Automatic Parallelization of C Programs Using Shared Data-Objects

by

Craig Sinclair

B.Comm., University of Calgary, Alberta, 1982
B.Sc., University of Calgary, Alberta, 1984

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in the Department
of
Computer Science

We accept this thesis as conforming
to the required standard

Dr. H. A. Müller

Dr. G. C. Shoja

Dr. K. F. Li

Dr. E. G. Manning

ISBN 0-315-62374-8

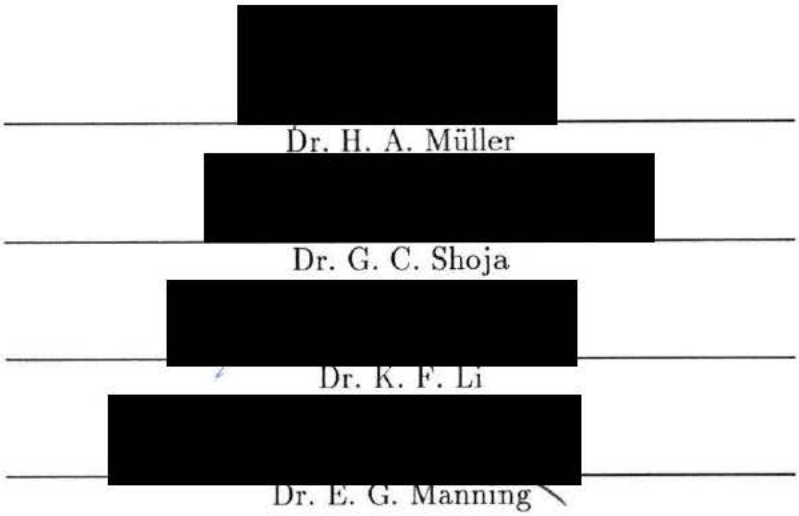
Supervisor: Dr. H. A. Müller

Abstract

The method for exchanging data among processes, when developing programs for parallel execution, has a crucial impact on the design of the programs. In tightly coupled systems programs can directly reference common data through shared memory, whereas in loosely coupled systems programs typically exchange data through message passing. Architectures have recently emerged which integrate the shared variable and message passing paradigms. The most recent addition is the *shared data-object model* proposed by Bal and Tanenbaum.

This thesis presents an *objected-oriented model for distributed programming* based on the framework of the shared data-object model. Our model provides data replication through *shared data-objects* and process management through *process objects*. Sequential programs are partitioned automatically by means of a preprocessor into a set of program blocks that can be executed in parallel. Shared data-objects provide distributed flow control by blocking accesses that violate the ordering specified by the sequential program. A divide-and-conquer algorithm and a ray tracing rendering program are used to illustrate our methodology for distributed programming.

Examiners:



Contents

List of Tables

List of Figures

Acknowledgements

This thesis was made possible by the encouragement and support from family, friends, and my committee members — Hausi Müller, Ali Shoja, and Kin Li. I would like first and foremost to thank my wife Stacey for her emotional support and undying belief in me and in this thesis. Special thanks to my supervisor, Hausi for his support and effort and to Dana, for unselfishly giving up her husband so he could help struggling students.

There is one fellow student that went above and beyond the call of friendship, Brad Richards. Without his countless hours of help, this thesis would not have come together. Thanks also to Brian Corrie and Rob Side who contributed precious time from their theses to lend their expert assistance. Without the initial encouragement from Graham Birtwistle and Jon Muzio, I would not have made the decision to come to Victoria and pursue graduate studies. Thanks also to my family in Calgary and Stacey's family who have supported me and put up with me for the last two years. Also, I would like to thank Jim Uhl and Philipp Heuberger for many thought invoking conversations.

This thesis is dedicated to my son, Graydon, who showed me that life is full of wonder and joy when viewed through innocent eyes.

Chapter 1

Introduction

Hardware has progressed from a single CPU with its own memory to a proliferation of architectural models such as MISD, SIMD, and MIMD. Unfortunately, the development of software for these systems is lagging behind the hardware advances. In the last decade, there has been substantial progress in the use and understanding of software designed for SIMD architectures. However, development of software for the other architectures, especially application software for MIMD systems, has been constrained and limited by the difficult task of writing parallel programs.

Researches have investigated numerous techniques and tools which allow programmers to specify concurrent and parallel programs. In developing their shared data-object model, Bal and Tanenbaum recognized the need for a simpler mechanism for transferring data [BT88]. This thesis extends this concept by enhancing the capabilities and availability of shared data-objects. In addition to communication, objects are also used to manage processes.

1.1 Parallel paradigms

The area of parallel processing provides several paradigms based on some form of message passing or shared memory technique. Recent proposals have incorporated, into a single system, attributes that are usually associated with message passing and attributes that are typically associated with shared memory.

The search for new approaches for implementing programs on MIMD systems, which has been partially responsible for the growing number of paradigms, has been influenced by a variety of different factors such as the underlying hardware, the degree of coupling among processors, and the reliability of the components. This has resulted in an astonishing number of research directions for the development of parallel programs. This thesis is based on ideas put forward in different systems. To clarify the similarities and differences between existing systems a classification scheme, derived from some of the key attributes of this research, is used to highlight the exact area of parallel programming this thesis addresses. The classification is based upon the following criteria.

- Type of hardware
- Type of communication
- Type of synchronization
- Underlying model
- Techniques used to introduce parallelism

A brief description of each criterion follows.

Type of Hardware

Originally, development of concurrent and parallel programs was divided into two groups due to the available hardware configurations — tightly coupled and loosely coupled systems. Both types of development tended to exclusively address the inherent capabilities of that particular type of system. Tightly coupled systems had processors which could share memory. This led to the development of communication methods such as *monitors*. In loosely coupled systems, the processors were usually autonomous entities with no common memory which resulted in longer communication distances (i.e., disjoint memory). This loose connectivity among processors meant that communication and synchronization were generally handled through message passing.

Communication

Current research has lead to different variations on the method of communication. There is no longer a direct relationship between the hardware type and the communication type. Instead, a spectrum of communication techniques has emerged with shared memory at one extreme and message passing at the other extreme [BT88]. This blurring of the original classification criteria is the direct result of new techniques such as implementing shared memory on loosely coupled machines. All of these techniques incorporate features, to various degrees, from both the shared memory and message passing paradigms.

Synchronization

Communication influences the type of synchronization implemented. The message passing approach uses messages to synchronize two processes (synchronously or asynchronously), while the shared memory approach uses constructs such as semaphores

and event counters. There are also special implementations such as *rendezvous* for message passing and pattern-directed blocking for shared memory.

Model

The design of a parallel programming environment is influenced by the conceptual model on which the system is based. The client/server model is the most common and prompted the development of languages like Ada and Distributed Processes (DP) [Han78]. The process model, which consists of cooperating tasks instead of a client/server relationship, encouraged the development of novel forms of addressing which do not require the sender to explicitly state the identity of the designated receiver. Ideas like the tuple space in LINDA allowed processes more freedom in exchanging messages [Gel85]. A variation of the process model, which was designed to simulate real objects such as an electronic light switch, led to processes being defined with object-oriented techniques (e.g., Concurrent Smalltalk [YT87]). In the last decade, research has also focused on models which use a proof system to derive parallel programs. Excellent examples of this type of system are Unity [CM88] and CSP [Whi87].

Parallelism

The final classification criterion is based on how parallelism is incorporated into programs. There are three basic schools of thought on this problem. The first believes that parallelization of programs is too complex a task to be handled automatically and therefore must be performed manually. The second believes that there must be both manual and automatic intervention to develop workable parallel programs. The final school believes that the problem is too complex to be handled by error-ridden programmers and, therefore, must be solved automatically.

This thesis concentrates on the *semi-automatic* parallelization of programs for asynchronous (i.e., nonblocking) message passing environments. The model can be described best as a shared data-driven-object model (data driven because it makes extensive use of data flow) rather than a process model. When implementing programs or algorithms which do not break down into nice concise processes with well-defined interfaces, the main problem is no longer the creation of processes, but ensuring that data is correctly exchanged. For example, global variables, aliasing, and side effects make it difficult to localize the effect of a procedure which will be executed in a separate process. When several processes are executing in parallel, the access to variables must be controlled to ensure the programs computes the correct result.

1.2 Problem

Processes exchanged information through the use of shared memory or message passing. The exchange has traditionally been controlled by semaphores, send/receive primitives and guarded regions. This type of control is not associated with the data being exchanged. For example, a semaphore can be used to ensure a variable is accessed by only one process. The control (i.e., the semaphore) is separate from the exchange (i.e., the variable reference). Often the interdependencies created from variable *conflicts* in programs are so complicated that it is hard enough for a programmer to guarantee that all conflicts are resolved properly [MU89] (conflict defines a relationship where one action affects another). In a complex program it would be simplier to handle the exchange and control in a single construct capable of accessing and controlling the use of the variable, instead of dealing with these issues separately.

The second problem that must be addressed by the shared data-driven-object model is the problem of partitioning programs into parallel segments. Automated

systems have great difficulties in efficiently devising coarse-grained partitions of a program. If a program is not optimally partitioned, the data flow among the distributed parts is so costly — in cycles and time lost due to data exchange — that much of the potential gain from parallelization may be erased. Automatic systems encounter this difficulty when analyzing sequential programs for coarse grain parallelization, because they generally do not have the required information to effectively partition a program and must somehow reconstruct it from the syntactic information. However, the programmer has this type of information and is in a much better position to determine viable alternatives. To allow the programmer to effectively partition a program, a dynamic method for designing and specifying the distributed structure is required.

A program that is to be executed concurrently or in parallel faces problems that are not encountered when executing a sequential program. To effectively support development of parallel programs, the model must provide the means to solve all of the following problems.

- Communication — Exchanging information and maintaining data consistency.
- Partitioning — Partitioning the problem into smaller units of execution.
- Placement — Mapping the solution to the architecture.
- Synchronization — Controlling the execution.
- Mutual Exclusion — Accessing shared resources.
- Exception Handling — Handling distributed errors.

To date, judging by the speed at which code is being written for MIMD architectures compared to SISD architectures, the support has not reached satisfactory levels even

though gains have been achieved. The support will be adequate when the development of parallel software becomes as easy as, and hopefully easier than, writing code for SISD architectures.

1.3 Approach

Conventional languages and their supporting libraries were designed for SISD architectures and without considering the problems of MIMD and distributed systems. Thus, it is not surprising that new languages were developed to tackle the problem. Unfortunately, this resulted in languages that had the necessary enhancements to create parallel code, but did little to ease the complexity introduced by the problems mentioned above which are unique to parallel programs. The problem was with the extensions which needed to be low-level operators to compensate for two things: (1) the lack of powerful parallel constructs; and (2) the inefficiencies within the compilers. This incorporation of low-level extensions defeated the purpose for using high-level languages and made the development of parallel software unnecessarily complicated. It is argued in this thesis that an SISD language can be used effectively to develop parallel programs.

In most cases, it is simpler to write and create a sequential version of a problem before executing it as a parallel program. The programmer can then debug the sequential program before parallel problems such as timing, race conditions, and deadlock complicate the task. Another benefit of writing sequential programs is the independence from the type of architecture on which the program is to execute. Since SISD is the weakest model, programs written for this model can be run on other models such as SIMD, MISD, and MIMD. Also, by using an established language, there is an existing base of programs, knowledge, and tools from which to build new

programs.

For this thesis, C++ was chosen as the SISD language to write parallel programs. C++ has been developed at AT&T [Str86] and is an object-oriented version of the popular programming language C [KR78]. C++ was chosen for its high degree of reusability and powerful data abstraction capabilities. These two attributes lend C++ well to programming problems that are best represented by the shared data-driven-object model.

The shared data-driven-object model is a specialization of the shared data-object model proposed by Bal and Tanenbaum [BT88]. They described a system which uses shared data-objects to communicate among processes. These shared data-objects are a versatile method of exchanging data among parallel segments of a program and are well suited to the complex data interactions of the shared data-driven object model.

The shared data-driven object model is more specific than the shared data-object model, because it uses data-flow information to handle synchronization and to solve the problem of data consistency. To keep the semantics of their model simple, Bal and Tanenbaum decided not to solve this problem. Instead, they left it to the implementation and gave several examples on how the implementation could handle the problems of synchronization and data consistency. By not addressing these problems directly, the shared data-object model addresses a wider range of possible situations than the shared data-driven-object model, which depends on information being available from the data-flow analysis of the program.

Another fundamental difference between this thesis and the shared data-object model lies in the way in which shared data-objects are implemented and the language that is used. Bal and Tanenbaum described a new language (called Orca) which is tailored towards application programming for parallel systems. As mentioned before, a goal of this thesis is to use an SISD language and to show that there is no need to

design and implement yet another specialized language for parallel programming.

This thesis implements shared data-objects using C++ objects. By using an existing sequential language, a homogeneous environment is created for sequential and parallel programming. In addition, benefits are gained from using object-oriented techniques such as encapsulation, inheritance, and polymorphism. Thus, the programmer can reuse all the methods and tools available for developing sequential C++ programs to specify, write, and verify parallel programs.

The additional information required for parallel execution is obtained by running a preprocessor on the sequential program. The preprocessor generates initialization routines for remote processes and supplies objects with the necessary data-flow information to ensure correct execution. Portability and incremental implementation were the main reasons for choosing a preprocessor over a compiler.

The preprocessor is not used to detect parallelism. As mentioned earlier, detecting coarse-grained parallelism is a difficult task and is not attempted by the preprocessor. Current compilers have implemented strategies for vectorizing loops with excellent success. But when the granularity of the parallelism is coarser, the cost of detecting potential parallelism is prohibitive. If and when compiler technology progresses to the point where parallelism can be detected, programs written using shared data-objects will be able to take advantage of this advance, since the programs are not based on language extensions. Currently, optimizing techniques (i.e., vectorization) can be used on the partitions to achieve better performance. Coarse-grained parallelism is not a substitute for fine-grained parallelism but an enhancement.

The block construct found in most imperative programming languages including C, and C++, is used to specify parallelism within a program. A block creates a logical and syntactical separation within a program without changing the semantics. Therefore, the specification of blocks provides a convenient way to describe potential parallelism.

Instead of having the preprocessor find coarse-grained parallelism, the programmer uses blocks to indicate which portions are to be parallelized. By evaluating the underlying algorithm, it is often evident to the programmer which blocks should be executed in parallel. However, the task of implementing the blocks so that they execute correctly in parallel can be tedious and error-prone for the programmer and is best left to a preprocessor.

Since there is a distinct separation between data flow and control flow, processes can be implemented as separate objects. It is difficult for the programmer to detect how blocks (i.e., processes) within a program interact and, thus, ensure that the proper information is exchanged at the correct time. With adequate data-flow analysis the preprocessor can determine the conflicts among blocks. Using this information to generate embedded data-flow constraints, which are implemented using standard C++ syntax, the preprocessor creates a parallel version of the sequential program built upon shared data-objects. The code generated by the preprocessor is compiled using the C++ compiler and linked to the shared data-object library and the process library. The program is executed on top of the existing communication layer, which is REM for this thesis [Sho88]. Shared data-objects provide a homogeneous interface for parallel programming that is independent of the underlying communication system. Thus, any communication system could be used provided that the libraries for the shared data-objects and processes are adjusted.

1.4 Overview

The main thrust of this thesis is to provide a convenient way of writing and implementing parallel programs that can be described in a sequential manner. By using data-flow information from the sequential description, data consistency and

synchronization are automatically achieved without, explicitly being specified by the programmer.

The remainder of this thesis begins with a description of related work. Chapter 3 describes the definitions and usages of shared data-objects and processes. Chapter 4 gives a detailed explanation of the techniques used in the model. It explains how data constraints can be imposed to guarantee the proper partial ordering of data transfers among shared data-objects. Chapter 5 outlines the preprocessor and Chapter 6 features three elaborate examples illustrating the distributed programming method and finally, some conclusions are drawn about the success of this approach for parallel programming.

Chapter 2

Background

The model developed in this thesis was derived from concepts in concurrent and parallel systems. This includes recent work on shared memory and shared data-objects. The model also draws extensively from disciplines that are not directly focused on parallel programming, such as compiler design and object-oriented programming.

2.1 Objects

The simplicity and power of this thesis's model is a direct consequence of using objects to define the major programming components. The central idea of object-oriented programming is to encapsulate associated data and operations into a single entity called an object. This programming style has proven to be a beneficial and surprisingly simple way of programming. Wegner discussed designs for Object-Oriented Languages (OOL) and characterized various attributes which defined the features of Object-Oriented Programming (OOP) including the use of objects, classes, inheritance, polymorphism, concurrency, and distribution [Weg87].

Objects

An object consists of data and procedures that are specifically designed for that object. The procedures are generally called *methods* and perform functions like updating the contents of the data or returning the current state of the data to a query. Methods are invoked by sending the object a message that specifies that particular method. If external information is required by the method then it can be supplied as parameters with the message.

Classes

A class is a template from which objects are instantiated. It defines the data and methods that are present in the object when the object is created. For example, Figure 2.1 defines a class for a pseudo clock. An object instantiated from class `Clock` contains three integers — `seconds`, `minutes`, and `hours` — which are accessed with the methods `Tick()` and `Display()`.

In the program fragment in Figure 2.2, an object `pseudoClock` is created by declaring it of type `Clock`. When `pseudoClock` is instantiated, it is initialized by the constructor `Clock()` which sets `seconds`, `minutes`, and `hours` to zero. The methods `Tick()` and `Display()` are called by passing the object `pseudoClock` the messages `Tick` and `Display` with no arguments. The message `Tick` increments the clock by one second and `Display` prints the current setting.

Inheritance

Inheritance allows the internal design of objects to be divided into separate layers. Thus, common data and methods which appear in more than one class can be stored in a separate layer. This layer can be included in other class definitions through the use of inheritance. In most object-oriented languages, the class which is inherited is

```
1   class Clock {
2      private :
3         int seconds;
4         int minutes;
5         int hours;
6      public :
7         Clock() { seconds = 0;  minutes = 0;  hours = 0; }
8         void Tick() {
9            seconds++;
10           if (seconds == 60) {
11              seconds = 0;
12              minutes++;
13              if (minutes == 60) {
14                 minutes = 0;
15                 hours++;
16                 if (hours == 24) hours = 0;
17              }
18           }
19        }
20        void Display() {
21           cout << "Hours : " << hours;
22           cout << "  Minutes : " << minutes;
23           cout << "  Seconds : " << seconds << "\n";
24        }
25  };
```

Figure 2.1: Definition of the class clock

```
        ⋮

1       Clock pseudoClock;
2
3       pseudoClock.Tick();
4       pseudoClock.Display();

        ⋮
```

Figure 2.2: Instance of the class clock

called the super-class whereas the class which does the inheriting is called the sub-class. An extension to single inheritance is multiple inheritance which means a class can inherit more than one super-class. In Figure 2.3 `Window` and `Object` are defined as base classes while `BinaryNode` is a derived class which inherits the class `Object`. The `WindowNode` class directly inherits both the `BinaryNode` class and the `Window` class; in addition, it indirectly inherits the `Object` class through the `BinaryNode` class. Thus, when an object of the `WindowNode` class is created, it includes the data and methods defined in the `WindowNode` class, the `Window` class, the `BinaryNode` class, and the `Object` class.

Polymorphism

A polymorphic construct (e.g., an item which references an object) has the ability to refer to different classes. In a strongly typed language such as C++ unconstrained polymorphism is not desirable. In C++ if a construct is typed to a particular class, it can refer to objects of that class and all derived sub-classes [Mey88, Str86]. If the class which the construct is typed to is a derived class, then the construct cannot refer to an object created from a super-class.

```
        ⋮

1   class Object : {
2         ... // data for Object
3     public :
4         ... // methods for Object
5   }
6
7   class BinaryNode : public Object {
8         ... // data for BinaryNode
9     public :
10        ... // methods for BinaryNode
11  }
12
13  class Window : {
14        ...// data for Window
15    public :
16        ...// methods for Window
17  }
18
19  class WindowNode : public BinaryNode, public Window  {
20        ... // data for WindowNode
21    public :
22        ... // methods for WindowNode
23  }

        ⋮
```

Figure 2.3: Class definition using multiple inheritance

Concurrent execution

To specify concurrent execution of objects, implementations have either provided explicit facilities within the language or supplied super classes which allow objects to inherit the necessary system capabilities. Objects which control concurrent execution are called *processes* and have the ability to create multiple threads of control [Weg87]. A thread is a locus of control (i.e., a virtual processor) — instructions and the necessary data structures in which to store state information. The amount of concurrency within a *process* is determined by two factors — the number of threads in existence and the number of active threads. To have true concurrent execution, there must be more than one active thread. If a *process* can create more than one thread, but only one thread is active at any given time, it is said to execute quasi-concurrently. If a *process* has only a single thread, it is executes sequentially. In this thesis light-weight threads are referred to as *light-weight processes* because they execute within the same address space but not necessarily on the same processor. The process which is created, or spawned, is called the child, while the process which created the child is called the parent.

Distributed execution

Distributed execution occurs when processes occupy different address spaces and communicate through message passing. Processes which execute in a distributed manner are called distributed processes or heavy-weight threads, because they occupy separate address spaces. Distributed processes are characterized by the amount of concurrency, the type of interaction, and the method of interconnection. Concurrency is determined by the number of light-weight threads which can be concurrently active. Interconnections can be dynamically created at run time, or statically at compile time. The term, heavy-weight process, is used to refer to heavy-weight threads that

can be created on any processor available. Parent heavy-weight processes are said to create children heavy-weight processes.

2.2 The development of concurrent and parallel programming

Concurrent and parallel systems need to provide a simple and effective method to describe and implement programs. Over the history of such systems several concepts and constructs have been introduced to address this problem. These can be divided into two main areas, which address the issues of process management and communication within a program.

In the past, the difficulties of implementing distributed systems often overshadowed the needs of the programmer. In recent systems however, there has been an effort to improve this situation and some of the following deficiencies have been successfully addressed by various systems.

Data Flow

- Simple specification for data exchange.
- Simple introduction of synchronization and mutual exclusion primitives to guarantee and maintain the proper ordering on data exchanges.
- Syntactic consistency within the language (i.e., the same as the sequential syntax).
- Different exchange patterns (1-1, 1-n, n-1, n-n) which are implicitly or explicitly stated.
- Compatible with various communication paradigms.

Control Flow

- Simple specification for process creation, execution, and termination.

- Simple introduction of synchronization, deadlock avoidance, and fairness to ensure correct execution between processes.
- Syntactic consistency within the language (i.e., the same as the sequential syntax).
- Different connection patterns (1-1, 1-n, n-1, n-n), are implicitly or explicitly stated.
- Compatible with various communication paradigms.

To determine the attributes to be included in the model developed in this thesis, existing systems for concurrent and parallel programming were examined. This section roughly parallels the evolution of concurrent and parallel programming and highlights the attributes that had a significant impact on the direction of this thesis.

Forks, locks, and semaphores

A common technique used to generate concurrent processes is the *fork* and *join* paradigm. After the *fork*, both processes continue executing accessing the same resources. This creates a problem when the processes need to coordinate on some activity or exchange information at a specified point. One way to solve this problem is to allow controlled access to common data. Locking is used to provide a single process exclusive access to the shared data. The notion of a lock has been embedded into many operating systems through the incorporation of semaphores which allowed processes to block on a *P(S)* and to continue executing when a matching *V(S)* is given. Semaphores have been so successful that advances in concurrency control seem obligated to compare themselves against semaphores. However, there is a serious drawback in using locks and semaphores. Coordinating the access to data and the synchronization of processes is left to programmer. Heavy use of shared data and semaphores leads to complex interdependencies among processes which can make the implementation a pain.

Monitors and messages

Monitors and message passing are used to alleviate the problem of data coordination and synchronization.

Monitors encapsulate data and procedures into passive entities and are designed to provide mutually exclusive access to data [Hoa74]. This is achieved by allowing only one process at a time to enter a monitor. If more than one process requests access, they are placed in a first-in first-out (FIFO) queue. Synchronization is accomplished using a conditional variable defined within the monitor. A conditional variable has two defined operations, *wait* which blocks a process, and *signal*, which unblocks a process. Because the conditional variable is defined inside the monitor, the blocked process must give up its exclusive access rights before another process can issue a *signal* operation.

Message passing further abstracts the idea of data coordination and synchronization. Unlike monitors, which are designed for shared memory, message passing makes no assumption about the underlying architecture. The notion of an *address space* which is machine dependent is replaced by a *name space* [Whi87] which is system independent. Thus, in most message passing systems data is not accessed according to a physical location in memory. However, systems incorporating shared memory features on top of message passing have recently emerged. The distinction between message passing and shared memory has started to blur [BT88, DoD83, Che85, Li86]

Messages are not only used to exchange information, but also to synchronize processes. There are two types of message protocols — synchronous and asynchronous — to support different types of synchronization. Synchronous message-passing blocks the sender and the receiver until the message is exchanged. In asynchronous communication, the sender is not blocked and continues once the message is released to the communication system. However, the receiver of the message can be treated differ-

ently from the sender. It can become blocked on a receive request until the message is obtained. Even though messages are a simple and yet powerful way to handle concurrent or parallel processes, they do not provide adequate support for development. This results in the programmer being responsible for the tedious work of collecting and sending messages. Sequential algorithms like a divide-and-conquer sort are not easily adapted to message passing.

Coupling

The type of communication between processes used to be based on how tightly coupled the processors were, but this is no longer true. Light-weight processes now execute on any system where communication is through common memory (real or apparent). Heavy-weight processes can execute on tightly or loosely coupled systems where processors do not share memory and communicate through messages.

Shared data

In a recent paper, Bal and Tanenbaum looked closely at different implementations which are based on shared data (cf. Section 2.3). They observed that there was no longer a clear classification of systems based upon message passing and shared memory [BT88]. By focusing on how data is shared, rather than on how a process controls data exchange, Bal and Tanenbaum presented a new view of concurrent and parallel programming. Their discussion is similar to a survey done by Whiddet [Whi87], but there are subtle differences. When evaluating the semantics and synchronization of different systems, Whiddet kept data and control strictly separated. He uses control and primitive constructs to isolate data instead of data being the focus of control.

The syntactic specification of shared data is straightforward, because it uses an addressing scheme similar to the one for sequential variables. Data is exchanged by

having two processes simply reference the same variable. For sequential languages, shared data provides a syntactically consistent method for accessing distributed data, since shared data can use the same rules which govern the creation and scope of variables. Different communication patterns are achieved by varying the processes which can reference the shared data. For example, a 1-n relationship is accomplished by having n processes which can reference the same shared data.

Controlling the access to shared data is difficult. Having n processes trying to access the same piece of data causes problems similar to those encountered with semaphores. As mentioned before, maintaining proper synchronization and mutual exclusion is extremely difficult as the complexity of the data exchange increases. This is further complicated by problems encountered in distributed systems like communication delays and unreliable transfers.

Since shared data is a new approach for distributed systems, there is limited experience in developing actual parallel programs. However, there are systems that include mechanisms which make it seem as if shared data exist. The first few systems discussed below are designed using message passing, but incorporate some form of shared data. The last few systems are designed explicitly with data sharing in mind. Experience with these systems suggests that shared data can meet not only the concerns stated for data exchange, but also those for process control.

```
Let G be a gate name.
Process A:  G ! 6
Process B:  G ? X:int (X mod 2 = 0)
Process C:  G ? X:int (X mod 3 = 0)
```

Figure 2.4: Three-way *rendezvous* of processes A, B, and C

CSP-like languages and communication ports

Communication in CSP-like languages provides synchronous methods of communications where the sender must explicitly state the designated receiver. This form of addressing is generally too restrictive. Therefore, a mechanism like a gate is used to allow processes to interact without knowing the name of the other party. In Figure 2.4, LOTOS (a CSP-like language) is used to specify a program where three processes must exchange information through a gate G (which is implemented as a shared FIFO queue in LOTOS) [Vuo89]. Data is only exchanged among the processes when a *rendezvous* is reached. This system is based on message passing and uses queues (i.e., gates) as an abstract data construct. This is similar to using shared data for exchanging information among processes.

Ada

Ada, which is based on an operation-oriented model, allows data to be visible to several processes [DoD83]. By using a shared variable, separate Ada tasks can access data according to the Ada scope rules. To allow tasks to execute on separate processors, the data is *replicated* on every processor. To enforce mutual exclusion on accesses, Ada restricts updates to the points when tasks perform a *rendezvous.* Unfortunately, this greatly reduces the shared variables functionality and their use is

generally avoided.

Object model

Objects can be used to represent light- or heavy-weight processes. If processes are defined in this manner, they can be created and referenced as if they are local objects. Since data is also represented as objects in these models, a program can handle data and processes in a uniform manner.

Problem-oriented shared memory

Cheriton implemented shared memory as a distributed service which could be tailored to a specific problem [Che85]. Instead of keeping a single version of the data, the data is replicated over distributed processors. The semantics are relaxed so that the replications are not guaranteed to have the latest update. This simplifies the implementation, because no atomic updates on the replicated data need to occur. Stale data can be handled by letting the programmer worry about it or by guaranteeing some degree of liveness. In this thesis the semantics for sharing data can be relaxed by using information generated from data flow.

The Agora shared memory

The Agora system allows processes written in different languages to access the same shared data. Access is through an extendible set of functions which is available to all programming languages supported by the system. Data are immutable elements that are accessed through a map. When data is changed, a new data element is created and a new entry is placed in the map. Garbage collection is used to recover data elements that are no longer needed.

Tuple space

Global memory is used to store tuples in LINDA — a tuple is a collection of data. Tuples are placed in and retrieved from a tuple space (global memory) by the following operations:

- read — reads an existing tuple.
- out — places a tuple into tuple space.
- in — reads and deletes a tuple.

The unique aspect of this system is the way tuples are referenced; they are not addressed by a location but by their contents [Gel85, Ahu86]. This type of addressing allows different patterns of communication to coexist. Tuples are also used to synchronize processes by blocking them on a `read` or `in` operation. The process is unblocked and allowed to continue when a suitable tuple is found. Updates on tuples are synchronized, because a tuple must first be removed from the global tuple space before it can be changed. Therefore, it cannot be accessed by another process until it is placed back into the tuple space.

Shared memory

In Kai Li's system, shared memory is addressed in the same manner as virtual memory [Li86]. Several processors may have a copy of a page of memory, but only one processor has read-write permission. When a write is to be performed on a particular page, a write page-fault interrupt occurs. All processors with that page are told to invalidate their copies. If the processor which had the page-fault does not have a valid copy of the page, then one is obtained from another processor. If any other processor has read-write permission when the fault occurs, it changes the permission to read-only, and

sends the requesting processor its valid copy of the page. This implementation allows a concurrent style of execution on distributed machines. However, if the program is incorrectly designed, thrashing can occur when separate processes are reading and writing to the same page causing an excessive number of alternating page-faults to occur.

Shared logical variables

In concurrent implementations of Prolog, communication is achieved through shared logic variables, which are used as communication channels [CG86, Gre87]. Unlike variables in the sequential versions of Prolog, shared logic variables can only be bound once, eliminating the possibility of a variable being unbound through backtracking. Synchronization is implemented in a data-flow fashion — processes are blocked when a variable is referenced which has not been bound yet. Since shared variables are treated like Prolog variables and are assigned through unification, the semantics for unification for the sequential and parallel versions are similar.

2.3 Sharing data through objects

In the paper "Distributed Programming with Shared Data," Bal and Tanenbaum discussed a new model based on shared data. This model uses objects, called data-objects, to distribute information across processors. They are passive entities used to encapsulate data which are only accessible through a set of indivisible operations specified for the data's type (i.e., class). Accessing the data through these operations is conceptually safer than accessing variables which are shared among processes.

When a process creates a child, objects can be shared among the processes by passing the objects as parameters. Any change to a data object is then visible to

all processes which share that object. Thus, the data objects act as communication channels among processes.

Data objects are under the control of the operating system. They are implemented in a distributed manner using selective replication and migration. Read operations are performed on a local copy of the object. After write operations are performed, they are propagated to all replicates. If simultaneous updates occur, they are serialized on a primary copy. Access statistics on objects are maintained to allow run-time optimization. If a process continually updates an object, it is advantageous to migrate the primary copy to that process.

The idea of replicating data is used to decrease the access time, which makes consulting a single copy preferable to consulting several processors. However, this creates a problem with maintaining consistency among replicates due to possible lag time in transmitting and receiving data. One solution is to allow inconsistencies to exist in the same way they exist in Agora's shared memory technique (cf. Section 2.2). A second alternative is to have immutable objects such as LINDA tuples. A third approach is to use a scheme of validation/invalidation similar to the one proposed by Kai Li for shared memory [Li86].

In Bal and Tanenbaum's system, the problem of consistency must be explicitly solved by the implementation. The solutions proposed for the implementations are dependent on the communication architecture. Three architecture classes are considered: point-to-point, reliable multicast, and unreliable multicast. The point-to-point implementation uses a two-phase locking protocol which is implemented at each processor by a process manager. The manger dynamically creates light-weight threads to handle multiple write requests to objects. The use of light-weight threads ensures that deadlock will not occur. Consistency is guaranteed by the two-phase locking which locks out all processes when an update is performed. After the update is com-

pleted the changes are sent to all replicates and the lock is released. The problem of multiple write operations is solved by serializing the writes on a primary copy.

The multi-cast implementation is for asynchronously parallel processes that do not require the strict temporal ordering imposed by the point-to-point implementation. As discussed by Lamport, processes are partially ordered at the point where messages are exchanged [Lam78]. By relaxing the semantics of total ordering, processes may use old data from a local shared data object after a remote process has written to its copy. In Figure 2.5 for example, Process 1 reads Object X after Process 2 writes to it. However, when Process 1 reads Object X, it does not obtain the new value written by Process 2. This program executes correctly according to the partial ordering, because the message to update the variable is not received until the read is complete.

For programs where partial ordering is sufficient, the use of reliable multicast guarantees that all processes receive a message in the proper order. This simplifies the task of the process manager, since all replicated copies can be updated with one multicast. The manager does not have to send an individual copy to each replicate.

The last implementation Bal and Tanenbaum consider is the unreliable multicast. They consider this system because of the problems in achieving reliable multicasts on current distributed systems. The algorithm they use is basically the same as in the reliable multicast, except that the manager must decide if the messages received are in the correct order. This can be done by sending a vector containing the current number of messages received along with the message. If a message is lost, it is indicated by a disparity between the vector at the processor and the one received with the message. When this happens, point-to-point transmission can be used to obtain the appropriate message.

They also describe a language called *Orca* which is based on shared data-objects for writing applications. Unfortunately, the objects described in the language do not

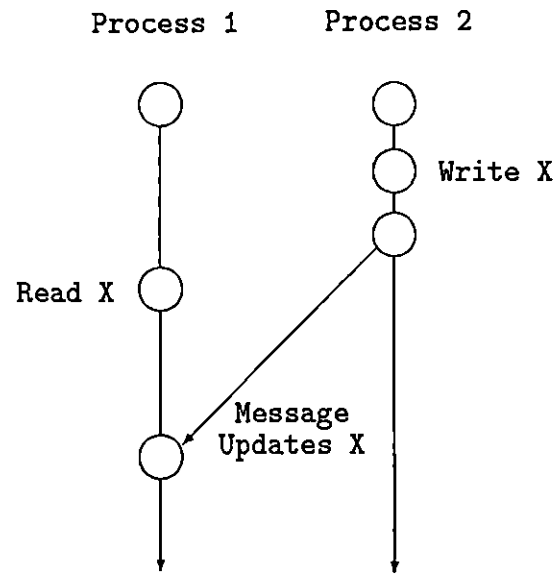


Figure 2.5: Read/write time line

include concepts such as inheritance or polymorphism. However, the language has three noteworthy properties — processes communicate indirectly through objects, access to data objects is synchronized through the use of indivisible operations specified by the object's type, and process synchronization is implemented through the use of guarded commands which suspends a process until one of the guards (a side-effect free boolean expression) becomes true.

2.4 REM

REM (Remote Execution Manager) was designed to support distributed execution and load balancing in a network of workstations [SCT87, Sho88]. It provides applications with the ability to create distributed processes on idle workstations and to establish communication channels among parent and children processes. REM is used by the shared data-driven object model to create heavy-weight processes and to provide communication among the processes.

REM runs as an application on top of UNIX. The interprocess communication is based on the **TCP/IP** and **SOCK_STREAM** protocol. Each workstation used for executing REM-based processes starts a network module which is responsible for all communication among workstations. This module is connected to a local module which handles communication among processes on the same workstation. When an application process starts up, it connects to the local module; through this connection it is able to create and communicate with remote processes.

The interface presented by REM consists of a simple yet powerful set of operations. Children are created by supplying the name of a file which contains the executable used to create the heavy-weight process to the local module. When the child is created, a channel is established between the parent and the child. Communication

over the channel is handled in an asynchronous manner by providing REM with an asynchronous handler. This handler is invoked when a signal is received from a remote socket, indicating that a message is pending. Other advantages that are gained by using REM are fault tolerance, I/O redirection, and load-sharing capabilities.

2.5 Computational models

The terms which were mentioned in the introduction — SISD, SIMD, MISD, and MIMD — identify four separate computer models. They differ as to whether computational units in the model can handle single or multiple streams of data and whether multiple units execute the same or different instructions. The acronyms stand for [Akl89].

- SISD — **S**ingle **I**nstruction stream, **S**ingle **D**ata stream.
- SIMD — **S**ingle **I**nstruction stream, **M**ultiple **D**ata streams.
- MISD — **M**ultiple **I**nstruction streams, **S**ingle **D**ata stream.
- MIMD — **M**ultiple **I**nstruction streams, **M**ultiple **D**ata streams.

The SISD model provides of a single computational unit which works on one stream of data. The underlying architecture of this model is often referred to as the von Neumann architecture. The programs which execute on the SISD model are called serial or sequential programs.

The SIMD model consists of a series of identical computational units, which receive the same instruction at every step of the computation. The data used by the computational units can be unique. If the data is not unique the accessibility of the data to units leads to the following refinement of the SIMD model.

- EREW — Exclusive Read, Exclusive Write
- CREW — Concurrent Read, Exclusive Write
- ERCW — Exclusive Read, Concurrent Write
- CRCW — Concurrent Read, Concurrent Write

In the EREW SIMD model, only one processor is allowed to read and write to a particular piece of data, while in the CRCW model more then one processor is allowed access to the same data.

The MISD model has a series of processors executing different instructions on the same data. This model is not suited to most applications, but there are special situations that lend themselves well to this type of computation. For example, a piece of equipment generates a single piece of data that is acted upon by several processors performing uniques tasks.

The MIMD model is the most general and powerful of the four models. Each processor can execute a different set of instructions on potentially unique sets of data. The interconnection, among the units of computation are through shared memory (tightly coupled), or through a network (loosely coupled). Configurations like distributed systems and networks can be considered part of the MIMD model. Since units are allowed to access the same data, the model can be further refined based on the access rights to the shared data. This refinement is identical to the one mentioned for the SIMD model.

2.6 Data-flow analysis of sequential programs

To effectively divide sequential programs into subtasks, data analysis needs to be performed to determine the interconnections among the subtasks. With this infor-

mation, one can guarantee proper data transfer among the partitions (i.e., the same ordering as for the sequential version). If there are no interconnections, or relatively few among a set of subtasks, then these subtasks are prime candidates for parallel execution. If the program is effectively partitioned into subtasks, it will benefit from the parallelization. Interconnections are found by examining the variables which are used and set in the subtasks.

The set of all variables which are accessed can be obtained by using static data-flow analysis techniques; a simple but effective procedure was proposed by Banning [Ban79]. In his paper, Banning dealt with the problem of determining how the execution of a program segment affects its variables. He showed that variables can be changed in two fashions — directly through a referenced parameter used in a module call or through the occurrence of a side effect.

When a variable is passed by reference to a module, there is a corresponding parameter in the module definition which refers to the same location in memory as the variable. Parameters in the module definition are mapped positionally to the variables supplied in the module call. Within the scope of the called module, the parameter name provides direct access to the variable. Therefore, to determine when the variable is changed it is simply a matter of determining when the parameter is changed.

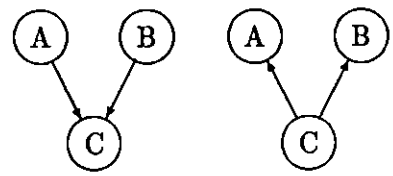
Another way of changing a variable is through a side effect. Unlike the previous case, the variable is not passed to the module as a referenced variable. Instead, the variable is visible to both the calling and called modules. Therefore, the called module can change the variable by directly referring to the variable's name. To determine when a side effect occurs, it is necessary to determine which updated variables in the called module are visible to the calling module.

So far, only a single module call has been considered, ignoring the possibility of

nested module calls. The effect a module call can generate is not restricted to the changes that occur directly in that module. Changes can also occur from a subsequent call or from a chain of calls which is anchored in the original module. This chain is called a *call-chain* and represents a sequence of module calls.

Since a call chain can be unbounded as in the case of recursive modules, Banning uses an activation chain. An activation chain represents the chain of module instances which occur during the actual execution of a program. When an instance starts up, all local variables and parameters passed by value are allocated space. Referenced parameters and global variables point to memory locations that have previously been allocated and are external to the instance of the module. Since the calling module is only concerned with the externally visible effects created by the called module, local variables and parameters passed by value are ignored. This means, only the changes seen in the global variables and referenced parameters are considered as one activation leads to another. Since a module can be called several times with different variables being passed to the reference parameters, the changes are specific to a particular call and not to a particular module.

Banning showed that an activation chain can be used to represent all possible call chains, and he showed that an activation chain can be represented in a manner that does not exhibit the unbounded nature of call-chains. He did this by mapping the activation chain to a directed graph, where every node in the graph represents a module, and all activations of the same module in the activation chain are mapped to this one node. Having a single node to represent every activation of a module is possible, since activations do not differ when the internal data of the module is ignored. The directed arcs in the graph represent specific calls to the modules, but point in the opposite direction to allow modified variables to be propagated back to the caller.



Call Graph Activation Graph

Figure 2.6: Activation graph

The external effect of a call is determined by tagging the directed arc with a function that indicates which variables were accessed during that call. It does this by returning changes caused by referenced parameters and side effects. In Figure 2.6 for example, FuncA and FuncB both call FuncC and therefore there is a directed arc from FuncC to FuncA and an arc from FuncC to FuncB. The arcs in the activation graph are tagged with a function that represents the side effects and the changes to the variables sent as referenced parameters.

To determine which variables are updated from the activation graph, it is necessary to iterate over all nodes in the graph propagating the updated variables along the directed arcs. If enough iterations are performed, the updated sets for each module will ultimately stabilize. At this point each node (i.e., module) contains the union of all the updated variables generated from outbound arcs. The iteration is required to resolve recursive activation paths; if there are no circular paths, the updated variables can be generated by a single pass through the graph.

The process of determining when a called module updates or uses a variable becomes complicated when two types of references occur. The first situation is when more than one parameter refers to the same variable (i.e., the same location in mem-

ory). The other type of reference is when a parameter and a global variable are pointing to the same location in memory. If two names are active within a module and reference the same memory location, then they are called aliases. To detect whether a variable is changed or used within a module, all possible aliases must be taken into consideration.

By doing static analysis it is possible to generate two sets of variables, one containing USED variables (i.e., variables that are read in a module), and the other containing MODified variables (i.e., variables that are written to in the module), taking into account procedure calls and aliases. The interconnection between two statements or groups of statements can be seen by comparing the MOD and USED sets. There are conflicts among statements, or blocks, if there is a nonempty set created from the intersection of the MOD and USED sets. Optimizations can be performed to minimize the occurrence of conflict such as eliminating variables from the USED set if they are set before they are used

2.7 Software tools for parallel development

In recent years, many researchers have investigated tools which aid in the development of parallel applications. This has been in response to the difficulties encountered when programming parallel applications. Programmers must take into account the logistical problems of parallel systems such as partitioning and placement while implementing the program. Tools are especially adept to handling the type of situation shown in the Figure 2.7 where the implementation constantly changes. Tools may remove the repetitious elements from each iteration, while leaving the important design decisions to the programmer.

Hypertool is a recent example of a tool that develops parallel applications using a

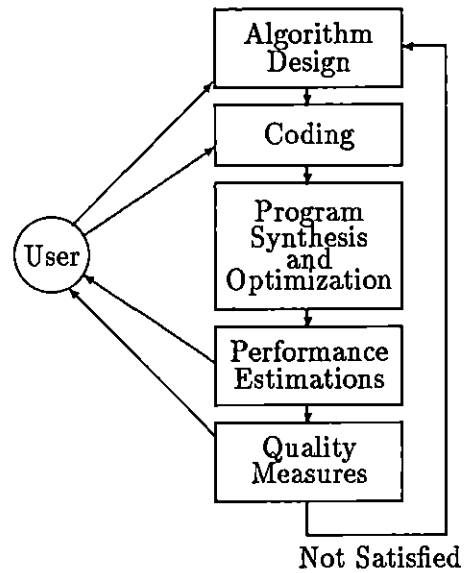


Figure 2.7: HyperTool's design methodology [WG89]

sequential language [WG89]. A program is partitioned into processes (which are the basic units of computation). Several processes are combined into a task by a scheduler using an algorithm for the quadratic assignment problem. Each task is assigned to a particular processor. It is the scheduler's job to take care of dependencies and to minimize the overhead by properly grouping processes together. The assignment of tasks to processors is done by a critical path heuristic which minimizes the distance among tasks according to the level of dependency (i.e., cost).

There are two other noteworthy features exhibited in this tool. First, the model uses information generated from an analysis of a macro data-flow graph to determine synchronization. The graph consists of nodes which represent the processes and the arcs represent the data dependencies among the processes. Second, the tool generates performance estimates which are used by the programmer to rework the parallel solution to obtain a better partition. This is an excellent example of an iterative approach to parallel programming.

Chapter 3

Distributed shared data and process objects

Object-oriented programming has its roots in the Sixties with the design of the class construct in the programming language Simula 67. In the Eighties numerous object-oriented programming languages have emerged. As a result, many programming techniques including distributed programming are being re-evaluated and redesigned to fit the object-oriented framework.

This thesis investigates two types of objects: *shared data-objects*, which extends the concept of a monitor; and *process objects*, which is an object representation of heavy-weight threads. Two types of objects are used to separate the distributed control flow of a program from the data flow. The control flow is managed by process objects (cf. Section 2.1) which provide methods for dynamically creating remote child processes and representing them as locally defined objects. Data flow is handled by shared data-objects (cf. Section 2.3) which provides a natural means of establishing communication among processes. By using objects in this manner, a versatile and

```
                ⋮

1       int current_count;
2       current_count = 0;
3
4       while (current_count != 10) {
5          current_count = current_count + 1;
6          cout << "Current Count = ";
7          cout << current_count.Access() << "\n";
8       }

                ⋮
```

Figure 3.1: Sequential successor function

uniform environment can be provided for developing parallel programs.

3.1 An introductory example

The C++ program fragment depicted in Figure 3.1 implements a simple successor function. The intent of the program is to increment a counter and display the current value. The counter is declared as an integer, set to zero, and incremented ten times inside a loop.

A parallel program which performs the same function as the simple successor function, based on the shared data-driven model, can be written in much the same fashion as the original sequential program. The integer `current_count` is changed to a shared data object by replacing the original declaration with a declaration of a `RemoteInt`. The `RemoteInt` class stores an integer value which is assigned a value by the method `Set(int)` and referenced by the method `Access()`. Being derived from

the shared data-object class, the `RemoteInt` can be accessed by several processes at the same time. The new program, shown in Figure 3.2 declares `current_count` as a `RemoteInt` on line 1. Instances where `current_count` are used as an *lvalue* in the original program are modified by replacing the assignment operator with a call to the method `Set()`. For example, in line 2 of the new program, `current_count` is set to zero with method `Set()`. When `current_count` is used as an *rvalue*, the reference to the integer is replaced by a call to the method `Access()`. In line 3, the conditional operation which checks the current value of `current_count` is modified to use the method `Access()`.

A child heavy-weight process is automatically created when an `InlineProcess` object is encountered and terminated when the object is deleted. The process can be created as a blocking or nonblocking process, which determines whether the parent heavy-weight process waits or not, for the child to finish. In the new program, the `InlineProcess` object `counter` represents the child. It is set up as a `BLOCKING` process to ensure that the child completes its task of incrementing `current_count` before the parent can delete it.

When the parent executes, it loops over the declaration of `counter`. Each time it declares this object, a new child is created. When this child starts up, it initializes all of its active shared data-objects (i.e., `current_count`) to the current value in the parent process. The child then executes line 6 which increments the `current_count`. Upon completion, the replicate of `current_count` in the child sends the new value back to the `current_count` located in the parent process.

The above analysis of the distributed execution illustrates the mechanics of our distributed programming methodology. Given a preprocessor, to understand the program it is typically sufficient to read the parallel version as if it was a sequential program.

```
                ⋮

1         RemoteInt current_count;
2         current_count.Set(0);
3         while (current_count.Access() != 10) {
4            {
5               InlineProcess counter("counter");
6               current_count.Set(current_count.Access() + 1);
7            }
8            cout << "Current Count = ";
9            cout << current_count.Access() << "\n";
10        }

                ⋮
```

Figure 3.2: Distributed successor

3.2 Process objects

In parallel programming, there must be a method for specifying when a process is to be created and when it is to be terminated. Objects provide a convenient method of representing and controlling resources that are external to the program. In fact, the creation and destruction of the external resource can be directly controlled by the creation and destruction of the object. In this manner, objects which are defined within a program provide a convenient method of representing children (heavy-weight processes). A child is created when a process object is defined, and is killed when the object ceases to exist.

When a child is created by a parent process, a communication channel is established between the two processes. This channel allows shared data-objects to pass information among its replicates. The process object sending the information takes

a packet of data received from a locally defined shared data-object, and sends it through the appropriate channel which is connected to the process specified by the shared data-object. The receiving process takes the packet and passes it along to the local replicate of the original shared data-object. With this method of transmitting data, processes are not required to understand the nature of the data being sent. Only a simple system for communication is needed which can send and receive blocks of data addressed to specific shared data-objects.

The process object is not required to regulate the execution pace of the child, because it is handled by the shared data-objects. Therefore, process objects do not explicitly provide synchronization, handle deadlock, or ensure fairness. This allows the class description for process objects to be succinct, because process objects provide only basic process control and communication services.

3.2.1 Process object description

Two types of process objects are provided to allow children to have different degrees of permanence. When processes are defined as objects, they abide by the C++ scope rules. This means, process objects are destroyed when the program leaves the scope in which the object is defined. The first type of process object creates a child heavy-weight process whose life span is the same as the object representing it in the parent heavy-weight process. When the object is destroyed the child is also destroyed. The second type of process object creates children which are permanent. With this type of process object the child is not destroyed when the process object is destroyed.

The provision of having two types of objects provides freedom in the way children are handled. There are cases where the remote child only needs to be executed once. Such a situation occurs when the child executes an inline portion of the program or

when the problem is recursive. For the latter example, a new child must be created for each recursive call and the child must die when the call terminates.

There are other cases where it is preferable not to destroy the child along with the process object. If a process object is defined inside a `for()` loop in C++, the child is created when the parent process enters the loop. Unfortunately, it is destroyed when the control returns to the beginning of the loop, because the current scope terminates. When several child need to be defined inside a loop it is better to kill the children when the shared data-objects used in the child are destroyed and not the process object.

There is a second reason for allowing a child process to continue. When the child is spawned, a heavy-weight process is created on a processor. For this to occur, there must be a substantial amount of work done by the communication system and by the operating system on that processor. The overhead associated with the creation of the child can be avoided if the process is reused.

Inline Processes

The `InlineProcess` class (cf. Figure 3.3) implements a process object which tightly ties the existence of the child to the life of the object defined in the parent process. The child only exists as long as the object is defined. When the object is destroyed, explicitly with the `delete` statement or implicitly when the current scope ends, the parent process destroys the remote child.

The `InlineProcess` object can be defined as a `BLOCKING` process. When the parent tries to destroy the object, it stays blocked until the child completes. The `InlineProcess` can also be defined as `NONBLOCKING`, which means that the parent does not wait for the child to complete, but destroys the object and hence the child.

When an `InlineProcess` object is defined and the child starts, it inherits the

```
                ⋮

1  class InlineProcess : private RemoteProcess {
2        public :
3          InlineProcess(char* name) : (name, BLOCKING)
4          InlineProcess(char* name, ProcessType type) : (name, type)
5  }

                ⋮
```

Figure 3.3: Definition of InlineProcess

current scope of the parent. Thus, the child obtains the state of all shared data-objects that are currently active in the parent process. This includes all globally and locally defined shared data-objects. To minimize the expense of starting up the child, this set is limited to the set of shared data-objects that are actually accessed within the child. This can be further optimized by limiting the set of shared data-objects to the objects which are used in a particular execution paths within the child.

All variables and objects, which are not derived from the shared data-object class and are declared within the same scope as the `InlineProcess` object are defined within the child. However, they do not contain the same values that are present in the parent process and thus must be re-initialized inside the block which defines the child. Constant variables are the one exception, they have the same value in both processes.

The code executed by the `InlineProcess` object is defined by the block surrounding the declaration. In Figure 3.4 for example , the `InlineProcess` object `child1` is declared as a `Blocking` process object. The code which is executed by the child is enclosed between the two curly braces { and }.

```
                ⋮
1       RemoteInt a;
2
3       {       // start of successor function definition
4
5               InlineProcess child1("child1", BLOCKING);
6               a.Set(a.Access() + 1);
7
8       }       // end of successor function definition
                ⋮
```

Figure 3.4: InlineProcess declaration

Offline Processes

As mentioned earlier, there are situations where the child heavy-weight process should not be tied to the life span of the process object. In this case, an `OfflineProcess` object (cf. Figure 3.5) should be defined instead of an `InlineProcess` process. `OfflineProcess` objects create children which exist as long as the program is still active.

When defining an `OfflineProcess` object, an identifier, which must be unique within the current scope, is used to refer to a particular child. The code which is executed by the child, is defined by the enclosing block as for an `InlineProcess` (cf. Figure 3.4). When the `OfflineProcess` object is defined, the parent process determines whether a child exists with the same name and numeric identifier. If there is such a process, the child is restarted; if not, a new process is created. In Figure 3.6, an `OfflineProcess` object is defined inside a doubly nested loop. During the first pass through the outside loop, three children are created in the inner loop.

```
                    ⋮

1  class OfflineProcess : private RemoteProcess {
2    public :
3      OfflineProcess(char* name,int local_id) : (name, BLOCKING)
4      OfflineProcess(char* name,int local_id,ProcessType type) : (name,type)
5  }

                    ⋮
```

Figure 3.5: Definition of OfflineProcess

Subsequent passes through the outer loop result in the children being restarted. A parent process is blocked when it tries to restart a child which has not completed and continues when it is able to restart the child.

3.2.2 Implementation

There are two classes of processes — the `ParentProcess` class, used to define the parent process; and the `RemoteProcess` class, used to define the parent's representation of a child heavy-weight process. When a child is created, the parent recognizes it as a `RemoteProcess` object, while the child (the actual heavy weight thread) is a `ParentProcess` object just like its parent. `ParentProcesses` can create n children; thus, an n-ary tree structure is created where a parent has n children while a child has only one parent.

`ParentProcess` objects communicate through an asynchronous handler which interrupts the process when a message is received from its parent or one of its children. This allows a process to continue executing while it is waiting for its parent or one

```
                ⋮

1         RemoteInt distributed_index;
2         RemoteInt a[3];
3
4         for (int i = 0; i < 5; i++) {
5            for (int j = 0; j < 3; j++) {
6               distributed_index.Set(j);  // pass index to child
7               {
8                  OfflineProcess child("child",j);
9                  int k = distributed_index.Access();
10                 a[k].Set( a[k].Access() + 1 );
11              }
12           }
13        }

                ⋮
```

Figure 3.6: OfflineProcess declaration

of its children to respond. Asynchronous execution does not create a problem with synchronization, because synchronization is enforced by the shared data-objects. A process is stopped if the shared data-object is not ready to be accessed.

Processes send and receive two types of messages which are used internally by the process objects. The first type deals with process control and the second type deals with relaying messages among shared data-objects. Process control is accomplished with only four messages:

- START — The parent asks a child to start executing.
- COMPLETED — The child informs a parent of its completion.
- EXIT — The parent asks a child to exit.
- ERROR — The parent or child is informed that an exception occurred in the sender.

Processes relay messages for shared data-objects which allow shared data-objects to exchange information among themselves. When sending a message, the shared data-object is responsible for specifying the destination and type of the message. The system provides the following message types.

- REQUEST — The shared data-object is requesting the contents of a replicate.
- SET — The shared data-object is setting the contents of a replicate.

3.3 Shared data-objects

The motivation for using objects as a medium of communication is to simplify the specification of data exchange. This is accomplished by providing a method for writ-

ing distributed programs in a sequential manner, where the same piece of data can be accessed by several processes by simply accessing the local replicate. This is implemented by using shared data-objects which encapsulate functions for data transmission, synchronization, and mutual exclusion into their class definition. When user-defined classes are declared, they can obtain these capabilities through the use of inheritance.

Since the classes for shared data-objects provide the communication, programmers can disregard the logistical problems of distributed programming such as message passing, and concentrate on the design. In addition, programs designed with a specific computational model in mind (e.g., MIMD) can be used on other models by substituting the appropriate library. The programs can run unaltered, because the interface provided by the shared data-objects is consistent with SISD and MIMD models.

3.3.1 Shared data-object description

Two design factors significantly influenced the capabilities built into the classes and the overall class structure for our model for distributed programming.

Support System

The services required to implement shared data-objects had to be either, provided by the system or handled internally. Maintaining a high degree of portability and efficiency were the major concerns when designing the services.

Interface

Not all of the services required to implement shared data-objects need to be visible to the programmer. The interface is kept simple to allow easy integration of shared data-objects into sequential and parallel programs.

3.3.1.1 Support system

One of the goals of this thesis was to allow programs to be written in a manner that is independent of the computational model. This is partially achieved by providing services within the shared data-objects which do not rely on features specific to the underlying communication system or model. The services rendered by the shared data-objects are designed to work in a message-based system which provides reliable communication. When a shared data-object sends a message to another data-object, it assumes that after some indeterminate delay, the message will reach its destination. This requirement does not overly restrict the portability of shared data-objects, because most loosely and tightly coupled systems can provide some form of reliable communication.

The messages sent by the communication system can be of fixed length. In the current implementation, which is built on top of REM [Sho88], the communication system imposes a 4000 byte limit on the size of a message. Shared data-objects which are larger than this limit partition their contents into packets of 4000 bytes or less. Since messages are not guaranteed to arrive in the correct order, objects do not assume that the packets arrive in the same order as they are sent. By using a simple numbering scheme, the packets are re-ordered as they arrive and stored appropriately. Only when the entire contents of the object is received will the object allow access.

Shared data-objects always send their contents to a specific replicate. Since the program is represented as a tree structure (this is discussed in detail in the next chapter), shared data-objects interact with a limited number of replicates. When two objects exchange information, it is done through point-to-point asynchronous communication. Point-to-point communication is satisfactory, because objects only exchange information with replicates located in the parent process or in one of its

children.

The communication system and underlying model do not have to provide sophisticated features such as infinite length messages, broadcasting, multicasting, or global referencing. By eliminating these features, most modern networks and protocols can be used to provide the basic requirement for shared data-objects. If the system does not use the message passing paradigm, as in a shared memory system, message passing can be simulated.

3.3.1.2 Interface

To facilitate execution on the different computational models, specific class libraries have been created for the MIMD and SISD models. Compatibility between the two models is maintained by presenting a common interface to the objects. The success of this approach is a direct result of using the object-oriented technique of inheritance to define shared data-objects.

The services provided by the shared data-object classes are designed for use by a preprocessor which converts a sequential program designed for the SISD model into a distributed one. A program generated by the preprocesser contains the necessary flow control (i.e., calls to the methods defined in the shared data-object class) to execute correctly. The converted program can still be compiled for, but not run on, the SISD model. It can be compiled because the interface for shared data-objects is consistent with all models. However, the child processes for the distributed version (which are produced by the preprocessor) are compiled as separate executables which do not execute on an SISD system (unless some form of concurrent execution is available which allows messages to be exchanged among the processes).

The methods defined in the shared data-object classes are not intended to be

used directly by the programmer. Their access is not restricted, because there are special situations where the programmer needs control of the shared data-objects. This type of situation usually occurs when the programmer can take advantage of some attribute in the problem that a preprocessor is not aware of such as access patterns. These services (cf. Appendix A.1) override the automatic insertion of methods by the preprocessor. This may introduce conflicts (e.g., a parent and child update the same object). If this occurs, errors such as race conditions, deadlock, and data corruption become possible. Therefore, extreme caution must be used when the automated checking is not available from the preprocessor.

3.3.2 Implementation

The class hierarchy used for the C++ implementation of shared data-objects is based on the functionality required to support distributed communication. Thus, there is a natural partition consisting of *Objects, Addressing, Synchronization, Communication*, and *User defined classes* (cf. Figure 3.7). The lowest layer, Virtual Object, is a *virtual* class, used to allow abstract data classes to polymorphically handle objects. Hash tables, arrays, and stacks are examples of abstract data classes used in the implementation. The other layers, used in defining shared data-object classes, are discussed in order from top to bottom. First, we show how to define a class based on shared data-objects and then outline how the underlying classes support the distributed implementation.

User defined classes

In a user-defined class, objects can be conceptually viewed as regular C++ objects which are accessible by remote processes. For example, a class can be defined which provides the same capabilities as a primitive type defined in C++. The first class

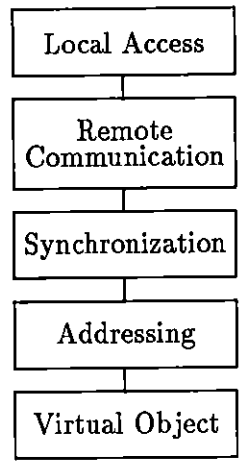


Figure 3.7: Shared objects class hierarchy

implemented is the class `RemoteInt` which has the same functionality as the primitive type `int`. Since the class `RemoteInt` inherits the shared data-object classes defined in the lower layers, details relating to distributed execution can be ignored.

The `RemoteInt` class is defined in the same manner as other classes in C++ (see Figure 3.8). However, the class description must conform to three specifications to function as a shared data-object. First, it must inherit the super-class `RemoteMemory` as a virtual to ensure that there is only one instance of `RemoteMemory` in its inheritance hierarchy. Second, it must pass its total size to the constructor of the class `RemoteMemory` so that its extent in memory can be determined. Lastly, the class must supply a virtual function `DataTransfer(void*)` which is used by the transfer methods in the shared data-object to copy the contents of the class (this method can ignore the information specifically related to the shared data-object). Through this

```
                ⋮

1   class RemoteInt : public virtual RemoteMemory {
2       int iv;
3     public :
4       RemoteInt() : RemoteMemory(sizeof(RemoteInt)) { iv = 0; }
5       int Access() const { return iv; }
6       void Set(int new_value) { iv = new_value; }
7       virtual void DataTransfer(void* copy)
8         { iv = ((RemoteInt*)copy)->iv; }
9   }

                ⋮
```

Figure 3.8: Definition of RemoteInt

method, the programmer specifies what information is to be exchanged among replicates. If some of the data defined in the class is left out of the transfer, the replicates may not be exact duplicates after the transfer of data. For the `RemoteInt` class, only the integer value stored in the class needs to be transferred in `DataTransfer()`. As seen in Figure 3.8, the method receives a void pointer which points to an object of its own type containing the remote information. Therefore, to obtain the value of the replicate, it recasts the pointer to the `RemoteInt` class and extracts the integer value `iv`.

The `RemoteInt` class locally stores and retrieves integer values through two user-defined methods, `Access()` and `Set(int)`. `Access()` returns the current value of the integer while `Set(int)` sets the value. To use `RemoteInt` in the expression `i = j + i` would require a simple translation to `i.Set(j.Access() + i.Access())`. This representation is less appealing than the original but this style is quite common for handling objects. Moreover, by using overloaded operators in C++ some of this

awkwardness can be avoided.

The class `RemoteInt` is now usable as a shared data-object. If an object of the class `RemoteInt` is defined and is active in a process defining an `InlineProcess`, the object is accessible by both processes. The scope of the new object follows C++ scope rules in both the current process and the `InlineProcess`. Also, both processes access the object through the user-defined methods as if it were a locally defined object.

Communication

The second layer of functionality defined in Figure 3.7 implements the remote communication among processors which enables replicated objects to exchange information. This layer is defined by the class `RemoteMemory` depicted in Figure 3.8 where it is inherited by the class `RemoteInt`. This class is responsible for transferring and receiving the local contents of an object to and from replicated versions. The general structure of the class is defined in Figure 3.9. The following methods are supplied for transferring data.

`RemoteGet()` — Ask for the contents of the parent's replicate and block until it is received.

`RemoteSet()` — Set the contents of the object to the parent's replicate.

`RemoteSend()` — Send the contents of the object to a replicate which resides at a specified address.

These commands are designed to be inserted by a preprocessor which inserts them when shared data-objects are required to transfer their contents to a replicate. However, these commands can be explicitly used by the programmer to force communication among processes.

```
                ⋮

 1   class RemoteMemory : public virtual RemoteData {
 2       void* memory;
 3       int totalsize;
 4       Bool raw;
 5     public :
 6       RemoteMemory(int size)
 7         { totalsize = size; memory = this; raw = False; }
 8
 9       RemoteMemory(int size, void* buffer)
10         { totalsize = size; memory = this; raw = True; }
11
                ⋮

12       virtual void RemoteGet();
13       virtual void RemoteSet();
14       virtual void RemoteSend(RemoteID&, ByteArray&, RemoteAddress&);
15
                ⋮

16   }
                ⋮
```

Figure 3.9: Definition of RemoteMemory

Synchronization

Shared data-objects create a situation where several children need to access the same object. To guarantee that a program executes correctly, access to an object and its replicates is synchronized. This is implemented through the use of *trace locks* which specify the order in which the processes access the object. This layer is implemented through the class `RemoteData` (cf. Figure 3.10). To separate read and write operations, two locks are maintained — one for reading from the object and one for writing to the object. The locks are only accessible through the following methods.

ReadDataAccess() This method blocks the local process if there are any outstanding readers for this object.

WriteDataAccess() This method blocks the local process if the pending writer has not written to the object.

ExpectedWriter(RemoteID) Returns true if the specified process is the last process to write to the object.

SetReadBlock(RemoteProcess) The specified process must read the current state of this object before the contents can be changed. This lock is actually a list of processes since more than one process can read the object without conflict. This method blocks if there is a pending write lock.

SetWriterBlock(RemoteProcess) The specified process is the next process to write to this object. This method is blocked if there is an outstanding read lock. If there is an outstanding write lock, it is simply overwritten.

SetReader(RemoteID) The specified reader is added to the list of processes which have successfully read the object. If this list equals the list of processes which need to read the object, then there are no outstanding readers.

SetWriter(RemoteID) The specified writer tries to become the last process to write to the object. If this writer is not the currently specified writer, the results are ignored.

Trace locks are used to specify which processes can read and write to an object. The task of inserting these statements is a tedious and potentially error-prone one. Therefore, this step is best left to preprocessor which can determine where the appropriate lock statements are to be inserted by analyzing the static data flow and control flow.

When a child heavy-weight process is created, it retains the current scope environment of the parent. Thus, the local and global shared data-objects which are currently active in the parent are also active within the child. By setting read and write locks on all variables visible to the parent and the child, sequential execution can be enforced. However, this is not necessary, because parallelism can be obtained by setting locks only on objects which are in conflict between the parent and the child.

Addressing

The final layer, *Address*, is used to identify replicated objects within different scopes and across different processors. Unique IDs are guaranteed in parents and children by incrementing a counter. However, the IDs are not guaranteed to be unique among children. This does not cause a problem, since an object is not shared among children unless the children share the object with a common ancestor process. In the latter case, the ID is unique, because it is unique within the ancestor.

The `VirtualAddress` class implements the addressing. Initially, all global variables are assigned a unique ID which is static across all processes. As new shared

```
                ⋮

1   class RemoteData : public VirtualAddress {
2         Bool vm;
3         RemoteID process;
4         ReadLock readers;
5         WriteLock writers;
6      public :
7         RemoteData();

                ⋮

8         /* Locking functions */
9
10        void ReadDataAccess();
11        void WriteDataAccess() const;
12        Bool ExpectedWriter(RemoteID id) const;
13        void SetReadBlock(RemoteProcess& proc);
14        void SetWriteBlock(RemoteProcess& proc);
15        void SetReader(RemoteID id);
16        void SetWriter(RemoteID id);

                ⋮

17  };

                ⋮
```

Figure 3.10: Definition of RemoteData

data-objects are declared within a given scope, they are assigned unique IDs starting with the highest number received from the parent process. In the case of the main process, the number is greater than the last ID assigned to a global variable.

3.4 Summary

This chapter introduced two types of objects — process objects and shared data-objects. The process objects provide a convenient method for creating, executing, and terminating children (heavy-weight processes). They also provide sufficient methods for shared data-objects to transmit data to and from replicates. The shared data-object classes are used to build user-defined classes. These classes can be replicated across several processors providing a convenient method for transferring data. When designing the class, the programmer only needs to be concerned with providing the local methods for the class. All methods used for distributed execution (except the `DataTransfer()` method) are supplied by the shared data-object class.

Chapter 4

Implementing the model

In the shared data-driven object model, data is encapsulated into shared data-objects which provide a safe mechanism for replicating and transferring data to multiple processes. Process objects are used to create, execute, and terminate children (heavy-weight processes). The transfer is performed by the shared data-objects, but controlled by statements inserted by a preprocessor which generates parallel programs from sequential ones.

The preprocessor is able to do this by performing a static data-flow analysis on the program. This type of analysis provides sufficient information, on the ordering of events in the program, to do the transformation. Since, the ordering of statements in a sequential program determines the order of execution, assertions can be made about the partial ordering of the processes (or more accurately the block of statements delegated to the processes) which execute in parallel. These assertions are maintained in the parallel version through the use of trace locks.

4.1 Interaction of process objects

Children are dynamically created by defining process objects as discussed in Chapter 3. If a process object is defined within that child, a new child process is created and connected. In this manner a tree structure is created where every process has one parent process and possibly several children and the main process of the program is at the root of the tree.

The portion of the program which is executed by the child process is specified by defining a block in the program which contains a declaration of a process object. A block is defined as a sequence of consecutive instructions that always executes from start to finish [Ken81]. In the parallel version, the blocks delegated to the children are removed from the parent. Therefore, the pace of the children which are executing in parallel, must be controlled to ensure that the order in which the variables are assessed is equivalent to the order defined by the original sequential code.

4.1.1 Controlling process objects

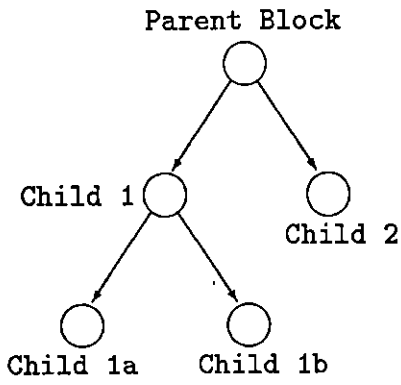
In data-flow analysis, an *activation* is usually considered the execution of a procedure. For the data-driven object model, this definition has been altered slightly. When performing an analysis of a program, which is to be parallelized, it is critical to understand what happens to shared data-objects when a child is spawned. Therefore, activations are defined as the execution of a block rather than a procedure. A Procedure is considered a special type of block which may or may not have parameters. The procedure calls define where the activations occur.

A network of processes, in the form of a tree, is created from the sequence of block activations which contain definitions of process objects. This sequence of activations

is defined by an *activation tree* which depicts the way control flow enters and leaves activations [ASU86]. Therefore, the activation tree can be used to represent and ultimately control the network of children.

In Figure 4.1, `InlineProcess` and `OfflineProcess` objects are declared inside nested blocks. These program fragments are used to define the activation tree on the left. A node in the tree represents the activation of a block and the label attached to the node indicates which process object is defined in the block. Since, a block is executed in the child associated with the process object, it can be deduced from the diagram that the activation tree has the same structure as the communication network which connects the processes. Section 2.6 also showed that the activation tree provides information on how data flows among the activations. Therefore, the activation tree depicts the flow of information over the communication lines. This information can be used (by inserting statements into the program by a preprocessor) to guarantee that the correct data flow occurs among the children at run time.

Data-flow analysis on the activation tree reveals which shared data-objects are used and set in an activation (i.e., a child process). This information is used by a shared data-objects in two ways. First, shared data-objects which are used and set in a child must transfer their current state to a child when it starts to execute. It must also obtain the final state from the child when the child terminates. Second, shared data-objects used or set in a child must ensure that subsequent children receive the correct value and subsequent accesses by the parent result in the parent accessing the correct value.




```
main(){ // Parent Block
   { // Child 1
      InlineProcess Child1("Child1")
      { // Child 1a
         InlineProcess Child1a("Child1a")
      }
      { // Child 1b
         InlineProcess Child1b("Child1b")
      }
   }
   { // Child 2
      InlineProcess Child2("Child2")
   }
}
```

Figure 4.1: Activation tree

```
                ⋮
1       RemoteInt current_count;
                ⋮
2       InlineProcess child1("child1");
3       current_count.SetReadBlock(child1) // lock is set
4       me.Start(child1);
                ⋮
```

Figure 4.2: Activation of remote child

4.1.2 Implementing control

In the parent process, the block of code executed by the child process is replaced by a definition of the object and a message to start the process. The creation of the process object is separated from the start to allow the shared data-object's locks, such as the one in Figure 4.2 for `current_count`, to be set before the child has had a chance to execute.

The action taken when a process terminates depends on the type of `process object` defined. If it is an `InlineProcess`, it terminates when the parent leaves the scope in which the `InlineProcess's` object is defined. If it is an `OfflineProcess`, it terminates when the topmost block in the activation tree is completed.

An `OfflineProcess` is the only type of process that can enforce synchronization when it is started. This occurs when the child process has previously been started and is still executing. The parent process is forced to synchronize with the child before restarting it to avoid concurrent use of the same process. This means the

parent process is blocked while trying to restart a process until the child terminates. This restriction exists because concurrent light-weight threads which could handle concurrent execution of the block have not been implemented.

`InlineProcesses` or `OfflineProcesses`, which have been defined as `BLOCKING` processes, synchronize with their parent process. This occurs when the parent tries to delete the process object which is associated with the child. The parent is blocked, thus synchronizing with the child, until the child responds informing the parent of its completion. If the child is unexpectedly terminated (e.g., crash) the parent blocks indefinitely.

4.2 Data transfer through shared data-objects

Shared data-objects, as described by Bal and Tanenbaum, simulate shared memory by replicating data across multiple processors. When a process accesses a shared data-object, it accesses a local copy which is kept consistent with copies existing in other processes. If an access changes the state of the data stored in the shared data-object, the change is propagated to the other copies. If more than one process tries to access the object, the accesses are ordered according to some criterion (e.g., first come, first served).

In the shared data-driven-object model the requirements for consistency, update propagation, and ordering of accesses are controlled by the sequential ordering. Shared data-objects in this model do not explicitly deal with these problems. They are dealt with by restricting the manner in which the shared data-objects can be connected. Objects are connected according to the activation tree generated by the program. As mentioned above, information is provided by compile-time data-flow analysis which resolves all access and propagation difficulties.

Shared data-objects are the basis for user-defined objects. These objects obtain replication, communication, and ordering capabilities by inheriting the shared data-object's classes. Conceptually, these objects can be viewed as clusters of replicated objects tied together by a private network. New data objects are created dynamically when a new process object is defined (i.e., by creating a child in which the shared data-objects are declared). Each object in the new process is connected to its copy in the parent process, thus, creating a network which follows the activation tree.

4.2.1 Naming

In the actual implementation each cluster is not connected by a separate network. They are multiplexed over a single network which is managed by the processes. This requires shared data-objects (which are replicated in several separate address spaces associated with the different processes) to be able to uniquely identify themselves. Naming schemes, which are based on physical locations, are not viable, since there is no guarantee that the replicates are located at the same physical address. Therefore, objects refer to each other by a virtual address which provides a unique name for each cluster. Each object in the cluster communicates with a replicate by simply supplying its own virtual address and the address of the process which contains the receiver.

4.3 Controlling shared data-objects

Control of the program is maintained by blocking accesses to shared data-objects to ensure that processes access the data in the correct order. References and updates in a distributed program are made according to a partial ordering of the program. The partial ordering in this model is computed from the activation tree, which specifies

a sequence of events that occur when the activation is started. This partial ordering specifies whether a process can access an object or must wait for another event to occur (i.e., the object is accessed by another process).

A static data-flow analysis on the activation tree is used to determine the potential conflicts that can co-exist between a parent and a child process. Conflicts occur when two processes alternately read and write from/to a shared data-object. If two processes write to a shared data-object before either reads it, there is no conflict. There is also no conflict if two processes try to read the same object.

Conflicts indicate which shared data-objects are affected by the upcoming activation so that trace messages are inserted into the program to dynamically inform these objects at run time about the pending operations (i.e., pending reads or writes by child processes). When a process tries to access an object, a statement is inserted prior to the access which asks the shared data-object, to make sure the pending operations have occurred. The shared data-object blocks the query if the correct operation has not occurred.

4.3.1 Data-flow analysis

Data-flow analysis is used to obtain information concerning the usage of variables, or in the case of the data-driven-object model, the usage of shared data-objects. The analysis typically uses a graph like the activation graph described by Banning used to describe the unbounded activation tree [Ban79]. For the data-driven-object model the primary type of activation is the block. Procedures are handled by creating a new activation and attaching a function to the arc for the parameters. The program is partitioned and mapped to the activation tree by breaking the program down into blocks until all process objects are defined in a single block.

```
                ⋮
1       if (Some_Condition) {
2          statement;
3          statement;
4       }
5       else {
6          statement;
7       }
                ⋮
```

Figure 4.3: Examples of potential blocks

In Figure 4.3 for example, the group of lines 1–7, 2–3, and 6 are all considered blocks. Groups of lines which are not blocks are 2–6, 3–5, and any other grouping that violates the boundary of the `if` statement. If a block is created from lines 1–7, three children are created by separating lines 2, 3, and 6 into separate blocks.

The activation tree is used to represent a program, because it reflects the sequences of activations which occur in the sequential program at run time. This makes it useful for generating trace information for the parallel version of the program. As a program proceeds through the activation tree, it enters an activation which is executed in a child. This child process has priority over the parent for all shared data-objects. This is always true, because the block which is executed by the child executes next in the sequential version. This has two ramifications; first, the child can never be locked out by the parent and, hence, can execute freely without considering the actions of the parent; second, the parent is always blocked from accessing a shared data-object required by a child until the child is finished with the object. By following the same argument, this extends not only to that child, but also to all activations within the

child that are executed in other processes.

To determine which shared data-objects cause conflicts between the parent and the child, the child's block is examined for objects used or set in the child and active in the parent. The set of USED objects for a child is found by evaluating expressions within the block and recording the objects that have had a constant method invoked. A method is said to be constant if it does not alter the state of an object (e.g., the `Access()` method defined for the `RemoteInt` class in Figure 3.8). Objects which invoke non-constant methods must be included in the USED set, unless it can be shown that the method only stores information in the object. The set of shared data-objects used in the activation is called the `USED` set. The set of MODified objects is found in a similar manner. The blocks are analyzed to find the objects which invoke a non-constant method. The set of shared data-objects that is modified in an activation is called the `MOD` set.

4.3.2 Synchronization

Children of the same parent are prioritized on the sequence of activations. The first child created is given priority access to shared data-objects over all subsequent children. Thus, the first child can block any subsequent child that requires access to a shared data-object. This property guarantees that the first child activated is not prevented from accessing an object and, hence, deadlock can never happen. Fairness, which is another aspect of resource sharing, is also achieved, because each child becomes free to execute as soon as an old child terminates.

4.3.3 Locking

Before executing a child, all shared data-objects in the USED set that are active in the parent process must be informed that this child is about to read their contents. If one of these shared data-objects is not written to by its specified writer, the operation to set the lock must block. If the child started to execute, it might read the object before the correct value is set by the appropriate writer which violates the correct partial ordering of the program.

The shared data-objects that are active in the parent and are in the MOD set are informed that the child is about to change their contents by writing to their replicates. If an object in the MOD set is not read by all pending readers, the operation to set the lock blocks. This prevents the child from executing until it can modify all the necessary shared data-objects in its block of code.

A simple example

In Figure 4.4, there are four variables A, B, C, and D which must be accessed in the correct order. When this program is run sequentially, the proper ordering is trivially achieved. If the two blocks defined within the code are executed in two children, the order must be maintained. By visually inspecting the code, it is clear that the first and second children can execute in parallel with the parent. This is true until the parent tries to reassign A, or requires the value in B or C, which must be set by the first and second children, respectively. Since there is no conflict between the two children, they can be executed in parallel.

To ensure that the correct partial order is maintained for the program, the parent must be forced to wait for both children to read `RemoteInt A`. This is accomplished by adding the first and second child to the list of processes which must read A. Similarly,

```
                    ⋮
1     // Parent's scope with four RemoteInts defined
2     // A, B, C and D
3
4     A.Set(1);
5       {
6          Inline child1("child1");
7          B.Set(A.Access());
8       }
9       {
10         Inline child2("child2");
11         C.Set(A.Access());
12      }
13    A.Set(2);
14    D.Set(C.Access() + B.Access());
                    ⋮
```

Figure 4.4: Simple example of parallel execution

the first child needs to be specified as the pending writer for B and the second child as the pending writer for C. The parent is blocked until both children have read object A . This is achieved by placing a `ReadDataAccess()` call to the object A prior to the statement that resets A. The parent is blocked until both children have returned the value to B and C, respectively. This is accomplished by placing a `WriteDataAccess()` call to objects B and C. When these methods are inserted the program executes according to the partial order regardless of the execution order of the children.

In Figure 4.5, the simple example of Figure 4.4 is expanded to include the necessary trace locks to ensure correct execution. Before the first child is executed, `RemoteInt` A has a read block placed on it to guarantee that it is not changed before child 1

has had a chance to read it. `RemoteInt B` has a write block placed on it to restrict any further read blocks until the first child has actually returned a value. The same procedure is applied to the second child except a write block is placed on `RemoteInt C` instead of on `B`. When the parent reaches the statement `A.ReadDataAccess()`, it blocks until all the processes which are scheduled to read `A` (which is currently both processes) have received the value. The parent also blocks on `B.WriteDataAccess()` or `C.WriteDataAccess()` until the children have returned their respective values.

Inserting trace and transfer commands

We mentioned at the beginning of this section that shared data-objects are informed of pending operations that are to occur in a child if they are used or set in that child. An example of this is given in Figure 4.5. However, this is not the only requirement. For trace locks to work, data must be transferred in a timely fashion. If the `WriteDataAccess()` method is invoked for an object, the child which is to write to this object must have its object (i.e., the replicate) set to the local copy. If the local object is never set by the replicate, the parent process is blocked permanently. Therefore, for trace locks to execute correctly, the shared data-objects in the parent process must transmit the current data before the child executes and the replicates in the child must return the final value to the parent when it finishes.

Shared data-objects use read and write trace locks to manage the information supplied for controlling the access to the object. To decide when accesses are allowed, the object's trace lock needs to know whether the object is used and/or set in a child process. Since the trace information is supplied before the activation, the locks are only supplied with information that is pertinent to the current execution. If the activation path in the parent process bypasses the activation of a child, the trace statements providing the information at the child activation are also bypassed.

```
                ⋮

 1     // Parent's scope with four RemoteInts defined
 2     // A, B, C and D
 3
 4    A.Set(1);
 5
 6    InlineProcess child1("child1");
 7    A.SetReadBlock(child1);   // Child 1 must read A as it is now.
 8    B.SetWriteBlock(child1);  // Child 1 must write to B.
 9    Start(child1);
10
11    InlineProcess child2("child2");
12    A.SetReadBlock(child2);   // Child 2 must read A as it is now.
13    C.SetWriteBlock(child2);  // Child 2 must write to C.
14    Start(child2);
15    A.ReadDataAccess();// Ensure all processes have read A
16
17    A.Set(2);
18    B.WriteDataAccess();  // Did the last specified process write to B
19    C.WriteDataAccess();  // Did the last specified process write to C
20    D.Set(C.Set() + B.Set());

                ⋮
```

Figure 4.5: Simple example with locks

```
                ⋮
1       A.RemoteGet();  // initialize A
2       B.Set(A.Access());
3       B.RemoteSet();
                ⋮
```

Figure 4.6: Child execution

Since the locking activity is isolated within the parent process, the children do not need to worry about coordinating their activity with the parent while they are executing. However, with the invocation of a child, the shared data-objects in the child's USED set must initialize themselves by obtaining the contents from the shared data-object in the parent process. If this is done during the startup, the parent is then free to overwrite all objects used in the child. When a child terminates, the shared data-objects in the MOD set must ensure that they transfer their contents to the parent's object.

Commands to explicitly read and write shared data-objects in the USED and MOD sets are inserted into the child. In Figure 4.5 for example, an InlineProcess called child1 is declared and executed. In Figure 4.6, the block of code executed by this process is expanded to include the necessary statements to initialize and then set the required shared data-objects. The used set for the child is defined as USED = {A} and the modified set is MOD = {B}. Therefore, statements must be inserted to initialize A and to set B. The statement to initialize A (line 1) must be placed prior to the first usage of the object and the statement to perform the set of the parent's object B (line 3) must be placed after the last statement in which B is set locally.

4.3.4 Embedded trace-lock command

Embedded commands provide more control when data is to be transferred. If an object is modified several times in a child, the parent must be delayed until the last modification occurs. If the parent is allowed to continue prior to the last update, the partial ordering of the program is violated.

By inserting commands instead of having the trace lock commands tied to the objects methods, the commands are issued at critical times to ensure the correct partial ordering. Thus, user-defined methods can be implemented without checking the status of locks or explicitly updating replicates. Therefore, user-defined object classes based on shared data-objects such as `RemoteInt` have the same execution speed as a comparable class defined in C++ which does not inherit the shared data-object class.

There are several benefits to using trace locks. They can be inserted without restructuring the program's code. Trace locks do not alter the path of execution through a program, only the pace. The number of messages required to regulate the processes is kept to a minimum. In the current implementation, all the current values for local shared data-objects are passed to the child in one message. Since the child is not affected by the parent, there are no subsequent messages required to obtain data or access rights from the parent.

4.4 Summary

This chapter discussed the shared data-driven-object model which provides shared data-objects, process objects, and data-flow directed execution. We showed that shared data-objects and process objects can synchronize distributed processes to

guarantee that the program's partial order is maintained. This is only true if the shared data-objects are given sufficient information to manage the locks and transfer data at the appropriate times. The information to perform the synchronization is obtained from performing data-flow analysis on the activation tree of the program. This analysis also stipulates when the shared data-objects must set the contents of a replicate. Therefore, by limiting the way processes are interconnected to the activation tree, problems with communication, addressing, synchronization, and mutual exclusion are avoided. Applications are able to implicitly provide solutions to these problems in the structure of their program.

Synchronization based on trace locks allows the preprocessor to automatically insert trace statements and transfer statements into a program. Both the placement and the contents of these statements can also be determined from a static data-flow analysis of the program. By using a preprocessor, the programmer need not be concerned about lock and transfer methods supplied by shared data-objects.

Chapter 5

The preprocessor

The preprocessor is intended to provide two services: the automatic partitioning of a program into separate executables based on the blocks specified in the program and the insertion of trace statements which allow shared data-objects to control the pace of the program. The classes for shared data-objects and the process objects were designed with the intent to simplify the preprocessing.

5.1 Partitioning

As mentioned in Chapter 3, the parent process and every child heavy-weight process sees itself as a `ParentProcess` object. The sequential description is split up into segments, according to the blocks inserted by the programmer, where each segment is executed by a `ParentProcess` object. The segment is used to define a method (cf. Section 2.1) which is declared in the `ParentProcess` class. This method is generated by the preprocessor and not supplied in the library for the `ParentProcess` class. When the `ParentProcess` object executes this method, it executes the segment of

code originally found in the sequential description. The effect therefore, is equivalent to executing the sequential description.

The first step in partitioning the sequential description is to alter it so it executes as a method in the `ParentProcess` class.

1. The main procedure of the program is rewritten as the method `ParentProcess::Execute()`.
2. The main procedure is replaced with a single call to the method `ParentProcess::Initialize()`.

Creating the `ParentProcess` object for the children processes is a slightly more complex task. The block of code which defines the child process must be removed, leaving just the declaration of the process object. If the block which enclosed the definition is not part of a statement, it is discarded and the child is created by the following steps.

1. Remove the block of code from the parent which defines the child.
2. Create an exact replicate of the parent's global environment in the child.
3. Place the block of code, which is defined for the child process, in the `ParentProcess::Execute()` method. Insert declarations for all locally declared shared data-objects which are active within the child.
4. Create the main routine for the child which makes a single call to the `ParentProcess::RemoteInitialize()` method.
5. Recursively perform these steps until all the embedded children have been removed.

5.2 Automatic partitioning

A preprocessor can be used to take the sequential description and generate the separate object-oriented programs described above. The file-name of the executable, which is created by the preprocessor, is defined by the argument which is supplied in object declaration (which must be a constant character string). In the Figure 3.4, an executable called `child1` is created. REM requires the name and the path of the executable to be able to create a process on a remote node. To automatically generate executables whose system name corresponds to the name supplied as a parameter, the preprocessor can generate a unique file-name based on name (e.g., the constant string) used in the declaration.

5.3 Automatic insertion of trace locks

The second task performed by the preprocessor is the insertion of trace lock statments which invoke methods defined in the `RemoteMemory` and `RemoteData` classes. The data-flow analysis performed by the preprocessor is described in Chapter 4. This model forms the basis for analytically determining and positioning trace lock statements.

5.4 Algorithm to partition a program

The following algorithm uses data-flow and control-flow information to recursively partition a program into separate programs. An example using this preprocessor is given in Appendix C.

5.4.1 Declarations

Let $PROC$ be the set of blocks defined in the sequential program description which have a `RemoteProcess` derived object (i.e., `InlineProcess or OfflineProcess`) declared in their braces. This set does not include any nested blocks which may contain declarations of `RemoteProcess` objects.

Let $USED[b]$ be the set of `RemoteData` derived objects[1] which have a constant or a non-constant method[2] invoked in block b.

Let $MOD[b]$ be the set of `RemoteData` derived objects which have a non-constant method[3] invoked in block b.

Let $GLOBAL$ be the enumerated set of `RemoteData` derived global objects.

Let $LOCAL[b]$ be the set of `RemoteData` derived local objects which are currently visible inside of block b.

Let $CODE[b]$ be the sequential description of the block b.

5.4.2 Procedure Generate_Parallel_Description()

Input. Sequential description of the program.

Output. Program which invokes the first process in the program.

Method. The main procedure in the C++ program is replaced by a single call which invokes the execute method for the process object. The code in main is treated as a block and is used as a parameter to `Transform_Block()`.

1. Generate $GLOBAL$.
2. $LOCAL[main] = \emptyset$
3. Call Transform_Block() passing: $GLOBAL$, $LOCAL[main]$, and $CODE[main]$.
4. Write out main statement which calls `ParentProcess::Execute()`.

[1] Objects are renamed to avoid duplicates.
[2] This should only include methods which do not modify the state of the object.
[3] This should only include methods which do modify the state of the object.

5.4.3 Procedure Transform_Block()

Input. $GLOBAL$, $LOCAL[parent]$ — local declarations of the parent, and $CODE[block]$ — sequential description of the block.

Output. Parallel version of the block.

Method. For the given code determine all blocks which are executed by a child process. For each block recursively call this routine to eliminate any nested blocks. Also replace each block with a declaration, lock and transfer information, and a start message.

1. Generate $PROC$ from $CODE[block]$
2. **for** each b in $PROC$ **do**
   (a) Retrieve *filename* from `RemoteProcess` declaration and make unique.
   (b) Set ID to the `RemoteProcess`'s declared name.
   (c) Generate $LOCAL[b]$, $USED[b]$, $MOD[b]$, and $CODE[b]$.
   (d) Call `Transform_Block()` passing: $GLOBAL$, $LOCAL[b]$, and $CODE[b]$.
   (e) Write out process declaration (remove the braces defining the current block, if it is not part of a C++ statement).
   (f) Call Replace_Block() passing: $GLOBAL$, $LOCAL[parent]$, $USED[b]$, $MOD[b]$, and ID.
   (g) Write out start message.
   (h) Call Generate_Child() passing: $GLOBAL$, $LOCAL[parent]$, $LOCAL[b]$, $USED[b]$, $MOD[b]$, *filename*, and $CODE[block]$.
   (i) Write out block b.

5.4.4 Procedure Replace_Block()

Input. $GLOBAL$, $LOCAL[parent]$ — local declarations of the parent, $USED[b]$ — used objects in the block, $MOD[b]$ — used objects in the block, ID — process declaration, $CODE[b]$ — sequential description of the block.

Output. Writes out trace lock statement, and transfer statements.

Method. Visible objects in the parent process, which are modified by the block, are told via a trace statement. Global variables which are used in the block are told which children will use or modify them. All local variables that are used or modified are pushed into a message which is delivered when a child starts up. This eliminates the need to send several messages (i.e., one for each local object).

1. **for** each `object` in $GLOBAL \cap USED[b]$ **do**
   (a) Write message `object.SetReadBlock(ID)`
   (b) Precede each statement which modifies `object` in $CODE[b]$ with `object.ReadDataAccess()`
2. **for** each `object` in $LOCAL[parent] \cup GLOBAL \cap MOD[b]$ **do**
   (a) Write message `object.SetWriteBlock(ID)`
   (b) Precede each statement which uses `object` in $CODE[b]$ with `object.WriteDataAccess()`
3. **for** each `object` in $LOCAL[parent] \cap USED[b]$ **do**
   (a) Write message `object.PackageAddress(ID)`

5.4.5 Procedure Generate_Child()

Input. $GLOBAL$, $LOCAL[parent]$ — local declarations of the parent, $LOCAL[b]$ — local declarations for this block, $USED[b]$ — used objects in the block, $MOD[b]$ — used objects in the block, *filename* — unique file to write out to, $CODE[b]$ — sequential description of the block.

Output. Writes out a file which contains the code for the child process.

Method. All headers and global variables are included maintaining the same order of declaration as in the sequential description. A main procedure is written which calls the execute method created for the child. The execute method contains local declarations and the code specified by the block.

1. Write the headers and global variables to *filename.*
2. Write the main procedure and the preamble for the externally defined methods declared for `ParentProcess` class (i.e, `main()` and `ParentProcess::Startup()`) to *filename*
3. Write out `me.OldVariables()` to *filename.* This indicates that the following variables are defined externally to this block.
4. **for** each `object` in $LOCAL[parent] \cap USED[b]$ **do**
   (a) Write message `ObjectClass object` to *filename*
5. Write out `me.NewVariables()` to *filename.* This indicates that the following variables are defined locally to this block.
6. **for** each `object` in $LOCAL[b] - LOCAL[parent]$ **do**
   (a) Write message `ObjectClass object` to *filename.*
7. **for** each `object` in $GLOBAL \cap USED[b]$ **do**
   (a) Write message `object.RemoteGet()` to *filename.*
8. Write out $CODE[b]$ to *filename.*
9. **for** each `object` in $GLOBAL \cup LOCAL[parent] \cap MOD[b]$ **do**
   (a) Write message `object.RemoteSet()` to *filename.*

5.5 Summary

A preprocessor can automatically partition the program into separate segments and insert trace information. With the preprocessor, programs can be described sequentially and quickly converted to separate parallel programs which execute in a distributed manner. The preprocessor's partition algorithm recursively removes segments and generates separate sequential programs. Every program is a separate executable that runs as a heavy-weight process on a processor.

The preprocessor performs data-flow and control-flow analysis to determine how the segments conflict with each other. The data-flow information is used to insert trace information at the point where the child process is executed. The control-flow information is used to generate accurate information on where accesses to shared data-objects, which conflict, occur. Statements to check whether the proper execution sequence has occurred are placed prior to the accesses.

The partitioning and data analysis performed on a program is straightforward once the program is correctly parsed. Accurately parsing C++ is difficult because the language is not LALR(1). The test programs used in this thesis were hand preprocessed which turned out to be a simple, but tedious and error-prone task. Even with hand preprocessing it was found that modifications were easier to do on the sequential version first, then re-hand preprocess the program. The effort saved by using the sequential versions compensated for the time lost in preprocessing.

Chapter 6

Design of parallel programs

Developing parallel programs has traditionally introduced added complexity by forcing the programmer to explicitly state when data is to be transferred. Four examples are presented which illustrate how the shared data-driven object model provides a simple and uniform method of programming. From a programmer's point of view, shared data-objects allow to transfer data by simply declaring an object which is visible to the declaration of one or more process objects.

6.1 Single assignment problem

Chandy and Misra introduced the concept of *program schema* which is a class of UNITY programs and associated mappings. They define the single assignment schema as programs where the "assign-section of the program consists of a single statement" [CM88]. They proposed a mapping of the program in Figure 6.1 to an asynchronous architecture by using a single element buffer in shared memory. The buffers are used to relay values of `x` and `y` between two processors which execute the functions `f(x, y)`

```
1   Program SSS
2      assign x := f(x, y) || y := g(x, y)
3   end {SSS}
```

Figure 6.1: Single statement schema

and `g(x, y)`. The buffers act as communication channels between the processes allowing the values to flow from one function to the other.

To implement this algorithm two shared data-objects are used instead of the two single element buffers. Shared data-objects provide the same capabilities as buffers placed in shared memory. In Figure 6.2, two `RemoteInt` objects, `channel1` and `channel2`, act as the channels described by Chandy and Misra. In this example, the functions are defined as `f(x, y) = x + y` and `g(x, y) = x * y`.

6.2 Divide-and-conquer sort

The divide-and-conquer sort illustrates the similarities between programming sequential programs and parallel programs using shared data-objects. In the sequential divide-and-conquer sort routine, depicted in Figure 6.3, the program receives an integer buffer (i.e., `buf`) with a specified `length`. If the buffer is greater than two, it is partitioned around a pivot value in the function `Partition()`. The two partitions are then used as parameters for recursive calls to `DivideSort`. Implementing this algorithm with messages would mean that the partitions would have to be bundled up into messages and then un-bundled in the children. Also, to execute the recursive calls in parallel would require the children to execute asynchronously. These problems

```
                ⋮

 1   main() {
 2      RemoteInt x;
 3      RemoteInt y;
 4      RemoteInt channel1;
 5      RemoteInt channel2;
 6
 7      x.Set(2);
 8      y.Set(1);
 9
10      while (True) {
11         channel1.Set(x.Access());
12         channel2.Set(y.Access());
13         {
14            OfflineProcess child1("child1", 1);
15            x.Set(x.Access() + channel2.Access());
16         }
17         {
18            OfflineProcess child2("child2", 1);
19            y.Set(channel1.Access() * y.Access());
20         }
21      }
22   }

                ⋮
```

Figure 6.2: Data-flow single statement schema

```
                ⋮

 1   void DivideSort(int* buf, int length) {
 2      int left = 0;
 3      int start, finish;
 4
 5      if (left + 1 > length) {
 6         if (*(buf+left) > *(buf+length))
 7            Swap(buf, left, length);
 8      }
 9      else if (left + 1 < length) {
10         Partition(buf, left, length, start, finish);
11         DivideSort((buf+left), (finish-left+1));
12         DivideSort((buf+start), (length-start));
13      }
14   }

                ⋮
```

Figure 6.3: Sequential divide-and-conquer sort

can be solved by using message passing, but if the programmer is not familiar with parallel programming, this type of problem is not trivial to implement.

To develop a parallel version of the divide-and-conquer sort, the serial `DivideSort()` is used as a template. Instead of recursively partitioning the data on a single machine as in `DivideSort()`, the data is partitioned and sent to two children. These processes recursively partition the data and if it is larger than some threshold value, the partitions are sent to two additional processes. If the data is smaller than the threshold, they are sorted using the sequential `DivideSort()` routine.

The parallel version of `DivideSort()` is shown in Figure 6.4 in pseudo C++ code. If the buffer is greater than the threshold, the parallel version partitions the data in

`Partition()`. The offset into the integer buffer and the length of the partition, for both partitions, are stored in `RemoteBuffer` objects. These are special immutable objects that allow fixed length sections of local memory to be replicated to remote processes. When `PtrA` and `PtrB` are declared, after the `Partition()` call, they are initialized so that they point to mutually exclusive portions of the buffer.

Instead of calling the sort routine twice recursively two blocks are declared which create `InlineProcess` objects. The `InlineProcesses child1` and `child2` extract the address and the length of the partition from the `RemoteBuffers`. This partition is then used to recursively call `P_DivideSort()`. The recursive call no longer happens in the parent process but in a child process.

There is a definite similarity between the sequential and parallel code for the divide-and-conquer sort functions. In fact, the actual program which is listed in Appendix B was tested sequentially before it was executed in parallel. Like the previous example, the parallel version was created by hand preprocessing the sequential version. Three versions of the program were executed on Sun 3/60's to compare the execution time for sorting 100,000 integers (see Table 6.1). Process 1 is a sequential version which calls `P_DivideSort()`; Process 2 is a sequential version which only calls `DivideSort()`; and Process 3 is a distributed version where a parent creates two children executing in parallel (Processes 4 and 5).

The timings for Processes 3, 4, and 5 includes the time required to send and receive the data from the child as well as the time required to sort the data. Note the slowdown experienced in Process 1 (compared to Process 2) due to the overhead encountered when executing `P_DivideSort()`. In Process 3, the parent, there is a marked improvement in execution time, because the parent is only responsible for partitioning the data once and sending and receiving the partitions from the children. This number could be improved by an order of magnitude if the communication

```
                    ⋮

 1   void P_DivideSort(int* buf, int length) {
 2      int left = 0;
 3      int start, finish;
 4
 5      if (left + THRESHOLD >= length) {
 6         DivideSort(buf, length);
 7      }
 8      else {
 9         Partition(buf, left, length, start, finish);

                    ⋮

10         RemoteBuffer PtrA(left_length, buf + left);
11         RemoteBuffer PtrA(right_length, buf + start);
12         {
13            InlineProcess child1("child1");

                    ⋮

14            int* a_buf =  PtrA.Address();
15            int  a_size = PtrA.Length();
16            P_DivideSort(a_buf, a_size);
17            PtrA.RemoteSet();
18         }
19
20         {
21            InlineProcess child2("child2");
22

                    ⋮

23            int* b_buf = PtrB.Address();
24            int  b_size = PtrB.Length();
25            P_DivideSort(b_buf, b_size);
26            PtrB.RemoteSet();
27         }
28      }
29   }

                    ⋮
```

Figure 6.4: Parallel divide-and-conquer sort

	Processes				
Time (secs.)	1	2	3	4	5
User	17.7	16.2	8.4	16.0	16.2
System	0.2	0.1	1.4	4.2	0.8

Table 6.1: Sort statistics

system provided a way to block the process until a message is received. In REM, the only facility provided is a wait routine which, once entered, cannot be exited safely. It is interesting to note that the children take as long as the sequential version which means that the communication costs equals the time saved by sorting only half of the buffer.

Table 6.2 contains the elapsed execution time. The distributed version of the algorithm is slightly slower than the sequential version (i.e., Process 3 vs. Process 1). That was to be expected due to the communication costs of the network and the efficiency of the sequential divide-and-conquer algorithm. Since there is an improvement in the time used on the parents node, a programmer could reduce the load on a particular workstation by spreading around the work of the algorithm. This improvement may justify the time spent to create a distributed version, because it took very little effort to produce. If the programmer is forced to use a message passing system to implement the algorithm, the improvement could probably not be justified, because of the time required to implement such an algorithm.

	Processes		
Time (secs.)	1	2	3
Real	39.9	35.5	137.4
User	35.4	33.6	26.1
System	1.2	0.9	2.9

Table 6.2: Program statistics

6.3 Fault-tolerant ray tracer

A distributed ray tracer which has been developed by Brian Corrie at the University of Victoria was selected as a major test case for our distributed programming methodology[Cor90]. By using a program which had previously been distributed by message passing, a subjective comparison can be made between the difficulties encountered in creating the message passing version and the version developed with our methodology. The new distributed ray tracer was implemented using shared data-objects, process objects, and hand preprocessing. When initially writing the new version, the original distributed ray tracer was used as a template. However, this was quickly abandoned, because the design was not compatible with shared data-objects which simulate shared memory. Instead, the sequential version of the ray tracer was selected and was found to be a much better guide for creating the new distributed version.

The sequential version of the ray tracer was transformed in about four hours to a parallel program (two hours were spent making the C code in the ray tracer compatible with C++). The sequential description of the ray tracer is depicted in Figure 6.5. The two-hour programming time for the new version compared favorably with an estimated time of two days for the message passing version. To upgrade

	Version		
Time	1	2	3
User	6369	1202.74	1286.36

Table 6.3: Ray tracer test statistics

the ray tracer to include a fault-tolerant algorithm required only an additional two hours. In Table 6.3, Version 1 shows the elapsed time required to generate a recursive tetrahedron on a single Sun 3/60 equivalent machine. The distributed ray tracer using message passing (Version 2) achieved a speed up of 5.3 for 8 processors. The distributed ray tracer using the shared data objects (Version 3) achieved a speed up of 4.95 which is a 7% reduction in performance.

The `OfflineProcess` receives three `RemoteInts` from the parent process, `Line`, `Init[j]`, and `Result[j]`. `Init[j]` signals to the `OfflineProcess` whether the ray tracer database is initialized. It is set to 0 when the `OfflineProcess` is initially created. After the `OfflineProcess` initializes the database, it sets the flag to one. On subsequent startups the `OfflineProcess` does not load the database, because the value it receives from the parent is one. `Line` denotes the current scanline. The line number is written to `Result[j]` when the `OfflineProcess` is finished. When the parent retrieves the line number from `Result[t]`, it knows that the scanline has been successfully completed. After the last scanline is complete, the parent then goes through the list of scanlines to make sure that they are all completed. If a scanline is not complete, it is sent back out to an `OfflineProcess`.

The experience gained from writing the distributed ray tracer indicated that the distributed version is only slightly more complex to write than the sequential version. Minor problems such as initializing remote processes have to be handled carefully so

that the program can run sequentially or in a distributed fashion. Incorporating the fault tolerance showed that programs designed with this method are as extensible as sequential programs, and more so than message-based programs with embedded send and receive statements.

```
                    ⋮

 1   Line.Set(current_line);
 2   fprintf(stderr,"Doing line %d\n", current_line);
 3   Results[j].Set(-1);
 4   {
 5      OfflineProcess child("child", j);
 6      if (Init[j].Access() == 0) {
 7         sprintf(errfile,"%s%d", (char*)ErrFile.Address(), Line.Access());
 8         sprintf(infile,"%s", (char*)InFile.Address());
 9         sprintf(outfile,"%s%d", (char*)OutFile.Address(), Line.Access());
10         if (CallInit) {
12            InitDefaults();
13            InitError(errfile);
14            InitScene();
15            InitNoise();
16         }
17         openScreen(outfile);
18         Init[j].Set(1);
19      }
20      blip(Line.Access());
21      doScanLine(Line.Access());
22      Results[j].Set(Line.Access());
23      child.SetTerminate(Cleanup);
24   }

                    ⋮
```

Figure 6.5: Remote process declaration.

Chapter 7

Conclusions

Through the use of the shared data-driven object model it is possible to specify a program in a sequential manner and to execute it on sequential and parallel systems. This thesis has discussed, and then shown the potential for using shared data-objects and process objects to control parallel execution. The preprocessor enables parallel programs to be generated automatically which execute exactly as described by the sequential description.

7.1 Summary of results

A major emphasis of this thesis was to provide an environment to develop programs which would function correctly on a variety of architectures while minimizing the work. This resulted in the shared data-driven model which combined the features of object-oriented programming with those of data-flow analysis. This model allows the generation of parallel programs that look and feel like sequential programs. For a programmer who is used to dealing with remote processes and message passing, the

in-line declaration of child processes may seem strange. For a sequential programmer, who is familiar with procedures and in-line declarations, this concept is more natural than defining and coordinating separate processes.

Using information generated from the activation tree enabled the distributed data and control flow to be implicitly stated in the program. Because the flow is stated in this fashion, data exchange, synchronization, and mutual exclusion is obtained by the use of shared data-objects. The use of process objects simplifies the creation, execution, and termination of processes and allows the program to execute without deadlock or starvation. Varying communication patterns can be achieved by declaring shared data-objects in different scopes. The whole system is compatible with different communication paradigms, because the exchange of information is based on a simple message passing scheme. Finally, since the programs are described sequentially the programs can be executed on various computational models. The shared data-driven object model addresses the deficiencies found in many of the current systems.

One of the main contributions of this thesis is the amalgamation of ideas from several disciplines to form a model that presented a consistent environment in which to program sequential programs in a distributed fashion. The use of the model allows programs to benefit from parallel execution without substantially increasing the burden of programming. Programmers who are not familiar with distributed programming but have a problem that lends itself well to coarse grained parallelization, will particularly benefit from this approach.

The developed examples using the shared data-driven model produced several interesting insights. It is not sufficient to write a sequential program and expect it to run efficiently in parallel. For the program to execute correctly, consideration must be given to what happens to shared data-objects when remote processes execute. A poorly designed program will run, but encounters a reduction in performance when

executed in parallel. When analyzing a program for potential parallelism or syntactical errors, a single program is easier to understand. By looking at the problem in a step-by-step manner, it is possible to ignore the nondeterministic order of execution of children, thus reducing the complexity of the task. When a program is altered, the freedom to pass information by simply declaring an object is helpful in the development and the debugging process.

7.2 Future research

The set of problems which is addressed by the shared data-driven object model is restricted to problems that can be described sequentially. There are many issues addressed by parallel programming which do not fit into this category but would be promising to explore in the future. Also, there are various optimization issues that would improve the performance of the resulting distributed programs.

1. Develop an optimizing preprocessor that identifies access patterns of objects to improve the data analysis. This would aid in minimizing the message passing to keep communication costs down. It would also maintain the minimal number of locks to be set on shared data variables which would allow the maximum amount of parallelization.

2. Investigate how an implementation can dynamically use features implemented in the communication system more effectively without losing generality. For example, the load sharing within REM determines the number of stations that can be used. This number could be fed back to the process which then tailors its algorithm to the available processors. Parallel algorithms such as the parallel

selection presented by Akl requires the actual number of processes to determine the optimum partition of the data [Akl89].

3. Instead of statically naming variables, it should also be possible to dynamically name them at run time. This would be analogous to dynamic linking of messages to objects in languages like Objective-C or Smalltalk-80. This would enable the creation of a service process that would communicate through a predefined set of shared data-objects. If a program could dynamically link locally defined shared data-objects with remote shared data-objects currently defined in an active process, then process relationships such as server/client, coroutine, and remote calls could be implemented.

4. Extend the data analysis to include pointers, variables, and objects that are not defined as shared data-objects.

Bibliography

[Ahu86] S. Ahuja. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[Akl89] S.G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison-Wesley, Wokingham, England, 1986.

[Ban79] J.P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Symposium on Principles of Programming Languages '79.* ACM, October 1979.

[BT88] H. Bal and A.S. Tanenbaum. Distributed programming with shared data. In *Int'l Conference on Computer Languages.* IEEE, October 1988.

[CG86] K.L. Clark and S. Gregory. Parlog: Parallel programming in logic. *ACM Trans. Program. Lang. Syst.*, 8(1):1–49, January 1986.

[Che85] D.R. Cheriton. Preliminary thoughts on problem-oriented shared memory: A decentralized approach to distributed systems. *Operating Systems Reviews*, 19(4):26–33, October 1985.

[CM88] K.M. Chandy and J. Misra. *Programming Specification, Unity.* MIT Press, Cambridge, Massachusetts, 1988.

[Cor90] B. Corrie. A graphics workbench for realistic image synthesis. Master's thesis, Department of Computer Science, Univeristy of Victoria, April 1990.

[DoD83] United States of America Department of Defense. *Reference Manual for the ADA Programming Language*, Febuary 1983.

[Gel85] D. Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, January 1985.

[Gre87] S. Gregory. *Parallel Logic Programmin in PARLOG.* Addison-Wesley, Wokingham, England, 1987.

[Han78] P.B. Hansen. Distributed processes: A concurrent programming concept. *Commun. ACM*, 21(11):934–941, November 1978.

[Hoa74] C.A.R. Hoare. Monitors: An operating system structuring concept. *Commun. ACM*, 17(10):549–557, October 1974.

[Ken81] K. Kennedy. A survey of data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis*, pages 5–54. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[KR78] B. Kernighan and D. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[Li86] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, New Haven. CT, 1986.

[Mey88] B. Meyer. *Object-oriented Software Construction.* Prentice-Hall, 1988.

[MU89] H.A. Müller and J.S. Uhl. Discovering and reconstructing software structure using graph filters. Technical Report DCS-110-IR, University of Victoria, March 1989.

[SCT87] G.C. Shoja, G. Clarke, and T. Taylor. Rem: A distributed facility for utilizing idle processing power of workstations. In *WG 10.3 Working Conference on Distributed Processing.* IFIP, October 1987.

[Sho88] G.C. Shoja. A distributed facility for load sharing and parallel processing among workstations. Technical Report DCS-95-IR, University of Victoria, March 1988.

[Str86] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[Vuo89] S.T. Vuong, editor. *Forte '89 Tutorial Notes.* ACM, December 1989.

[Weg87] P. Wegner. Dimensions of object-based language design. In *OOPSLA '87 Proceedings.* ACM, October 1987.

[WG89] M. Wu and D.D. Gajski. Hypertool: A programming aid for multicomputers. In *Int'l Conference on Parallel Processing.* IEEE, August 1989.

[Whi87] D. Whiddett. *Concurrent Programming.* Ellis Horwood Limited, Chichester, West Sussex, 1987.

[YT87] Y. Yokote and M. Tokoro. Concurrent programming in concurrent-smalltalk. In Y. Yokote and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 129–15. MIT Press, Cambridge, Massachusetts, 1987.

Appendix A

Class library

A.1 Shared data-object services

ReadDataAccess() Block process until it can read from the object.

WriteDataAccess() Block process until it can write to the object.

WriteUnlock() Do not block any further write accesses.

ReadUnlock() Do not block any further read accesses.

A.2 Process services

OfflineWaiting(OfflineID) Returns TRUE if the offline process is waiting to start execution or does not exist. Returns FALSE if the process is currently active.

Appendix B

Divide-and-conquer sort

B.1 Parallel sort algorithm

Sequential version

```
// This may look like C code, but it is really -*- C++ -*-

#include <stream.h>
#include <sys/time.h>
#include <sys/resource.h>

#include "CCheader.h"
#include "data.h"
#include "jobs.h"

extern "C" {
    int rand();
    void srand(int);
}

extern void GenerateData(int*, int);
extern void P_DivideSort(int*, int);
extern void DivideSort(int*, int);
```

```
extern void Partition(int*, int, int, int&, int&);
extern void Swap(int*, int, int);
extern char* fmttime(struct timeval*, struct timeval*);

int const MAX_NUMBER = 100000;
int const THRESHOLD = MAX_NUMBER - 1;

main() {
    struct rusage time1, time2; // used for timing

    int buffer[MAX_NUMBER];
    int seed;

    seed = 126;
    srand(seed);

    GenerateData(buffer, MAX_NUMBER);

    getrusage( 0 /*RUSAGE_SELF */, &time1);

    P_DivideSort(buffer, MAX_NUMBER);

    getrusage(0 /* RUSAGE_SELF */, &time2);
    cerr << "user time = ";
    cerr << fmttime( &time2.ru_utime, &time1.ru_utime);
    cerr << "\n";
    cerr << "system time = ";
    cerr << fmttime( &time2.ru_stime, &time1.ru_stime);
    cerr << "\n";

    for(int i = 0;i < MAX_NUMBER;i++) cout << buffer[i] << " ";

}

void GenerateData(int* buf, int size) {
    for(int i = 0; i < size; i++, buf++) {
        *buf = rand() % MAX_NUMBER;
    }
}

void P_DivideSort(int* buf, int length) {
    int left = 0;
```

```
    int start, finish;

    if(left + THRESHOLD >= length) {
        DivideSort(buf, length);
    }
    else {
        Partition(buf, left, length, start, finish);
        int* left_part = buf + left;
        int* length_part = buf + start;
        RemoteBuffer PtrA((finish-left+1)*sizeof(int),
                  (void*)left_part);
        RemoteBuffer PtrB((length-start)*sizeof(int),
                  (void*)length_part);
        {
            InlineProcess child1("child1");

            int* a_buf = (int*) PtrA.Address();
            int  a_size = PtrA.Length()/sizeof(int);
            P_DivideSort(a_buf, a_size);
            PtrA.RemoteSet();
        }

        {
            InlineProcess child2("child2");

            int* b_buf = (int*) PtrB.Address();
            int  b_size = PtrB.Length()/sizeof(int);
            P_DivideSort(b_buf, b_size);
            PtrB.RemoteSet();
        }


    }
}

void DivideSort(int* buf, int length) {
    int left = 0;
    int start, finish;

    if(left + 1 > length) {
        if(*(buf+left) > *(buf+length))
            Swap(buf, left, length);
```

```
    }
    else if(left + 1 < length) {
        Partition(buf, left, length, start, finish);
        DivideSort((buf+left), (finish-left+1));
        DivideSort((buf+start), (length-start));
    }
}

void Partition(int* buf, int left, int length, int& start, int& finish) {


    int pivot = *(buf + (int)(length/2));
    start = left;
    finish = length - 1;

    do {
        while(*(buf+start) < pivot) start++;
        while(*(buf+finish) > pivot) finish--;
        if(start <= finish) {
            Swap(buf, start, finish);
            start++;
            finish--;
        }
    } while(start <= finish);
}


void Swap(int* buf, int first, int second) {
    int tmp;
    tmp = *(buf+first);
    *(buf+first) = *(buf+second);
    *(buf+second) = tmp;
}



char *fmttime(struct timeval *to, struct timeval *from) {
   long diffs, diffu;
   static char inbuff[80];
   diffs = to->tv_sec - from->tv_sec;
   diffu = to->tv_usec - from->tv_usec;
   if( diffu < 0 ) {
```

```
    diffu += 1000000;
    diffs -= 1;
    }
    sprintf(inbuff, "%d.%06d", diffs, diffu);
    return(inbuff);
}
```

Parent process

```
// This may look like C code, but it is really -*- C++ -*-

#include <stream.h>
#include <sys/time.h>
#include <sys/resource.h>

#include "CCheader.h"
#include "data.h"
#include "jobs.h"
#include "rembuffer.h"


extern "C" {
    int rand();
    void srand(int);
}


extern void GenerateData(int*, int);
extern void P_DivideSort(int*, int);
extern void DivideSort(int*, int);
extern void Partition(int*, int, int, int&, int&);
extern void Swap(int*, int, int);
extern char* fmttime(struct timeval*, struct timeval*);

int const MAX_NUMBER = 100000;
int const THRESHOLD = MAX_NUMBER - 1;

main() {
    fprintf(stderr, "[main] Starting main.\n");
```

```
    me.Initialize();
}

void ParentProcess::Startup() {
}


void ParentProcess::Execute() {
    struct rusage time1, time2; // used for timing

    int buffer[MAX_NUMBER];
    int seed;

    seed = 126;
    srand(seed);

    GenerateData(buffer, MAX_NUMBER);

    getrusage(0 /*RUSAGE_SELF */, &time1);

    P_DivideSort(buffer, MAX_NUMBER);

    getrusage(0 /* RUSAGE_SELF */, &time2);
    cerr << "user time = ";
    cerr << fmttime( &time2.ru_utime, &time1.ru_utime);
    cerr << "\n";
    cerr << "system time = ";
    cerr << fmttime( &time2.ru_stime, &time1.ru_stime);
    cerr << "\n";

    for(int i = 0;i < MAX_NUMBER;i++) cout << buffer[i] << " ";

}

void ParentProcess::Execute(int, char**) {

}

void P_DivideSort(int* buf, int length) {
    int left = 0;
    int start, finish;
```

```
    if(left + THRESHOLD >= length) {
        DivideSort(buf, length);
    }
    else {
        Partition(buf, left, length, start, finish);

        int* left_part = buf + left;
        int* right_part = buf + start;
        int  left_length = (finish-left+1)*sizeof(int);
        int  right_length = (length-start)*sizeof(int);

        RemoteBuffer PtrA(left_length, (void*)left_part);
        RemoteBuffer PtrB(right_length, (void*)right_part);

        InlineProcess child1("child1");
        InlineProcess child2("child2");

        PtrA.PackageAddress(child1);
        me.Start(child1);

        PtrB.PackageAddress(child2);
        me.Start(child2);
        }
}

/*  See Sequential version for remainder of the program. */
```

First child

```
// This may look like C code, but it is really -*- C++ -*-
#include <stream.h>
#include <sys/time.h>
#include <sys/resource.h>

#include "CCheader.h"
#include "data.h"
#include "jobs.h"
#include "rembuffer.h"
```

```
extern "C" {
        int rand();
        void srand(int);
}


extern void GenerateData(int*, int);
extern void P_DivideSort(int*, int);
extern void DivideSort(int*, int);
extern void Partition(int*, int, int, int&, int&);
extern void Swap(int*, int, int);
extern char* fmttime(struct timeval*, struct timeval*);
extern void statistics();

int const MAX_NUMBER = 100000;
int const THRESHOLD = MAX_NUMBER - 1;

struct rusage time1, time2; // used for timing

main(int argc, char* argv[]) {
        me.SetTerminate(statistics);
        getrusage(0 /*RUSAGE_SELF */, &time1);
        me.RemoteInitialize(argc, argv);
}

void ParentProcess::Startup() {
}

void ParentProcess::Execute() {

        me.OldVariables();
        RemoteBuffer PtrA;
        me.NewVariables();

        int* a_buf = (int*) PtrA.Address();
        int  a_size = PtrA.Length()/sizeof(int);

        P_DivideSort(a_buf, a_size);

        PtrA.RemoteSet();

}
```

```
void ParentProcess::Execute(int, char**) {
}

void P_DivideSort(int* buf, int length) {
        int left = 0;
        int start, finish;

        if(left + THRESHOLD >= length) {
                DivideSort(buf, length);
        }
        else {
                Partition(buf, left, length, start, finish);
                int* left_part = buf + left * sizeof(int);
                int* length_part = buf + start * sizeof(int);
                RemoteMemory PtrA((finish-left+1), (void*)left_part);
                PtrA.Salloc();
                RemoteMemory PtrB((length-start), (void*)length_part);
                PtrB.Salloc();

                InlineProcess child1("child1");
                PtrA.PackageAddress(child1);
                me.Start(child1);

                InlineProcess child2("child2");
                PtrB.PackageAddress(child2);
                me.Start(child2);

        }

}

/*  See Sequential version for remainder of the program. */
```

Second child

```
// This may look like C code, but it is really -*- C++ -*-
#include <stream.h>
#include <sys/time.h>
#include <sys/resource.h>
```

```
#include "CCheader.h"
#include "data.h"
#include "jobs.h"
#include "rembuffer.h"

extern "C" {
        int rand();
        void srand(int);
}


extern void GenerateData(int*, int);
extern void P_DivideSort(int*, int);
extern void DivideSort(int*, int);
extern void Partition(int*, int, int, int&, int&);
extern void Swap(int*, int, int);
extern char* fmttime(struct timeval*, struct timeval*);
extern void statistics();

int const MAX_NUMBER = 100000;
int const THRESHOLD = MAX_NUMBER - 1;

struct rusage time1, time2; // used for timing


main(int argc, char* argv[]) {
        me.SetTerminate(statistics);
        getrusage(0 /*RUSAGE_SELF */, &time1);
        me.RemoteInitialize(argc, argv);
}

void ParentProcess::Startup() {
}

void ParentProcess::Execute() {

        me.OldVariables();
        RemoteBuffer PtrB;
        me.NewVariables();
```

```
        int* b_buf = (int*) PtrB.Address();
        int  b_size = PtrB.Length()/sizeof(int);


        P_DivideSort(b_buf, b_size);

        PtrB.RemoteSet();

}

void ParentProcess::Execute(int, char**) {
}

void P_DivideSort(int* buf, int length) {
        int left = 0;
        int start, finish;

        if(left + THRESHOLD >= length) {
                DivideSort(buf, length);
        }
        else {
                Partition(buf, left, length, start, finish);
                int* left_part = buf + left;
                int* length_part = buf + start;
                RemoteMemory PtrA((finish-left+1), (void*)left_part);
                RemoteMemory PtrB((length-start), (void*)length_part);

                InlineProcess child1("child1");
                PtrA.PackageAddress(child1);
                me.Start(child1);

                InlineProcess child2("child2");
                PtrB.PackageAddress(child2);
                me.Start(child2);

        }

}

/*  See Sequential version for remainder of the program. */
```

Appendix C

Preprocessing Example

C.1 Sequential description

```
RemoteInt a;

main() {
    RemoteInt b;

    b.Set(5);
    {
        InlineProcess child1("child1");
        RemoteInt c;

        c.Set(b.Access());
        {
            InlineProcess child2("child2");
            RemoteInt d;

            d.Set(5);
            a.Set(c.Access() + d.Access());
        }
    }
    b.Set(a.Access());
}
```

C.2 Trace of preprocessing algorithm

Trace is only until child2 is written

```
Call to algorithm A :
  Sets generated :
      GLOBAL = {a}
      LOCAL[] = {}

Call to Transform_Block(): for main
  Parameters :
      GLOBAL = {a}
      LOCAL[] = {}
      CODE[main]
  Sets generated :
      ID = ''''
      LOCAL[main] = {b}
      MOD[main] = {a,b}
      USED[main] = {a,b}

Call to Transform_Block(): in main for child1
  Parameters :
      GLOBAL = {a}
      LOCAL[main] = {b}
      CODE[child1]
  Sets generated :
      ID = ''child1''
      LOCAL[child1] = {b,c}
      MOD[child1] = {a,c}
      USED[child1] = {b,c}

Call to Transform_Block(): in child1 for child2
  Parameters :
      GLOBAL = {a}
      LOCAL[child1] = {b,c}
      CODE[child2]
  Sets generated :
      ID = ''child2''
```

```
      LOCAL[child2] = {b,c,d}
      MOD[child2] = {a,c}
      USED[child2] = {c,d}

Call to Replace_Block(): in child1 for block child2
  Parameters :
      GLOBAL = {a}
      LOCAL[child1] = {b,c}
      USED[child2] = {c,d}
      MOD[child2] = {a,c}
      ID = ''child2''
  Write out to child1 :
      a.SetWriteBlock(child2);
      c.SetWriteBlock(child2);
      c.PackageAddress(child2);

Call to Generate_Child(): in child1 for block child2
  Parameters :
      GLOBAL = {a}
      LOCAL[child1] = {b,c}
      LOCAL[child2] = {b,c,d}
      USED[child2] = {c,d}
      MOD[child2] = {c,a}
      filename = ''child2''
      CODE[child2]
  Write out to child2 :
      me.OldVariables();
      RemoteInt c;
      me.NewVariables();
      RemoteInt d;
      a.RemoteGet();

      print out --- CODE[child2]

      a.RemoteSet();
      c.RemoteSet();
```

Parent file

```
#include ''CCheader.h''
```

```
#include ''data.h''
#include ''jobs.h''
#include ''rembuffer.h''

RemoteInt a;
main() {
    me.Initialize();
}
void ParentProcess::Startup() {
}
void ParentProcess::Execute() {
    me.NewVariables();
    RemoteInt b;
    InlineProcess child1(''child1'');

    a.SetWriteBlock(child1);
    b.PackageAddress(child1);
    me.Start(child1);

    b.ReadDataAccess();
    a.WriteDataAccess();
    b.Set(a.Access());
}

void ParentProcess::Execute(int,char**) {
}
```

Child1 file

```
#include ''CCheader.h''
#include ''data.h''
#include ''jobs.h''
#include ''rembuffer.h''

RemoteInt a;
main(int argc,char* argv[]) {
    me.RemoteInitialize(argc,argv);
}
void ParentProcess::Startup() {
}
```

```
void ParentProcess::Execute() {
    me.OldVariables();
    RemoteInt b;
    me.NewVariables();
    RemoteInt c;

    c.Set(b.Access());
    InlineProcess child2(``child2'');

    a.SetWriteBlock(child2);
    c.SetWriteBlock(child2);
    c.PackageAddress(child2);
    me.Start(child2);

    a.WriteDataAccess();
    a.RemoteSet();
}

void ParentProcess::Execute(int,char**) {
}
```

Child2 file

```
#include ``CCheader.h''
#include ``data.h''
#include ``jobs.h''
#include ``rembuffer.h''

RemoteInt a;
main(int argc,char* argv[]) {
    me.RemoteInitialize(argc,argv);
}
void ParentProcess::Startup() {
}
void ParentProcess::Execute() {
    me.OldVariables();
    RemoteInt c;
    me.NewVariables();
    RemoteInt d;
```

```
    d.Set(5);
    a.Set(c.Access() + d.Access());

    a.RemoteSet();
}

void ParentProcess::Execute(int,char**) {
}
```

VITA

Surname: **Sinclair** Given Names: **Craig**
Place of Birth: **Regina, Saskatchewan** Date of Birth: **August 28, 1959**

Educational Institutions Attended:

University of Calgary	1977 to 1984
University of Victoria	1988 to 1990

Degrees Awarded:

B.Comm	1982	University of Calgary, Alberta
B.Sc.	1984	University of Calgary, Alberta

Partial Copyright License

Title of Thesis:

Automatic Parallelization of C Programs Using Shared Data-Objects

Author: [redacted]
Craig Sinclair
April 25, 1990