ehud. The family of concurrent logic programming languages. ACM Computing Surveys 21(3), (Sept. 1989), 413-510.

[Shapiro et al. 89] Shapiro, M., Gourhant, Y., Habert, S., Mosseri, L., Ruffin, M. and Valot, C. SOS: An Object-Oriented Operating System - Assessment and Perspectives. Computing Systems, 2(4), (Fall 1989), 287-337.

[Shapiro et al. 89b] Shapiro, Marc., Gautron, P. and Mosseri, L. Persistence and migration for C++ objects. ECOOP '89 Proceedings, Cambridge University Press, (1989), 191-204.

[Shibayama 89] Shibayama, E. How to invent distributed implementation schemes of an object-based concurrent language: A transformational approach. OOPSLA '88 Proc., Sept. 25-30, 1988, San Diego, CA, SIGPLAN Notices 23(11), (Nov. 1988), 297-305.

[SIGPLAN 89] Proc. of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming. San Diego, (Sept. 26-27, 1988), (Edited by G. Agha, P. Wegner and A. Yonezawa.), SIGPLAN Notices, 24(4), (April 1989).

[Smith and Zdonik 87] Smith, K.E. and Zdonik, S.B. Intermedia: A case study of the differences between relational and object-oriented database systems. OOPSLA '87 Proc. SIGPLAN Notices 22(12), (Dec. 1987), 452-465.

[Snyder 86] Snyder, A. Encapsulation and inheritance in object-oriented programming languages. OOPSLA '86 Proc., 38-45, SIGPLAN Notices 21(11), (Nov. 1986).

[Stefik and Bobrow 86] Stefik, M. and Bobrow, D. G. Object-oriented programming: Themes and variations. The AI Magazine, (Winter 1986), 40-62.

[Stroustrup 82] Stroustrup, Bjarne. An experiment with the interchangeability of processes and monitors. Software-Practice and Experience, V.12, (1982), 1011-1025.

[Stroustrup 83] Stroustrup, Bjarne. Adding classes to the C language: An exercise in language evolution. Software-Practice and Experience, V.13, (1983), 139-161.

[Stroustrup 84] Stroustrup, B. Data abstraction in C. AT&T Bell Lab. Tech.Journal, 63(8), (Oct.1984), 1701-1732.

[Stroustrup 85] Stroustrup, B. A set of C++ classes for co-routine style programming. in UNIX System V AT&T C++ Translator Release Notes. #307-175, Part 3, AT&T, (1985).

[Stroustrup 86] Stroustrup, Bjarne. The C++ Programming Language. Addison-Wesley, Reading, MA. (1986).

[Stroustrup 86b] Stroustrup, Bjarne. An Overview of C++. Object-Oriented Programming Workshop, (June 9-13, 1986), SIGPLAN Notices, 21(10), (October 1986), 7-18.

[Stroustrup 87] Stroustrup, Bjarne. The evolution of C++: 1985 to 1987. Proceedings of the USENIX C++ Workshop, Santa Fe, NM, (Nov. 9-10, 1987), 1-21. (Revised and expanded in [Stroustrup 89])

[Stroustrup 87b] Stroustrup, Bjarne. What is Object-Oriented Programming? IEEE Software, (May 1988), 10-20, and appeared in Proc. First European Conf. on Object-Oriented Programming, Springer-Verlag, NewYork, (1987), 51-70, and also in Proceedings of the USENIX C++ Workshop, Santa Fe, NM, (Nov.9-10,1987), 159-180.

[Stroustrup 87c] Stroustrup, Bjarne. Possible Directions for C++. Proceedings of the USENIX C++ Workshop, Santa Fe, NM, (Nov. 9-10, 1987), 399-416.

[Stroustrup 88] Stroustrup, Bjarne. Design Issues in C++. Discussion Session, Workshop on Object-Oriented Programming. ECOOP 1987, Paris. SIGPLAN Notices, 23(1), (January 1988).

[Stroustrup 88b] Stroustrup, Bjarne. Type-safe linkage for C++. (The Journal of the USENIX Association) Computing Systems, 1(4), Univ. of California Press, (Fall 1988), 371-403.

[Stroustrup 89] Stroustrup, Bjarne. Evolution of C++ (The evolution of C++: 1985 to 1989). UNIX System V, AT&T C++ Language System (Release 2.0), Selected Readings (Select Code 307-144), (Ch.2), (1989), and also appeared in USENIX Computing Systems 2(3), (Summer 1989), 191-250.

[Stroustrup 89b] Stroustrup, Bjarne. Multiple Inheritance for C++. UNIX System V, AT&T C++ Language System (Release 2.0), Selected Readings (Select Code 307-144), (Ch.5), (1989). Published also in USENIX Computing Systems, 2(4), (Fall 1989), 367-395. Published also in the proc. of the EUUG Spring Conference, (May 1987).

[Stroustrup 89c] Stroustrup, Bjarne. Parameterized types for C++. USENIX Computing Systems, 2(1), (Winter 1989), 55-85.

[Sullivan 86] Sullivan, B. M. YACC Reference Manual. TR-86-12, The Wang Institute of Graduate Studies, (1986).

[Talia 90] Talia, D. Survey and comparison of PARLOG and Concurrent Prolog. SIGPLAN Notices, 25(1), (Jan. 1990), 33-42.

[Tebra 87] Tebra, Hans. Optimistic And-Parallelism in Prolog. Proc. of PARLE(1987), V.2, Lecture Notes in Computer Science, #259, Springer-Verlag, 420-431.

[Test et al. 87] Test, J.A., Myszewski, M. and Swift, R.C. The Alliant FX/series: A language driven architecture for parallel processing of dusty deck Fortran. Proc. of PARLE(1987), V.1, Lecture Notes in Computer Science, #258, Springer-Verlag, 345-356.

[Treleaven et al. 86] Treleaven, P.C., Refenes, A.N., Lees, K.J. and McCabe, S.C. Computer architectures for artificial intelligence. Treleaven, P. and Vanneschi (Eds.) Proceedings of the Future Parallel Computers, An Advanced Course, Pisa, Italy, (June 9-20, 1986), Lecture Notes in Computer Science, No.272, Springer-Verlag, Germany, (1987), 416-492.

[Tripathi and Berge 89] Tripathi, Anand and Berge, Eric. An implementation of the objected-oriented concurrent programming language SINA. Software-Practice and Experience, 19(3), (March 1989), 235-256.

[Tsujino et al. 84] Tsujino, Y., Ando, M., Araki, T. and Tokura, N. Concurrent C: A programming language for distributed multiprocessor systems. Software-Practice and Experience, 14(11), (November 1984), 1061-1078.

[Tsukakoshi et al. 87] Tsukakoshi, M., Katayama, H., Abe, K. and Yamamoto, K. The supercomputer SX system: Fortran 77/SX and support tools. Proc. of the 2nd International Conference on Supercomputing (ICS '87), Vol.1, (1987), 72-79.

[Ullman 88] Ullman, J. D. Principles of Database and Knowledge-base Systems. (v.1), Computer Science Press, (1988).

[Ungar and Smith 87] Ungar, David and Smith, Randall B. Self: The power of simplicity. OOPSLA '87, SIGPLAN Notices 22(12), (Dec. 1987), 227-242.

[Utter and Pancake 89] Utter, S. and Pancake, C. M. A bibliography of parallel debuggers. SIGPLAN Notices 24(11), (Nov. 1989), 29-42.

[Veen 86] Veen, Arthur H. Dataflow machine architecture. ACM Computing Surveys, 18(4), (December 1986), 365-396.

[Wasserman 90] Wasserman, A.I., Pircher, P.A. and Muller, R.J. The object-oriented structured design notation for software design representation. IEEE Computer 23(3), (March 1990), 50-63.

[Watanabe and Yonezawa 88] Watanabe, T. and Yonezawa, A. Reflection in an object-oriented concurrent language. OOPSLA '88 Proc., Sept. 25-30, 1988, San Diego, CA, SIGPLAN Notices, 23(11), (November 1988), 306-315.

[Wegner 86] Wegner, Peter. Introduction to the special issue of the SIGPLAN Notices on the Object-Oriented Programming Workshop. OOP Workshop, IBM Yorktown Heights, (June 9-13, 1986), SIGPLAN Notices 21(10), (Oct. 1986), 1-6.

[Wegner 86b] Wegner, P. Classification in object-oriented systems. OOP Workshop 86, SIGPLAN Notices 21(10), (Oct. 1986), 173-182.

[Wegner 87] Wegner, P. Dimensions of object-based language design. OOPSLA '87 Proc., SIGPLAN Notices, 22(12), (Dec. 1987), 168-182.

[Wegner 89] Wegner, P. Granularity of modules in object-based concurrent systems. Proc. of ACM SIGPLAN Workshop on Object-Based Concurrent Programming. SIGPLAN Notices 24(4), (April 1989), 46-49.

[Wegner 90] Wegner, P. Concepts and paradigms of object-oriented programming. ACM SIGPLAN OOPS Messenger, 1(1), (August 1990), 7-87.

[Weihl 88] Weihl, W. E. Commutativity-based concurrency control for abstract data types. IEEE Trans. on Computers, 37(12), (Dec. 1988), 1488-1505.

[Weihl 89] Weihl, William E. Local atomicity properties: Modular concurrency control for abstract data types. ACM ToPLaS, 11(2), (April 1989), 249-282.

[Wirth 74] Wirth, Niklaus. On the design of programming languages. Information Processing 74 (IFIP 74), North-Holland Publishing Co., (1974), 386-393.

[Wirth 75] Wirth, N. An assessment of the programming language Pascal. IEEE Trans. on Software Engineering, (June 1975), 192-198, [Reprinted in Horowitz, E. (Ed.) Programming Languages: A Grand Tour (2nd Ed.), Computer Science Press (1985), 137-142].

[Wirth 76] Wirth, Niklaus. Programming languages: what to demand and how to access them, Professor Cleverbyte's visit to heaven. Institut fur informatik, Eidgenossische Technische Hochschule Zurich, #17, (March 1976).

[Wirth 77] Wirth, N. Modula: a language for modular multiprogramming. Software-Practice and Experience, Vol.7, (1977), 3-35.

[Wirth 77b] Wirth, N. The use of Modula. Software- Practice and experience, V.7, (1977), 37-65.

[Wirth 77c] Wirth, N. Design and implementation of Modula. Software-Practice and Experience, V.7, (1977), 67-84.

[Wirth 88] Wirth, N. From Modula to Oberon. Software- Practice and Experience, 18(7), (July 1988), 661-670.

[Wirth 88b] Wirth, N. The programming language Oberon. Software-Practice and Experience, 18(7), (July 1988), 671-690.

[Wirth 89] Wirth, N. Designing a system from scratch. Structured Programming, 10(1), Springer-Verlag, New York, (1989), 10-18.

[Wirth 90] Wirth, Niklaus. Talks in East Central Oklahoma State University, Ada, Oklahoma: "Perspectives on Computer Science Education", and "Language Constructs to Support Program Extensibility", (May 7-8, 1990).

[Wolf 89] Wolf, Wayne. A practical comparison of two object-oriented languages. IEEE Software, (Sept. 1989), 61-68.

[Workshop 88] Report on a Workshop: Critical Research Directions in Programming Languages. Sponsored by the Office of Naval Research, (Oct. 13-14, 1988), SIGPLAN Notices, 24(11), (Nov. 1989), 10-25.

[Yang et al. 89] Yang, C.-Q., Chen, C.-Y. and Stinson, J. E. A distributed environment for executing C++ programs. Proceedings of the 1st Annual IEEE Symposium on Parallel and Distributed Processing, Dallas Texas, (May 22-23, 1989), 188-195.

[Yew 88] Yew, Pen-Chung. Architecture of the Cedar parallel supercomputer. Parallel Systems and Computation, G. Paul and G. S. Almasi (Eds.), Elsevier Science Publishers B.V.(North-Holland), (1988), 137-148.

[Yin et al. 90] Yin, M-L., Bic, L. and Ungerer T. Parallel C++ programming on the Intel iPSC/2 Hypercube. Proc. of the 4th Annual Parallel Processing Symposium., Fullerton, CA, (April 4-6, 1990), 380-394.

[Yokoto and Tokoro 87] Yokoto, Y. and Tokoro, M. Experience and evolution of ConcurrentSmalltalk. OOPSLA '87 Proc. SIGPLAN Notices, 22(12), (Dec. 1987), 406-415.

[Yonezawa and Tokoro 87] Yonezawa, A. and Tokoro, M. Object-Oriented Concurrent Programming. The MIT Press, Cambridge, MA, (1987).

[Zayas 87] Zayas, E.R. Attacking the process migration bottleneck. Proc. of the 11th ACM Symposium on Operating Systems Principles. ACM/SIGOPS 21(5), (1987), 13-24.

[Zimmerman and Crichton 89] Zimmerman, B.A. and Crichton,G.A. An object-oriented environment for the Caltech/JPL Mark III Hypercube. C3P-767, California Institute of Technology, Pasadena, CA 91109, (April 26, 1989).

2

VITA

Chang-Hyun Jo

Candidate for the Degree of

Doctor of Philosophy

Thesis: THE DESIGN AND IMPLEMENTATION OF AN OBJECT-ORIENTED PARALLEL PROGRAMMING LANGUAGE

Major Field: Computer Science

Biographical:

Personal Data: Born in Pusan, Korea, April 25, 1958, the son of Soon-Kyu and Boon-Sun Jo.

Education: Graduated from Myong Ji Senior High School, Seoul, Korea, in February, 1976; received Bachelor of Economics in Statistics from Sung Kyun Kwan University, Seoul, Korea, in February, 1984; received Master of Science degree from Oklahoma State University in July, 1988; completed requirements for the Doctor of Philosophy degree at Oklahoma State University in May, 1991.

Professional Experience: Software Engineer, Electronic Research Lab., Hyo Sung Corp., Seoul, Korea, December, 1983, to April, 1985; Teaching Assistant, Department of Computer Science, Oklahoma State University, August, 1987, to May, 1991.