4-10-1995

Object-oriented concurrent programming on the connection machine with COOL (Concurrent Object-Oriented Language)

Maria Rosa Drake
Florida International University

DOI: 10.25148/etd.FI15101225

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

OBJECT-ORIENTED CONCURRENT PROGRAMMING ON THE CONNECTION MACHINE WITH COOL (Concurrent Object-Oriented Language)

A thesis submitted in partial satisfaction of the

requirements for the degree of

MASTER OF SCIENCE

IN

IN COMPUTER SCIENCE

by

Maria Rosa Drake

1995

To: Arthur W. Herriott
College of Arts and Sciences

This thesis, written by Maria Rosa Drake, and entitled Object-Oriented Programming on the Connection Machine with COOL (Concurrent Object-Oriented Language), having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this thesis and recommend that it be approved.

Yi Deng

Masoud Milani

Raimund K. Ege, Major Professor

Date of Defense: April 10, 1995

The thesis of Maria Rosa Drake is approved.

Dean Arthur W. Herriott
College of Arts & Sciences

Dean Richard Campbell
Division of Graduate Studies

Florida International University, 1995

DEDICATION

I dedicate this thesis to my parents, Luis and Rosy Drake for their love, patience, and encouragement. To my sister Maria Luisa and brother Frederick, and their families, for understanding that ‘thesis time’ came first. To my grandmother Rosa and my aunts Angelines and Mimi for their constant encouragement. Finally, to dedicate this thesis to my very good friend Julio Ibarra for his constant support and encouragement

ACKNOWLEDGMENTS

First and foremost I want to express my deepest appreciation and heartfelt thanks to Dr. Paul Gallagher, Florida International University, for his unwavering support and interest in my project.

I want to thank Josh Simons, Thinking Machines, for all his help in providing technical assistance and material on the Connection Machine and C^*; Robert Whaley, Thinking Machines, for answering many questions about C^* and the facilities at the Naval Research Lab in Washington.

I would also like to thank all the people at IRM for their support and understanding throughout this project. Special thanks go to Jeff Reisberg for being there during the home stretch.

Finally, a very special thanks to my major professor, Dr. Raimund K. Ege, for all his help, support and patience thoughout this project.

ABSTRACT OF THE THESIS

OBJECT-ORIENTED CONCURRENT PROGRAMMING ON THE CONNECTION MACHINE WITH COOL

(Concurrent Object-Oriented Language)

by

Maria Rosa Drake

Florida International University, 1995

Miami, Florida

Professor Raimund K. Ege, Major Professor

The quest for speed and the need to solve ever more complex problems has led to the development of powerful computer systems, such as the Connection Machine. Concurrent processing promises a solution to the problem. COOL (Concurrent Object-Oriented Language) has been developed in order to provide the Connection Machine with a subset of C^{++} which includes several concurrent constructs. The Connection Machine has an inherently parallel architecture which can be taken advantage of with software.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

According to Ted G. Lewis [LR92] "Parallel computing is one of the most exciting technologies to achieve prominence since the invention of electronic computers in the 1940s".

The demand for computers has risen dramatically; in addition to the number of computers in use at this time, the demand is increasing exponentially for faster and better systems. Not only is the amount of memory and mass storage important, but the speed of the processor(s) involved in computations is just as important, if not more so. In addition, the current programming trend is moving toward object-oriented languages where inheritance, encapsulation, abstraction, information hiding and reusability are becoming prime factors for software development.

For the most part, computer systems which are widely available today are based on a single processor and are known as single instruction single data (SISD) systems [Dun90]. Yet as processor prices decrease, we are beginning to see more systems with more than one processor. By having systems with more processors, performance could significantly improve. The Connection Machine by Thinking Machines is such a system. The Connection Machine can have 65,536 physical processors, and an unlimited number (subject to memory limitations) of virtual processors [CM2].

Object-oriented languages were first developed to improve software reuse and quality. Software quality improves by making software easier to maintain. Inheritance reduces the amount of coding, yet does not increase its complexity. Modularity allows for quick location of code segments. Software quality also improves by allowing software developers, via inheritance, to add new functionality without altering working software [Lan87]. Due to the qualities found in object-oriented languages, more and more programmers are now using them.

Programming languages have evolved from assembly languages in the 1950s to procedure-oriented languages in the 1960s, structured programming in the 1970s, distributed, functional and relational paradigms in the 1980s, and objected-oriented languages in the 1990s. Object-oriented programming is like structured programming was in the 1970s: it is becoming a part of *standard* programming practice. [Weg90]

2. WHAT IS COOL?

The purpose of this thesis is to develop an extension to C* 6.0 (a parallel programming language for the Connection Machine) which provides object-oriented constructs such as classes, inheritance, modularity, information hiding, encapsulation and reusability. This language, COOL (Concurrent **O**bject-**O**riented **L**anguage) will augment

C^* 6.0 with a subset of the functionality of C^{++} and will execute on the Connection Machine 2 (CM-2) and the Connection Machine 5 (CM-5).

COOL has a concurrent object-oriented model, yet it's execution model is data parallel. COOL maps objects onto processors. Objects become active when they receive a request message from another object; therefore, there is a means of synchronization built into the language. COOL supports standard C as well as the parallel features and constructs found in C^* 6.0. Most important of all, COOL adds object-oriented constructs to C^* 6.0, thus becoming one of the few, if not the only, object-oriented language to be used on the Connection Machine family of computers which takes advantage of the features found in C^* 6.0.

The C^* language, developed by Thinking Machines for use on their Connection Machine, is a data parallel extension of the ANSI C programming language. C^* programs are similar to standard C programs. Parallel code looks like serial code, but is executed simultaneously in all parallel processors [CM2].

C^{++} is an enhanced version of the popular C programming language which was developed at AT&T Bell Laboratories by Bjarne Stroustrup. C^{++} provides object-oriented constructs to C, such as classes, inheritance, information hiding, and reusability.

3. CONCEPTS AND TERMINOLOGY

Various concepts and terms need to be explained to understand COOL and its functionality.

3.1 Parallel Programming

Parallel Programming is the ‘new wave of computing’ [LR92]; it allows programs to perform tasks concurrently. There are two major paradigms in parallel programming: synchronous and asynchronous. See Figure 1.

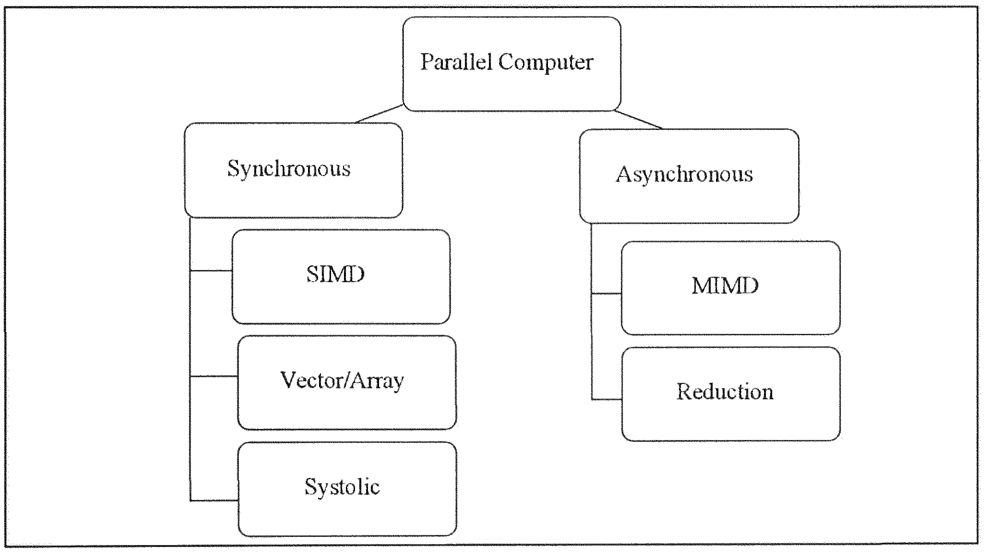

Figure 1 : Parallel Computing Paradigms [LR92]

1- *Synchronous* implies lockstep coordination in hardware by forcing tasks to be performed at the same time. Three of the major forms of synchronous parallel computing include:

- *Single Instruction Multiple Data* (SIMD) involves multiple processors simultaneously executing the same instruction on different data. For example, all processors would execute

  ```
  a=b+c
  ```

  at the same time.

- *Vector/Array* uses a pipeline approach where each processor is working on a task, yet tasks are different among processors. For example, bank tellers may organize themselves into a coordinated assembly line of workers where each teller would perform a specific task: Teller 1 would get the customer's number; teller 2 uses the account number to validate the customer; teller 3 updates the account by posting a transaction and teller 4 would take or return cash to the customer. The pipeline effect comes when teller 1 passes its results to teller 2; teller 2 to teller 3 and finally teller 3 to teller 4. [LR92]

- *Systolic* paradigm incorporates features of SIMD and the pipeline paradigm. It is called systolic because of the tighter coordination of processors. This paradigm addresses performance requirements of special-purpose systems by achieving significant parallel computational and by avoiding I/O and memory bandwidth bottlenecks. [Dun90] To carry out the bank analogy, we reorganize bank tellers as a two-dimensional array and circulate account folders from teller to teller with a minimum of memory accesses. [LR92]

2.- *Asynchronous* implies there is no lockstep mechanism and process coordination is done in software. This is the more general form of parallelism. Two of the major forms of asynchronous parallel computing are:

- *Multiple Instruction Multiple Data* (MIMD) involves multiple processors autonomously executing diverse instructions on diverse data.

- *Reduction*, or demand driven architecture, is based on the graph reduction model. This is achieved with reducible expressions which are replaced by their computed value as computation progresses. Reduction implements an execution paradigm in which an instruction is enabled for execution when it's results are required as operands for another instruction already enabled for execution. [Dun90]

3.2 Concurrent Programming

Concurrent programming differs from parallel programming in that it allows for asynchronous events to take place simultaneously while parallel programming allows events to take place simultaneously, yet in lockstep fashion. Concurrent programming implies multiple independent activities within a system, including control and communication. [KL89]

3.3 Object-Oriented Programming

Object-oriented programming is one of the current paradigms in programming languages. Object-oriented programming languages focus on *objects*, their behavior, and their collaboration. This programming approach came about because it is more natural for humans to think in terms of *objects*, and how they relate and interact with one another.

3.3.1 Object-Oriented Programming Concepts

Several concepts need to be explained in order to understand how COOL works.

3.3.1.1 Class

A class may be considerd as a template from which new objects may be created. Classes include a method (member function) for each type of operation its instances can perform. Objects of the same class have common operations. Every object is an instance of some class.

3.3.1.2 Object

Objects are autonomous entities that respond to messages by executing methods or operations. Each object has a set of operations and a state that remembers the effect of the operations. An object's private properties include the set of instance variables that make up it's private memory and the set of methods, described by a class, which describe how to carry out operations [Weg90].

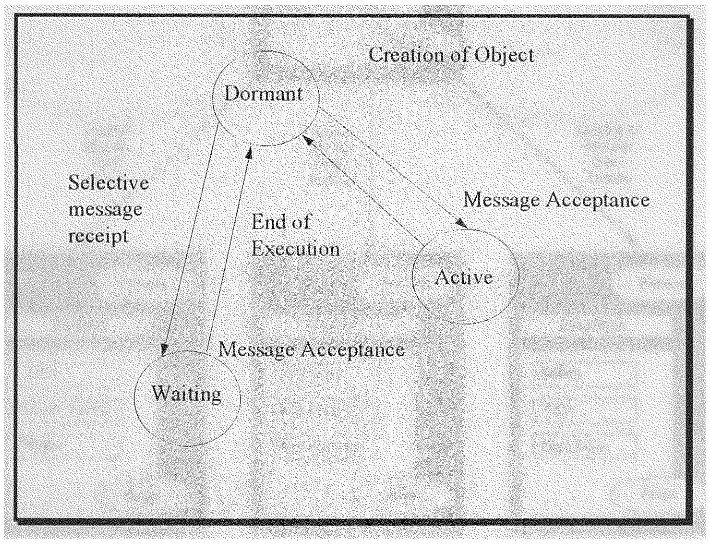

Figure 2: Object

3.3.1.3 Inheritance

Inheritance is a mechanism for building new classes from other existing class types therefore creating a more specialized class from a broader existing class.

Inheritance, or class derivation, allows for organizing, building, and using reusable classes. It makes it possible to define new classes based on existing classes, therefore creating a hierarchical collection of classes; therefore, classes may 'inherit' operations from a base class or super class. See Figure 3.

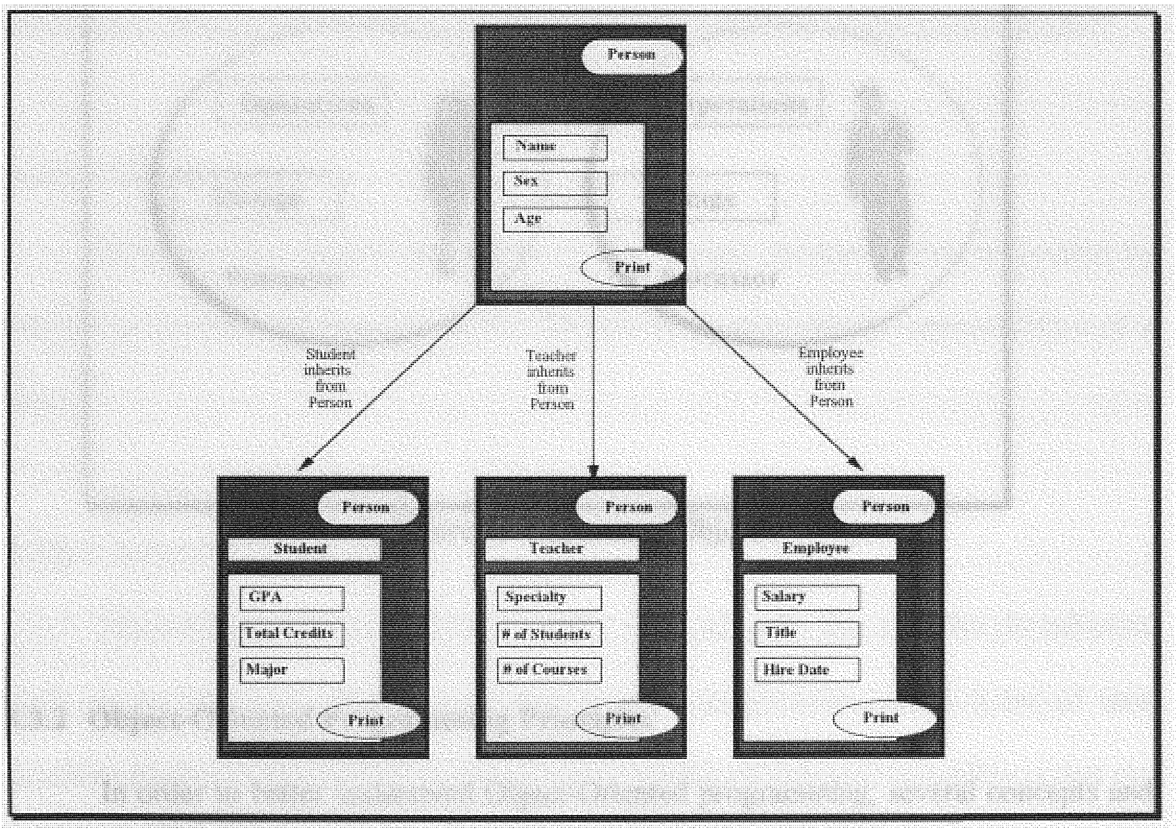

Figure 3: Inheritance

3.3.1.4 Concurrent Objects

A concurrent object [TB90] consists of operations, local storage and a (virtual) processor. Concurrent objects become active when receiving a request message. Concurrent objects clearly model the real world. Each object executes different code concurrently. Figure 4 illustrates objects representing a man and a woman; where each object may be executing different code based on their job.

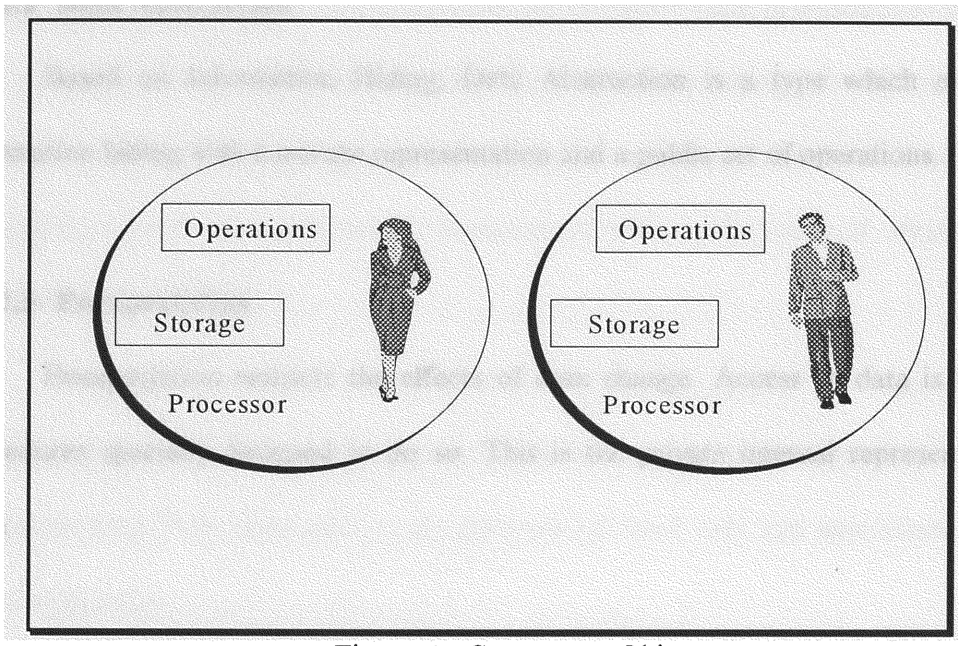

Figure 4: Concurrent Objects

3.3.2 Object-Oriented Programming Principles

In order to better understand Object-Oriented programming, several concepts and principles will be explained.

3.3.2.1 Information Hiding

As a rule, the representation of a class is private, yet the fields can be public. This private/public specification is known as information hiding. Information hiding is a formal mechanism for restricting user access to the internal representation of a class type.

3.3.2.2 Data Abstraction

Based on Information Hiding, Data Abstraction is a type which encompasses information hiding with a private representation and a public set of operations. [Bud91]

3.3.2.3 Encapsulation

Encapsulation restricts the effects of data change. Access to data is handled by procedures specially designed to do so. This is the private internal representation of a class.

3.3.2.4 Software Reusability

When behavior is inherited from another class, the code that provides the behavior need not be rewritten. This is important because programmers spend a lot of time rewriting code they had written before. Another benefit of software reusability is that the more code is reused the more reliable it will be since the code has been used in other situations and the likelihood of error decreases. [Bud91]

3.3.3 Benefits of Object-Oriented Languages

Objects are autonomous entities and interact with other objects via messages; therefore, objects lend themselves to greater modularity. Their behaviors are defined and implemented within each object and are independent of the implementation of other objects. This independence eases software development and maintenance. In addition, since most data and variables are encapsulated within the object, the parameter passing burden on the programmer is reduced since the appropriate methods (messages) are encapsulated within the object and the programmer need not be aware of what parameters need to be passed.[Pok89].

As classes inherit from superclasses, the code developed in the superclasses is reused. When inheriting from superclasses, the programmer need only add specialization suitable to the application.

3.3.4 Why use C^{++} instead of other Object-Oriented Languages

C^{++} is an enhanced version of the popular C programming language which was developed at Bell Laboratories. C^{++} evolved to meet the following goals:

- Retain the extremely high machine efficiency and portability that C has been famous for.
- Retain compatibility between C^{++} and C.
- Repair long-standing flaws, particularly C's lax treatment of types.
- Upgrade C in line with modern data-hiding principles.

C^{++} is a proper superset of the C language, with a few incompatibilities. Its enhancements include the ability to define classes, operations on objects of those classes, and a comprehensive set of ways to control operations on those objects, such as operator overloading, constructors and destructors. [Cox86]

C^{++} has become increasingly popular not only because of the popularity of C, but because of its own features, such as information hiding and other object-oriented features. Most C programs need not be ported to C^{++} since C^{++} is based on Standard C.

3.4 Concurrent Object-Oriented Programming

Concurrent Object-Oriented Programming implies that we have autonomous and concurrently executing objects which excute asynchronously.

Concurrent objects model the real world since objects in the real world are inherently concurrent. Concurrent objects interact with each other via message passing mechanisms, thus allowing objects to exploit parallelism and distributed processing. If object-oriented programming is natural, concurrent object-oriented programming is even more natural since it more closely resembles the 'real world'. In addition, concurrent objects allow for effective software development and execution in open distributed systems. [TB90]

Concurrent object-oriented programming languages may be used in many fields such as artificial intelligence, distributed databases, distributed operating systems, distributed simulations, and language parsing.

3.5 The Connection Machine

The Connection Machine is a data parallel computer system. Data parallel computing associates one processor with each data element. One of the major benefits of programming in a SIMD system such as the Connection Machine is that programming can be done in a serial fashion, with some operations executing in parallel the same instruction stream. The programmer need not be concerned with parallelism and some of the usual problems associated with parallelism, such as synchronization, race conditions, and deadlock.

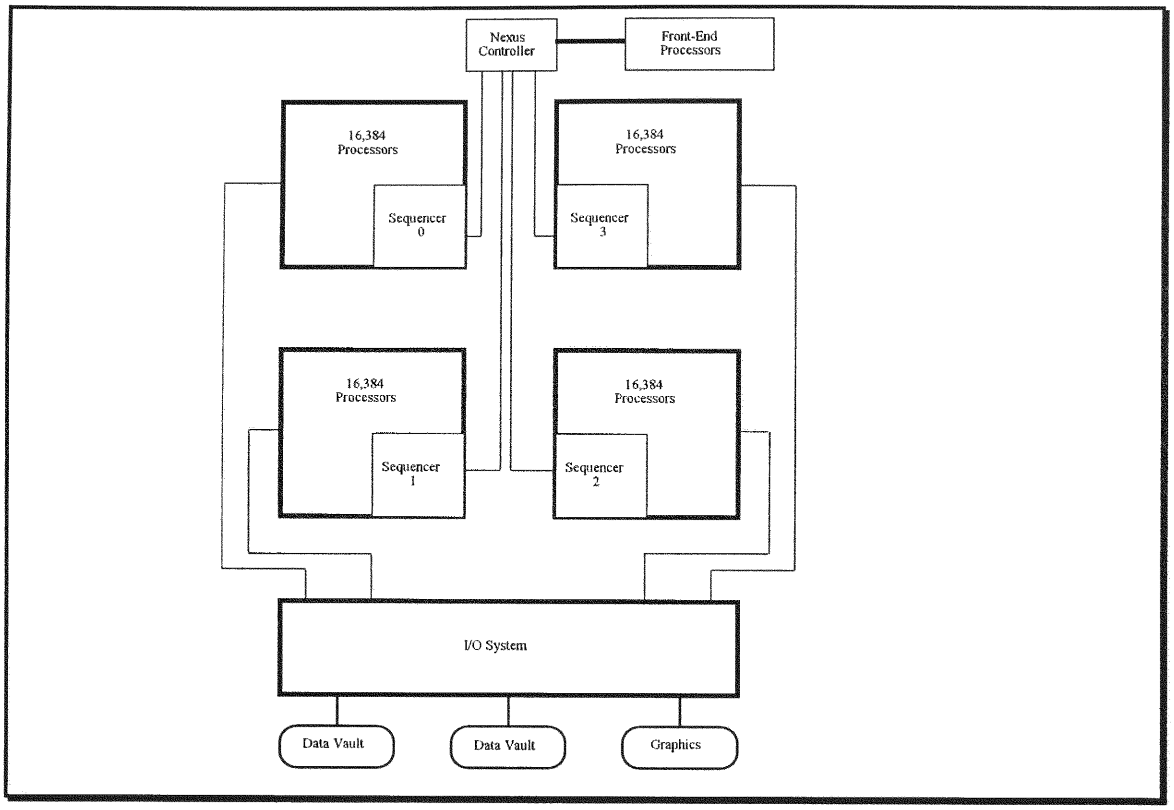

Figure 5: Connection Machine Configuration

C* 6.0 is available on the CM-2 and the CM-5. Both compilers parse the same language and code is for the most part portable. A few minor differences include:

- On the CM-2, a 'bool' data type is a bit; on the CM-5, it is a byte.
- On the CM-2, C* allows calls to PARIS (PArallel Instruction Set); PARIS is not supported on the CM-5.
- The CM-5 has runtime libraries for implementing MIMD.

3.5.1 What is the CM-2

The CM-2 is a data parallel system. Data parallel systems associate one processor with each data element. This computing style exploits the natural computational parallelism inherent in many data intensive problems. [CM2]

3.5.2 What is the CM-5

The CM-5 provides high performance plus ease of use for large, complex, data intensive applications. Its architecture is designed to scale to teraflops or teraops performance for terabyte-size problems. The CM-5 continues and extends support for the parallel programming model of the CM-2. The CM-5 takes advantage of the latest developments in high-speed VLSI, RISC microprocessors and networking. It includes fine and course-grained concurrency, MIMD and SIMD control, and fault tolerance. [CM5]

3.5.3 Differences between CM-2 and CM-5

The major difference between the CM-2 and the CM-5 is that the CM-5 can execute MIMD code since each node contains a SPARC processor rather than a simple bit-processor under strict front-end control. Furthermore, if the high-performance arithmetic hardware is included, then node memory is partitioned into four independent banks. The arithmetic hardware consists of four vector units. These vector units are memory controllers and a computational engine controlled by a memory-mapped control-register interface. Finally, the CM-5 can compute and move data faster than the CM-2.

3.5.3.1 Front End Processors on the CM-2

A front-end computer, usually a multi-user Sun workstation running SunOS, is the user's gateway to the Connection Machine system. Through the front-end, users develop, compile, debug, and execute their application programs. The front-end computer's file system holds all system software and user programs for the Connection Machine. A Connection Machine can have from one to four front-ends. See Figure 6.

3.5.3.2 Interprocessor Communications Network

The CM-2 supports various forms of communications within the parallel processing unit: routing, NEWS grid, spreads, and scanning.

- The router allows any processor to communicate with any other processor. Another way to view this communication process is that any processor may access any memory location within the parallel processing unit, with all processors making memory accesses at the same time.
- The NEWS (North East West South) grid is for communication operations between processors that are nearest neighbors within a Cartesian grid.

- Spreads allow a value from one processor to be sent to all other processors.
- Scanning combines communication and computation on NEWS grids. Simultaneously, in every row of a grid along a particular dimension, scanning computes all the partial sums of that row.

3.5.3.3 Sequencer for CM-2

The task of the sequencer is to decode commands from the front-end and broadcast the *nanoinstructions* to the data processors, which then execute the same instruction simultaneously.

3.5.3.4 Data Processors CM-2

Data processors execute arithmetic and logical instruction, calculate memory addresses, and perform interprocessor communication. In this respect, they are similar to a serial computer. The difference is that the processors do not fetch instructions from their respective memories; instead, the processors are collectively under the control of a single microcoded sequencer. The CM-2 can support from 4096 to 65,536 processors, each with either 64K or 256K of RAM.

3.5.3.5 CM Configuration at NRL

The CM-2 at NRL is a 16K processor system with two sequencers, two gigabytes of RAM, and a 50MB/sec I/O bus. Each sequencer decodes instructions for 8K processors. The data vault is a Parallel Disk array with 10 gigabytes of mass storage. The front ends are a Sun 4/690, two Sparc2 and a Sun 4/480.

The CM-5 at NRL has 32 sparc nodes, each with 4 vector units and 128 MB of RAM for a total of 16384 MB (32 * 4 * 128) of vector RAM; 24 2 gigabyte drives, 4 partition managers, an I/O Control Processor, a Compile Server and backup partition manager. The Partition Managers, I/O Control Processor, the Compile Server and File Server are all Sun Sparcstation II.

3.5.4 What is C* (Data Parallel Language)

C^* implements ANSI C; therefore, programs written in Standard C will run under C^*. C^* provides features and constructs which allow for data parallel computing. The compiler will translate C^* code into serial C code with calls to PARIS (**PAR**allel **I**nstruction **S**et) on the CM-2.

C^* is well suited for applications which require dynamic behavior, since it allows the size and shape of parallel data to be determined at run time. For example, the ***sizeof*** operator may be applied to a shape (a parallel data structure) or shape type in order to return the number of bytes in a shape object. This is needed so that the programmer can use a storage allocation system call to allocate storage for shapes. For the example,

```
    Sptr = (shape *) malloc(sizeof(Sc));  /* shape named Sc */
```

and

```
    Sptr = (shape *) malloc(sizeof(shape)); /* shape type */
```

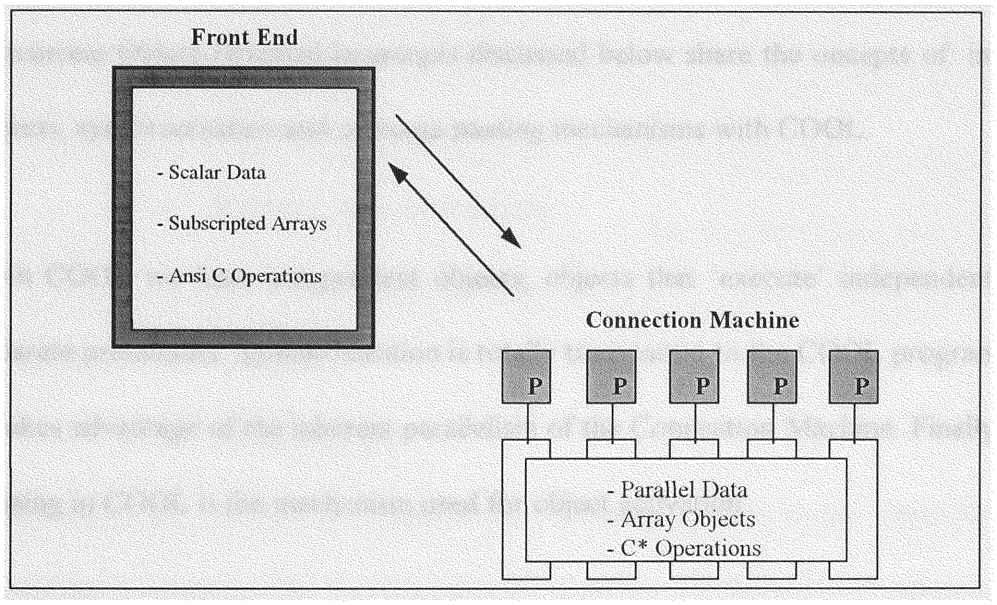

Figure 6: Computing on the Connection Machine

each allocate a new shape object which can be referenced by indirecting Sptr. [Fra91]

In addition, C* 6.0 provides programmers with the standard benefits of C, such as block structure, access to low level-facilities, string manipulation, and recursion. Furthermore, C* provides parallel processing capabilities without requiring the programmer to indicate synchronization explicitly in programs. [CM2] Unfortunately, C* 6.0 lacks object-oriented language constructs.

4. RELATED WORK

Advances in hardware and Software Engineering have lead to the development of several concurrent Object-Oriented languages, each with a different philosophy. The concurrent Object-Oriented languages discussed below share the oncepts of: independent objects, synchronization and message passing mechanisms with COOL.

With COOL we have independent objects; objects that 'execute' independently and on separate processors. Synchronization is totally transparent to the COOL programmer since it takes advantage of the inherent parallelism of the Connection Machine. Finally, message passing in COOL is the mechanism used for object activation.

4.1 Actor

The basic philosophy of the Actor model is that it organizes programs as collections of active objects which communicate in parallel via message passing. [YT87]

The Actor theory specifies that everything is an actor: functions, coroutines, data, processes, numbers, lists, databases and devices should be considered a actors and be capable of receiving messages. [Lieb87]

Actor objects communicate via message passing. Each actor has his own behavior when he receives a message. The script of an actor is a program which determines what the actor will do when it receives a message.

The Actor model of computation implies [TB90]:

- Maximal concurrency
- Bounded non-determinism
- Asynchronous processing
- Locality
- No assignment

Actor uses the concept of 'replacement behavior'. Replacement behavior is when computing a message, the current behavior specifies it's replacement behavior, i.e., the behavior which will compute the next message and how it will compute the next message.

Actors have concurrent activities; actors are not passive objects. Computing messages may be pipelined. Actions of a behavior are concurrent and message passing is asynchronous which implies that the sender and receiver are computing concurrently. Actor does not provide synchronization constructs. There are no message priority or message order preservation.

4.2 ABCL/1

ABCL/1 has an object-oriented computation model designed for modeling and describing a wide variety of concurrent systems [YT87]. The primary scheme of information computation is a set of abstract entities called objects and concurrent message passing. Objects are autonomous information processing agents which become active upon receiving a message. An object can send a message to any object as long as it knows the name (or object id) of the target object. This implies that message passing takes place in a point-to-point (object-to-object) fashion. Broadcasting of messages is not allowed [YST+87].

ABCL/1 supports asynchronous and synchronous message passing. ABLC/1 provides synchronization constructs:

- Serialized Object: the activation of a serialized object takes place one at a time and on a single first-come-first-served message queue for ordinary messages is associated with each object.

- Now Type Message Passing: a message passing of the now type does not end until the result is returned.
- Select Construct: when an object executes a select construct, it changes into the waiting mode.
- Parallel Construct: requires that its execution completes only when the execution of all the components complete. [YST+87].

Messages are prioritized, and message order preservation is preserved. ABCL/1 also allows for passive objects.

4.3 ConcurrentSmalltalk

ConcurrentSmalltalk incorporates concurrent constructs, a synchronization mechanism and atomic objects to Smalltalk-80 [YT87]. Syntax and semantics of ConcurrentSmalltalk are based on and upward compatible with Smalltalk-80.

In ConcurrentSmalltalk an object has internal states, which can only be modified by the object itself. In order to modify the state of another object, an object sends a message to it. The receiving object determines whether to accept the arriving message or not. When the receiving object receives the message, it executes the message corresponding to the request specified in the message.

4.4 C^{*++}

C^{*++} implemented multiple inheritance using C^{*} 5.0 on the Connection Machine. C^{*} 5.0 had *domains*, a construct very similar to classes on which multiple inheritance was built upon, yet maintianing the data parallel model of computation. [Gij90]

4.5 Ellie

Ellie is semantically and syntactically a rather simple language but it relies on some sophisticated ideas that all together constitute a very general and powerful language. All items in Ellie are objects. Ellie objects are similar in nature to Smalltalk-80 where all data items are objects on which methods may be applied in a uniform way. Block structured programming is supported in Ellie, but in Ellie blocks are object abstractions that locally may define attributes and objects. Delegation and multiple inheritance is supported by an Ellie mechanism called interface inclusion. Fine-grained parallelism and synchronization is expressed explicitly by means of bounded and unbounded (remote) procedure/method calls (RPC and URPC).

Ellie compilers have been implemented and are running on MS-DOS based personal computers, Unix based VAX systems, Sun 3 and Sun 4, and HP 9000 workstations [And91].

4.6 POOL-T

Is a parallel object-oriented language which is used to describe large systems on parallel architectures. POOL-T is object based. The data stored is stored in so-called instance variables. [Ame87]

POOL-T does not provide passive objects, yet primitive objects have no activity. Message passing, which is the synchronization construct in POOL-T, is synchronous. There are no message priorities, yet message ordering is preserved.

5. COOL

Providing object-oriented constructs to C* will greatly enhance the language since software developed will be easier to update and maintain, thus improving software quality. Code (tools) developed with object-oriented constructs will be easier to reuse for new program development, thus increasing productivity. Programming will become more intuitive since the programmer will need to think only about a collection of parallel objects instead of a collection of parallel data.

In sequential object-oriented languages objects interact with each other sequentially. By providing object-oriented constructs to C^*, objects will be able to communicate with each other in parallel. For example, we may have ten instances of object employee which when sent a message from single object payroll, would update ten different employee payroll records simultaneously, with one instruction.

COOL provides synchronization constructs (wait statement, explained later). Objects may be either synchronous or asynchronous, yet in the background all processes are synchronous due to the data parallel nature of the Connection Machine.

5.1 Based on C^*

COOL is based on C^* 6.0. One of the reasons for this decision is that C^* is well suited for applications which require dynamic behavior, since it allows the size and shape of parallel data to be determined at run time. COOL would allow existing C^* programmers to easily port their code to COOL in order to take advantage of COOL's concurrency and Object-Oriented constructs.

5.2 COOL Constructs

COOL maintains the inherent constructs found in C and C^{++}. Furthermore, COOL implements a subset of C^{++} constructs (classes, inheritance, ...) as well as a series of new concurrent constructs which will be described in detail later on.

5.3 COOL Objects

A COOL object provides abstraction and encapsulation to C^{*}. COOL objects may be in any of the following states:

- Unitialized, which serves as a prototype for new objects
- Initializing makes an object active; it may also serve as prototype for other objects
- Initializing an object is possible until the object receives a message from another object
- Active implies that the object is currently executing
- Passive implies that the object is waiting for activation

Object activation is the method of communication of COOL objects. A COOL object becomes active when one of its methods is invoked. COOL classes may inherit from other predefined classes or superclasses.

5.4 Concurrent Constructs Implemented in COOL

Six Concurrent Constructs have been implemented in COOL.

5.4.1 Send and Wait

This implies an synchronous object call where an object requests a result and waits for it. For example,

```
shape [1024]shape_name;
      /* Defines shape */
class Employee:shape_name employee;
      /* Declares class object*/
[22]employee..raise(10);
```

where Employee is the class of all employee objects, and it defines an ***int raise(int)*** method or member function. ***[22]employee*** is a single employee object which is mapped to processor 22.

5.4.2 Send, no Wait

This implies an asynchronous object call without synchronization. For example,

```
[22]employee..raise(10)**;
```

where one employee object receives the message to raise it's salary by 10 percent. It responds by acknowledging the receipt of the message, then it executes an appropriate method concurrently.

5.4.3 Send to more than one object and wait

This implies that the receivers are all instances of the same class. For example,

```
shape [1024]shape_name;
      /* Defines shape */
class Employee:shape_name some[50];
      /* Declares class object*/
some..raise(5);
```

where what is specified within square brackets [right indexing] indicates an array. The employees that are in the ***some*** array all get the same message. They respond collectively by executing an appropriate method. All employee objects return an integer, which is deposited into the array ***result***, which needs to have the same dimension. This case is appropriate for a data parallel model of execution.

5.4.4 **Send to more than one object, no wait**

This case implies several asynchronous object calls without synchronization. For example,

```
shape [1024]shape_name;
      /* Defines shape */
class Employee:shape_name some[50];
      /* Declares class object*/
some..raise(5)**;
```

where several employee objects receive the message to raise their salary by five percent. They respond by acknowledging the receipt of the message, then they execute an appropriate method concurrently.

5.4.5 Singular rendezvous

This implies the ability of sending a message to an object and not waiting for its answer (return value) immediately. Instead, we issue an asynchronous ***send***, perform other operations, and then wait for the return value. For example,

update code examples to reflect COOL current syntax

```
while_done_concurrently(a = employee[23].raise(10))
      {
      printf("hello world");
      ...other statements, without involving variable a
      printf("done with intermediate concurrent work");
      }
```

or, the above code could be written as

```
a = employee[23].raise(10)&
printf("hello world");
...other statements, without involving variable a
printf("done with intermediate concurrent work");
wait(a);
```

where a specific employee gets the message ***raise***. The employee acknowledges receipt and then concurrently executes its ***raise*** method. The sending program (the object where the above code is located) continues to execute as if it was a send with no wait. Then, it reaches the ***wait(a)*** statement: it waits until the employee object is done raising its salary and has produced a return value. Obviously, variable ***a*** should not be touched in the meantime.

5.4.6 Parallel rendezvous

This case is similar to cases four and five (multiple send, no wait). Messages are sent to a set of objects, all from the same class. There is a return value. The sender does not want to wait for the return value at the moment, but it will wait sometime in the future. For example,

```
while_done_concurrently(result = some.raise(5))
      {
      printf("hello world");
      ... other statements, without involving result
printf("done with intermediate concurrent work");
}
```

5.5 COOL Details

As in C^{++}, the '::' scope operator is used to indicate to COOL that the function being defined is a method of the specified class. For example,

```
void Screen::checkRange(int row, int col)
       {
       .... some code
       }
```

indicates that function checkRange is a method of class Screen.

In C^{++} inheritance is noted by the ':' operator. In COOL it was decided to use '<>' as the inheritance operator for readability purposes. The inheritance operator implies that the derived class inherits the members of the class it is derived from. For example,

```
class Employee <> Person
```

implies that Employee is inheriting the characteristics of superclass Person.

Classes are defined as in C^{++} by using the 'class' type. Data members and function members are defined as in C^{++} with the exception that data members should be declared first, followed by the '!!' separation operator, and then followed by function members. The reason for defining the '!!' operator was that while parsing there was no way of knowing when a data or function member was being defined; COOL takes the class definition and converts it to a struct for C^{*}. Within a struct we can only have data members, not function members.

The *overload* statement should be used within a class definition if function members are expected to be used with scalar and parallel data. For example,

```
class Employee
       {
       float salary;
       !!
       float raise(float salary, float rate);
       overload raise;
       float:current raise(float:current salary, float rate);
       }
```

The above example initially defines function *raise* as a scalar function; after the *overload* statement, function *raise* is defined as a parallel function. Therefore, based on the variables passed into the function, the compiler will determine if the function is scalar or parallel.

As in C^{*} 6.0, the *overload* statement should be at the beginning of the file that contains the declarations or function prototypes. Furthermore, the statement *must* appear before the declaration of the second version of the function, and it must appear in the same relative order with respect to the function declarations in a compilation unit [C*]

6. IMPLEMENTATION

6.1 Strategy Used to develop COOL

COOL has been implemented using traditional compiler construction tools: Lex and Yacc. C was used during the code generation stage.

Since C^* 6.0 and COOL are based on ANSI C, the first step was to locate versions of the ANSI C Yacc and Lex specifications; these specifications where found in the public domain over the Internet. With these specifications, we proceeded to generate the C^* 6.0 Yacc and Lex specifications. This took some time due to our unfamiliarity with C^*. Most if not all of the C^* syntax is recognized by COOL.

Once the C^* grammar was completed, the necessary rules for the constructs to be implemented in COOL were included. One of the goals for the COOL grammar was to have class representations be similar to C^{++}. This will be explained later on as the syntax of COOL is fully explained.

When testing of the grammar was completed, the necessary C^* code was generated to support the constructs being implemented.

6.2 Object-Oriented and C^* Constructs Implemented in COOL

COOL implements classes and inheritance in a similar manner as C^{++}.

6.2.1 Classes

COOL implements a simple model of classes as defined in C^{++}. All methods (functions) are ‘public’. At this time, COOL does not support ‘friend’ and ‘private’ functions nor does it support ‘constructor’ and ‘destructor’ functions. These enhancements are left for a future version of COOL

6.2.2 Inheritance

COOL implements a one level inheritance hierarchy. The reason was to prove that inheritance was possible; multiple inheritance can be easily implemented in future versions of COOL.

6.2.3 New Reserved Words and Operands

Table 1 briefly describes C^* reserved words and operands implemented in COOL:

C^* Reserved	Description
bool	unsigned single-bit integer type
current	current shape
dimof	returns # of positions along an axis
everywhere	work on active and inactive variables of the current shape. Active variables are those currently executing. The state of the variable depends on the nature of the instruction being executed: scalar or paralllel.
overload	provides more than one version of a function
pcoord	function which returns parallel variables whose elements are initialized to their coordinates along a specified axis
physical	a C^* predefined shape; it is a 1 dimensional shape with # of positions equal to number of physical processors allocated at run time
positionsof	returns total number of positions in shape
rankof	returns number of dimensions in shape
shape	shape declaration
shapeof	provides size of shape
where	sets context of parallel operations within its body
with	chooses current shape to use on parallel variables
<?	minimum of two variables

>?	maximum of two variables
%%	modulus of it's operands
-=	negative of the sum of values
+=	sum of values of parallel variable elements
&=	bitwise AND of values
^=	bitwise XOR of values
|=	bitwise OR of values

Table 1: C* Reserved Words and Operands Implemented in COOL

Table 2 briefly describes COOL reserved words and operands implemented in COOL:

COOL Reserved Words and Operands	Description
class	Define and declare a class object
wait	Wait for an event before proceeding
**	No wait; similar to '&' in Unix
<>	Inherit from another class
..	Inherit method
::	Declare method

Table 2: COOL Reserved Words and Operands

6.3 COOL Syntax

This section describes COOL's syntax as well as its limitations and restrictions.

6.3.1 COOL Limitations and Restrictions

Several limitations and restrictions were discovered while developing COOL:

- Definition of class function members should be defined outside of the class body.
- As in C^{++}, class definitions should be defined in a "class header" and a "class body". This "class header" should also contain the function member prototypes as well as the functions definitions themselves. The "class header" file must reside in the current directory. Syntax for the include file is:

  ```
  #include "def.h"
  ```
- COOL assumes that include statements of the form

  ```
              #include <stdio.h>
  ```

  are including system specific information and will not preprocess them. These include files will be pre-processed during final compilation on the Connection Machine at the Naval Research Lab in Washington D.C.

  Source files must be are invoked with the **-P** command line switch. This parameter is from the C preprocessor and the purpose is to use CPP's facilities for manipulating C preprocessor statements such as #define and

#include.

- When declaring a variable to represent a previously defined class, the word *class* must precede the declaration. For example

```
class Definition d;
```

  refers to a class which has been previously defined.

- Identifiers can have a maximum of 20 characters. Supported characters are the same as those for ANSI C.
- COOL provides inheritance one level deep at this time.
- Functions which will be used by both parallel and scalar (non-parallel) variables need to be declared using C*'s overloading features. Overloading means that a different version of the function is defined for each type of input needed. This needs to be done for class methods as well. For example,

```
overload raise;
float raise (float x);
      /* scalar version */
      {
      return(x+1);
      }

float:current raise(float:current x)
      /* parallel version*/
      {
      return(x+1);
}
```

- When defining a class, data members and function members need to be grouped together and separated as follows:

```
class Definition
        {
        int rate;
        float money;
        !!
        float raise(float);
                }
```

6.3.1.1 COOL Notation

6.3.1.1.1 Comments

Comments in COOL are the same as in C* and C. '/*' signifies the beginning of a comment and '*/' signifies the end of a comment

6.3.1.1.2 Legal Identifiers

Identifiers may use the following characters

```
A through Z
a through z
0 through 9
_ (underscore)
```

As in C* and C, the first character of an identifier must be a letter or an underscore.

6.3.1.1.3 Reserved Words and Operators

C* and C reserved words also are reserved words in COOL. For a complete list of C* and C reserved words and operators, please see appendices A and B.

COOL reserved words include:

```
class
wait
```

COOL operators include

```
**
!!
<>
..
::
```

6.3.1.1.4 Separators for Reserved words and Identifiers

```
nl/lf          newline linefeed
tab
space
/*             begin comment
*/             end comment
```

6.3.2 Scope Rules

COOL scope rules follow those of C* and therefore ANSI C; a name is known throughout the scope in which it is declared

6.3.3 COOL Grammar (YACC) Definition

The COOL grammar where the extensions to C* have been implemented are included below. The complete C, C* and COOL grammar may be found in appendix....

The following YACC rules define the WAIT statement used in the constructs defined in sections 5.5.1 and 5.5.3.

```
primary_expr
        : identifier
        | CONSTANT
        | string_literal
        | '(' expr ')'
        | WAIT '(' identifier ')'
        | PCOORD '(' constant_expr ')'
        | '.'
        ;
```

The following YACC rules define a CLASS.

```
class_specifier
        : class_def identifier '{' class_declaration_list '}'
        | class_def '{' class_declaration_list '}'
        | class_def identifier
        | class_def identifier INHERIT identifier '{'
          class_declaration_list '}'
        | class_def identifier ':' shape_expression identifier
        ;


class_def
        : CLASS
        ;


class_declaration_list
        : class_declaration
        | class_declaration_list class_declaration
        ;

class_declaration
        : specifier_list class_declarator_list ';'
        ;

class_declarator_list
        : class_declarator
        | class_declarator_list ',' class_declarator
        ;

class_declarator
        : declarator
        | ':' constant_expr
        ;
```

The following YACC rules define a method.

```
function_declarator
        : direct_declarator '(' parameter_type_list ')'
        | direct_declarator '(' ')'
        | direct_declarator '(' identifier_list ')'
        | direct_declarator COLON_COLON '(' ')'
        | direct_declarator COLON_COLON direct_declarator
          '('parameter_type_list ')'
        | direct_declarator COLON_COLON direct_declarator
          '('identifier_list ')'
        ;
```

7. SAMPLE PROGRAM

A simple payroll system will be used to demonstrate the effectiveness of COOL. The system has an Employee class from which 16 Faculty employee objects, 16 A&P employee objects and 16 USPS employee objects will be created. Note that A&P and USPS are employee classifcations used at Florida International University. The program calculates employee salaries and total salary expenditure by employee classification.

7.1 COOL Code

This is the sample COOL program used to demonstrate COOL. The code is preceded by a line number for clarification later on.

```
0   #include <stdio.h>
1   #include <stdlib.h>

2   class Employee        /* Class declaration */
3   {
4      int hours;         /* Class member */
5      float rate;        /* Class member */
6      float salary;      /* Class member */
7      !!
8      void doSalary();   /* Member function/method declaration */
9   }

10  void Employee::doSalary(int hours ) /* Member function definition */
11  {
12     Salary = hours * rate;
13  return;
14  }

15  main()
16  {
17     shape [16]shape_is;      /* Number of Processors to use */
18     class Employee:shape_is faculty;
19     class Employee:shape_is ap;
20     class Employee:shape_is usps;
21     int i;
22     loat faculty_total;
23     float ap_total;
24     float usps_total;


25     faculty_total=0;
26     ap_total=0;
27     usps_total=0;
28     with (shape_is)
29           {
30           faculty.rate=50.0;      /* Data replication */
31           ap.rate=40.0;           /* Data replication */
32           usps.rate=30.0;         /* Data replication */
33           faculty.hours=(pcoord(0)*5);
34           ap.hours=(pcoord(0)*8);
```

```
         usps.hours=(pcoord(0)*10);

         ap.doSalary()**;        /* Send to more than one No Wait */
         faculty.doSalary();     /* Send to more than one and wait */
         usps.doSalary();        /* Send to more than one and wait */
         wait(ap.salary);        /* Parallel Rendezvous */
         [10]ap.rate=45.0;       /* Change rate for employee # 10 */
         [10]ap.doSalary();      /* Send, no wait */


         printf("\nFaculty #     Rate  Hours Salary\n");
         printf("======================================\n");
         for (i=0; i<16; i++)
         printf("%d  %f    %d    %f
   \n",i,[i]faculty.rate,[i]faculty.hours,[i]faculty.salary);
         printf("\nA&P #         Rate  Hours Salary\n");
         printf("========================================\n");
         for (i=0;i<16;i++)
               printf("%d  %f    %d    %f
   \n",i,[i]ap.rate,[i]ap.hours,[i]ap.salary);
         printf("\nUSPS    #     Rate  Hours Salary\n");
         printf("=========================================\n");
         for (i=0;i<16;i++)
         printf("%d  %f    %d    %f
   \n",i,[i]usps.rate,[i]usps.hours,[i]usps.salary);

         faculty_total+=faculty.salary; /* Data Reduction */
         ap_total+=ap.salary;          /* Data Reduction */
         usps_total+=usps.salary;      /* Data Reduction */

         printf("\n\nTotal Salary Expenditures\n");
         printf("=============================\n");
         printf("Faculty Total Salary = $ %f\n",faculty_total);
         printf("A&P Total Salary = $ %f\n",ap_total);
         printf("USPS Total Salary = $ %f\n",usps_total);
         printf(" \n");
   }
}
```

7.2 Explanation

Lines two through eight are used for declaring class Employee with three members (hours, rate and salary) and one method (doSalary). Line ten illustrates a member function definition using the :: operator . Sixteen faculty, A&P and USPS objects are created on lines eighteen through twenty. Data replication (same value is assigned to the same variable on all active processors) is also illustrated .

The new constructs implemented in COOL are illustrated on lines thirty-six through forty-one. `ap.doSalary()**` invokes method `doSalary` for all A&P employees and does not wait for completion before proceeding to the next statement in the program. `faculty.doSalary` invokes method `doSalary` for faculty objects and waits until the operation is completed on all sixteen faculty objects before proceeding with the next statement. Before we go ahead and print, we need to be sure that A&P salaries have been computed; therefore, we issue a `wait(ap.Salary)`. Printing, however, may not be done concurrently, so we use a for loop to go through each object.

7.3 Sample Program output

```
Faculty # Rate       Hours      Salary
======================================
0         50.000000 0     0.000000
1         50.000000 5     250.000000
2         50.000000 10    500.000000
3         50.000000 15    750.000000
4         50.000000 20    1000.000000
```

```
5          50.000000 25   1250.000000
6          50.000000 30   1500.000000
7          50.000000 35   1750.000000
8          50.000000 40   2000.000000
9          50.000000 45   2250.000000
10         50.000000 50   2500.000000
11         50.000000 55   2750.000000
12         50.000000 60   3000.000000
13         50.000000 65   3250.000000
14         50.000000 70   3500.000000
15         50.000000 75   3750.000000

A&P #      Rate      Hours      Salary
========================================
0          40.000000 0    0.000000
1          40.000000 8    320.000000
2          40.000000 16   640.000000
3          40.000000 24   960.000000
4          40.000000 32   1280.000000
5          40.000000 40   1600.000000
6          40.000000 48   1920.000000
7          40.000000 56   2240.000000
8          40.000000 64   2560.000000
9          40.000000 72   2880.000000
10         40.000000 80   3200.000000
11         40.000000 88   3520.000000
12         40.000000 96   3840.000000
13         40.000000 104  4160.000000
14         40.000000 112  4480.000000
15         40.000000 120  4800.000000

USPS #     Rate      Hours      Salary
=========================================
0          30.000000 0    0.000000
1          30.000000 10   300.000000
2          30.000000 20   600.000000
3          30.000000 30   900.000000
4          30.000000 40   1200.000000
5          30.000000 50   1500.000000
6          30.000000 60   1800.000000
7          30.000000 70   2100.000000
8          30.000000 80   2400.000000
9          30.000000 90   2700.000000
10         30.000000 100  3000.000000
11         30.000000 110  3300.000000
```

```
12         30.000000 120   3600.000000
13         30.000000 130   3900.000000
14         30.000000 140   4200.000000
15         30.000000 150   4500.000000

Total Salary Expenditures
=======================================
Faculty Total Salary =    $ 30000.000000
A&P Total Salary =        $ 38400.000000
USPS Total Salary =       $ 36000.000000
```

7.4 Timing Details

Timing was not an issued during the development of COOL. The main consideration was the feasibility of implementing COOL. Future versions of COOL may either user C* optimization features or implement optimization directly into COOL.

8. CONCLUSIONS

There are many benefits to be achieved by using COOL on the Connection Machine, as will be explained shortly; there are also a variety of enhancements that can be added to COOL to make it a richer and more versatile programming language.

8.1 Benefits of COOL

COOL combines the object-oriented and parallel paradigms. COOL combines classes, inheritance and encapsulation with the parallel processing capabilities of the Connection Machine; therefore, the benefits of both 'worlds' will be available to the programmer. Application development for use on the Connection Machine will be easier to develop and maintain thanks to the object-oriented constructs added to C^*. Furthermore, the inherent parallelism of the Connection Machine provides for increased throughput.

8.2 Future Work and Enhancements to COOL

COOL could be further enhanced to become a concurrent version of C^{++} for the Connection Machine by providing more levels of inheritance as well as multiple inheritance. Multiple inheritance can be achieved by providing COOL with the necessary constructs. Private, protected and public access levels would also need to be implemented to become a concurrent version of C^{++} for the Connection Machine. Parallel rendezvous and other language constructs not yet defined may also be implemented for COOL in order to make it a more flexible language.

COOL currently provides very crude error checking (ie syntax error); better error messages and semantic error checking are features that may be included future versions of

COOL. Optimization has not been implemented in COOL; therefore, performance could be greatly improved be taking advantage of the optimization features of the target system (CM-2 or CM-5).

Finally, COOL may also be enhanced to take advantage of the MIMD features of the CM-5, not just SIMD which is what this implementation currently supports. Furthermore, performance and optimization enhancements may be added to a future version of COOL.

REFERENCES

[Ame87] P. America. *POOL-T: A Parallel Object-Oriented Language* in Object-Oriented Concurrent Programming, The MIT Press, 1987.

[And91] B. Anderson. *Fine-Grained Parallelism in Ellie.* Ph.D. Dissertation (partial), DIKE Report no. 91/4, Department of Computer Science, University of Copenhagen, Denmark, June 1991.

[Bail] J. Bailey. The Ghosts of Computers Past, Understanding Data Parallel Architecture in the Context of the Human Computing Era. In *Proceedings of the Conference on Scientific Applications of the Connection Machine*, pages 3-23, World Scientific, 1988.

[Bud91] T. Bud. *An Introduction to Object-Oriented Programming.* Addison-Wesley, 1991.

[C*] *C* Programming Guide Version 6.0 Beta*, Thinking Machines Corporation, 1990.

[CBF91] S. Chatterjee, G. Blelloch, and A. Fisher. Size and Access Inference for Data-Parallel Programs, *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, Toronto Ontario Canada June 26-28, 1991 pages 130 - 144.

[CM2] *Connection Machine CM-200 Series Technical Summary.* Thinking Machines Corporation, June 1991.

[CM5] *CM5 Technical Summary.* Thinking Machines Corporation, November 1992.

[Cox86] B. J. Cox. *Object-Oriented Programming, An Evolutionary Approach.* Addison Wesley, 1986.

[Dun90] Ralph Duncan. A Survey of Parallel Computer Architectures. In *Computer*, pp. 5-16, February 1990.

[Fra91] J. Frankel. *C* Language Reference Manual, A Draft Technical Report.* Thinking Machines, April 1991.

[Gir90] S. Girimaji. *Data-Parallel Programming with Multiple Inheritance on the Connection Machine*; Masters Thesis, Florida International University, 1990.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80, The Language and its Implementation*, Addison Wesley, 1983.

[HLJ+91] P. Hatcher, A. Lapadula, R. Jones, M. Quinn, and R. Anderson. A Production-Quality C* Compiler for Hypercube Multicomputers. In *Third ACM SIGPLAN Symposioum on Principles and Practice of Parallel Programming*, pages 73-82, ACM Press, 1991.

[KL89] Won Kim and Frederick H. Lochovsky. *Object-Oriented Concepts, Databases, and Applications.* Addison-Wesley, 1989.

[Lan87] T. Laning. Making Products Using Object-Oriented Programming. In *OOPSLA'87 Addendum to the Proceedings*, pages 105-111, ACM Press, 1987.

[Lie87] H. Lieberman. *Concurrent Object-Oriented Programming in Act 1* in Object-Oriented Concurrent Programming, The MIT Press, 1987.

[Lip89] S. Lippman. *C^{++} Primer*, Addison Wesley, 1989.

[LR92] T. G. Lewis and Hesham El-Rewini. *Introduction to Parallel Computing*, Prentice Hall, 1992.

[Pok89] B. Pokkunuri. Object-Oriented Programming. In *SIGPLAN Notices*. Vol 24, Number 11, pages 96 - 101, ACM Press, November 1989.

[TB90] M. Tokoro and J.P. Briot. Object-Oriented Concurrent Programming. *TOOLS Pacific, Technology of Object-Oriented Languages and Systems.* 1990.

[Weg90] P. Wegner. *Concepts and Paradigms of Object-Oriented Programming.* In OOPS Messenger, Vol 1 Number 1, ACM Press, August 1990.

[YST+87] A. Yonezawa, E. Shibayama, T. Takada and Y. Honda. *Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1* in Object-Oriented Concurrent Programming, The MIT Press, 1987.

[YT87] A. Yonezawa and M. Tokoro. *Object-Oriented Concurrent Programming: An Introduction* in Object-Oriented Concurrent Programming, The MIT Press, 1987.

Appendix A: C Reserved words used with COOL

auto	break	case
cdecl	char	const
continue	default	do
double	else	enum
extern	far	float
fortran	goto	huge
if	int	long
near	pascal	register
return	short	signed
sizeof	static	struct
switch	typedef	union
unsigned	void	volatile
while		

Appendix B: C* Reserved Words used with COOL

bool	current	dimof	everywhere
overload	pcoord	physical	positionsof
rankof	shape	shapeof	where
with			

Appendix C: COOL YACC Grammar Specification

The COOL Yacc Grammar specification is submitted seperately.

Appendix D: COOL LEX Specification

The COOL Lex specification is submitted seperately.

Appendix E: COOL Main program and Sample Program

The COOL Main program and Sample Program code are submitted seperately.

Appendix E: COOL Main program and Sample Program

The COOL Main program and Sample Program code are submitted seperately.