4-1990

Data-parallel programming with multiple inheritance on the connection machine

Sanjay Girimaji
Florida International University

ABSTRACT

DATA PARALLEL PROGRAMMING WITH MULTIPLE INHERITANCE ON THE CONNECTION MACHINE

by

Sanjay Girimaji

The demand for computers is oriented toward faster computers and newer computers are being built with more than one CPU. These computers require sophisticated software to program them. One such approach to program the multiple CPU machines is through the use of object-oriented programming techniques. An example of such an approach is the use of C* on the Connection Machine.

Though C* supports many of the object-oriented concepts, it does not support the concept of software reuse through inheritance. This thesis introduces a new language called C^{*++} , an extension of C* language to support inheritance. We also discuss the issues invloved in the implementation of multiple inheritance in programming languages.

This thesis describes the differences between C^{*++} and C* . It also discusses the various issues involved in the design and implementation of the translator from C^{*++} to C* . It also illustrates the advantages of programming in C^{*++} through an example. Since C^{*++} is designed to support software reuse which allows the users to create quality software in shorter time, it is anticipated that C^{*++} will have widespread use in programming the Connection Machine.

DATA-PARALLEL PROGRAMMING WITH MULTIPLE INHERITANCE ON THE CONNECTION MACHINE

by

Sanjay Girimaji

A thesis submitted in partial fulfillment of the

requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

at

FLORIDA INTERNATIONAL UNIVERSITY

1990

DATA-PARALLEL PROGRAMMING WITH MULTIPLE INHERITANCE ON THE CONNECTION MACHINE

by

Sanjay Girimaji

A thesis submitted in partial fulfillment of the

requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

at

FLORIDA INTERNATIONAL UNIVERSITY

Committee in charge:

Professor Raimund K. Ege — Chairperson

Professor Farahengiz Arefi

Professor Doron Tal

April 1990

To Professors Raimund K. Ege, Farahengiz Arefi and Doron Tal

This thesis, having been approved in respect to form and mechanical execution, is referred to you for judgment upon its substantial merit.

Arthur W. Herriott, Acting Dean
College of Arts and Sciences

The thesis of Sanjay Girimaji is approved.

Professor

Professor

Major Professor

Date of Examination:

ACKNOWLEDGEMENTS

I would like to express my appreciation and gratitude to all those who have been involved in the preparation of this thesis: to my professors who initiated, supervised, and contributed to the final product, to all the graduate students for their support and help.

Sanjay Girimaji

Table of Contents

List of Figures

1 INTRODUCTION

The rise in demand for computers has been phenomenal. In the recent past the demand has been oriented toward faster computers with larger memories. With the sequential computers reaching almost their physical limits, researchers have turned toward other approaches to achieve speed in computation.

This has lead to the development of multiple CPU machines. Instead of one processor solving the problem, multiple CPUs combine to solve the problem. Intuitively this leads to faster solutions to the problem being solved. Some examples of multiple CPU machines are ILLIAC 4, N Cube, IBM RP3 and the Connection Machine. To program such hardware, sophisticated software is needed.

Software evolution, on the other hand, has been evolving towards faster development and higher levels of abstraction and specification. These advances have lead to object-oriented methodology, the methodology in which an object models an instance of an abstraction of a real world concept. An object combines with other objects to solve a real world problem. Some examples of object-oriented languages are Smalltalk 80 [Gol84] and C++ [Lip89]. The terminologies for an object and related information may vary among these languages, but the underlying concepts remain the same.

The mapping between object-oriented language and multiple CPU machines seems to be natural. Each object can be mapped to a CPU in solving the real world problem. For the generation of multiple CPU machines, object-oriented languages seems to be one of the natural choices. An example of such a culmination of multiple CPU machine

with an object-oriented language is the usage of C* on the Connection Machine. There are other object-oriented languages which support parallel programming. Languages like concurrent C++ [YT87], concurrent Smalltalk [YT87], ACTOR [Agh86] and SINA [YT87] support object-oriented concepts for concurrent computation . Even though these languages can support SIMD computation, they were primarily designed for MIMD computations. On the other hand C* was primarily designed to support parallel computation on a specific parallel machine, the Connection Machine. C* makes efficient usage of the Connection Machine architecture and is more suitable than any other general purpose concurrent object-oriented programming language. This efficiency is at the cost of portability. Since speed is the major criteria in multiple computing, C* has a major advantage over other general purpose concurrent object-oriented language. *Lisp is another programming language which can be used on the Connection Machine [Gro89]. *Lisp which supports parallel processing and interprocess communication between processors, directly relates most of its parallel instructions to the underlying mechanism. Hence *Lisp programs are faster on the Connection Machine than C* programs. However higher levels of abstractions can be specified better by using C* . So to use object-oriented concepts on the Connection Machine, C* seems to be the most logical choice with its blend of object-oriented concepts and speed of execution of C* programs on the Connection Machine.

Though possessing many of the object-oriented principles, C* does not have the properties of information hiding and software reuse through inheritance. One of the

major means of rapid software development is lost in C* because of the lack of software reuse. This thesis is an attempt to address the problem of software reuse through inheritance. In this approach we introduce a new language called C^{*++} , which is an extension of C* . C^{*++} allows the specification of subclasses and by using subclasses we can reuse software. C^{*++} supports multiple inheritance of classes and the user has the flexibility of reusing many more classes. Because of multiple inheritance, conflicts regarding naming arise and such conflicts have to be resolved before execution. One method of resolving such conflicts is by allowing the user to resolve all conflicts before execution and C^{*++} follows such a methodology. The user has the ability to rename ancestral data and also to selectively inherit functions from the ancestral classes. (Ancestor is used synonymously with parent in this thesis). We also provide with a translator for converting C^{*++} programs to C* programs which can then be compiled and executed on the Connection Machine.

The thesis is organized as follows. In section 2 we discuss the approaches used to achieve multiple CPU computing while in section 3 we discuss the object-oriented concepts. With the necessary background information, we describe in section 4 the programming environment on the Connection Machine. Section 5 describes the C^{*++} language, its differences with C* and the translation of C^{*++} programs to equivalent C* programs. We also illustrate the differences between C^{*++} and C* through an example. In the final section we discuss our conclusion and suggest ideas for further research and development.

2 PARALLEL COMPUTATION

Flynn [Sto80] classified computers based on the instruction and the data streams. The single instruction and a single data stream computers were the conventional Von Neumann machines and were called the SISD machines. MISD machines are those in which there are multiple instruction streams and a single data stream. In both these architectures there is only one computational engine. However for faster computations multiple computational engines are needed.

In parallel computation, the basic concept is to have multiple computational engines operating on a problem. By using multiple engines, it is possible to solve a given problem faster than by using only one computational engine. Flynn proposed two more architectures which supported parallel computation in principle. They are are Single Instruction Multiple Data (SIMD) type and Multiple Instruction Multiple Data (MIMD) type of machines [Sto80]. We will study each of them briefly .

2.1 SIMD Machines

In SIMD machines a single instruction is executed by many different processors simultaneously. In this case there is only one instruction executed by all the processors on multiple data. Fig. 1 depicts the architecture of the machine which supports SIMD type of computation. The major components in SIMD machines are the control unit, instruction bus, the processors and the interconnection network.

In Fig. 1, the control unit is the front-end of the SIMD machine in which the

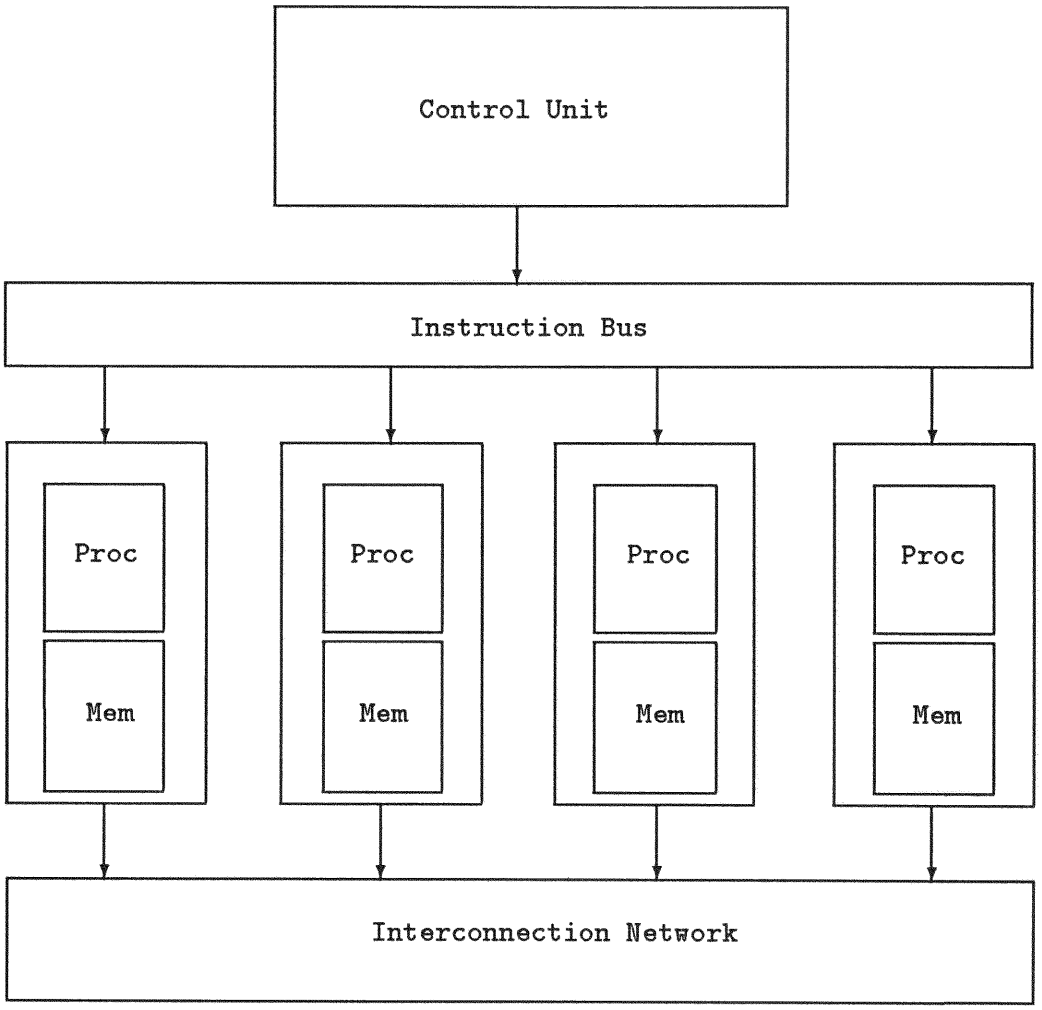

Figure 1: **Graphical Representation Of SIMD type Processing**

program resides. The control unit sends the instructions to the processors through the instruction bus. Each processor, which has its own memory, executes the instruction provided. It communicates with the other processors through the interconnection network. The interconnection network can be of any topology like the cube or mesh connection. At the beginning of each instruction, all the processors evaluate a boolean guard to check if they have to execute the instruction. If the guard is true, then the processor executes the instruction, else the processor waits until the beginning of the next instruction at which point all the processors are synchronized again. Hence in SIMD type of computing all the processors execute in a step-lock fashion.

SIMD type of computing is a simple model in which synchronization of processors is built-in. The program is stored in the central control unit and need not be copied into all the processors. The processor's memory can be used for storing the data associated with that processor. SIMD machines are fast and memory efficient for data-parallel programs, programs which are parallel in data and scalar in control. Programs which simulate propagation of heat through metals, sound through water and pressure through structures are examples of data-parallel programs.

The Connection Machine is an example of a SIMD machine. Let us analyze the solution to the problem of heat flowing through metal using the Connection Machine. In the case of heat flowing through metal, the metal sheet can be represented as a grid. Each element of the grid can be associated with a processor which contains information about temperature at that point (each object is mapped to a processor).

The temperature in each of the processors is proportional to the temperatures in its neighboring processors (namely to the north, east, west and south of the processor). So whenever there is a change in temperature in an object, the temperature in the neighboring objects is also affected. The heat is transmitted to all other parts till equilibrium is reached. During the time taken to attain the equilibrium, the temperature at a given instant in all the processors can be evaluated simultaneously on a SIMD machine like the Connection Machine. To evaluate the temperatures, thousands of processors could be iterating for hundreds of times till equilibrium of temperature is reached in all of them. The computational complexity of this algorithm would be O(Man) (M is the number of iterations while N is the number of processors used in SIMD approach) on a sequential machine while it is O(M) on a SIMD machine.

Based on the temperature changes in the processors to reach equilibrium and the time taken to reach it, it is possible to evaluate the intensity of the temperature and its origin. In this approach the control mechanism for all the processors is the same while the data can be different (data parallel) and such programs are best suited to SIMD type of computation. SIMD machines are not efficient if the programs are not data-parallel in nature.

2.2 MIMD Machines

In MIMD machines, multiple processors execute multiple control program on multiple data. The control program and the data can be different for each of the processor

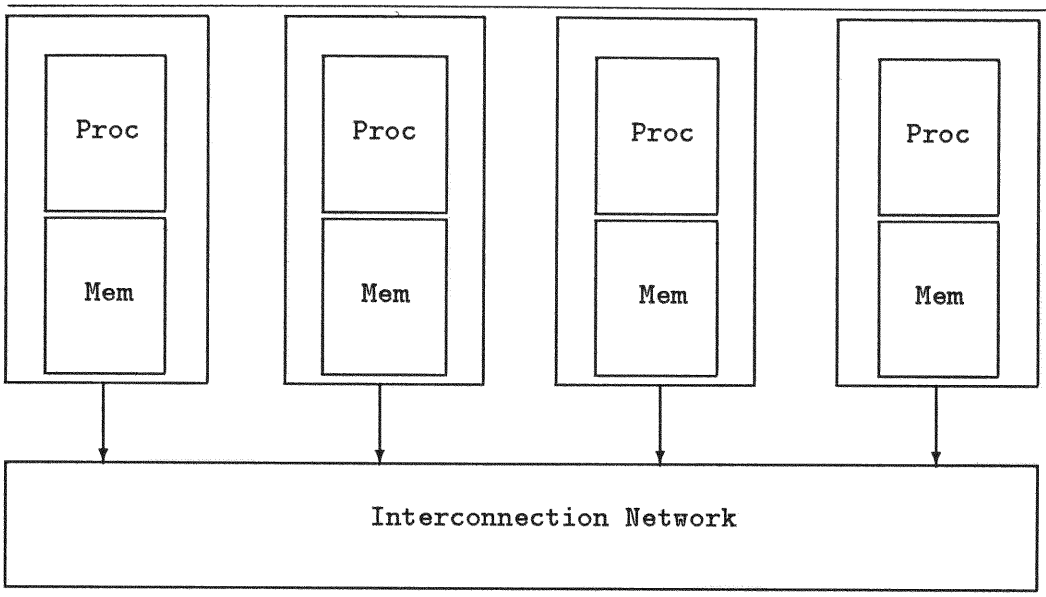

Figure 2: **Graphical Representation Of MIMD type Processing**

in this mechanism. Let us study the architecture of a MIMD machine to see how such computations can be achieved.In MIMD type machines, the major components are the processing element and the interconnection network. The processing element consists of the processor and the memory. Unlike in SIMD machines, each processor stores the program in its own memory.

In the interconnection network any processor can be connected to every other processor or they can be connected in a mesh or a hypercube like connection. Each processor executes asynchronously (with other processors) and communicates with other processors in solving the problem. If the communication is through shared memory, then different machines could try to access the same data and data access conflicts could arise. Such conflicts have to be resolved. However if the memory is

local to each processor and the communication is done through the inter-connection network, then there are no memory access conflicts.

The main advantage of an MIMD machine is its flexibility. The flexibility offered by MIMD machines is at the cost of overheads involved in the communication between processors. Because of its flexibility, MIMD computers can solve a wide range of problems. MIMD type computers are not memory efficient since each processor has its own copy of the program. MIMD machines are suitable for control parallel programs, programs in which the control is different for each processor. An example of such a situation is a rule based expert system, in which actions are triggered by certain conditions being met. For a given condition many different actions can be triggered. Each of these different actions can be executed by a different processor. Expert system and Prolog programs are some examples of control parallel programs in which a goal can be reached by using different premises. By following different paths of execution, the same goal can be reached at, and all of them can be executed concurrently. Each such path of execution could be executed on a different processor and the solution could be reached faster.

A simple example of a control parallel program is a Proplog program [MW88] that is used to check if a student can register for advanced study in computer science. A student satisfies the criteria if he/she has completed three basic courses and one course in a special group or two basic courses and two special courses. The rules for the program are given below.

```
advancedCS  :- group1A11, group2One.
```

```
advancedCS  :- group1Two, group2Two.

group1All   :- compilers, opSys, arch.

group2One   :- fileStruct.     group2One  :- dbms.
group2One   :- progLang.       group2One  :- compComm.
group2One   :- ai.             group2One  :- graphics.
group2One   :- research.       group2One  :- micros.

group1Two   :- compilers, opSys.
group1Two   :- compilers, arch.
group1Two   :- arch, opSys.

group2Two   :- fileStruct, progLang.
group2Two   :- fileStruct, dbms.
.
.
.
group2Two   :- fileStruct, micros.

.
.
.
/* All combinations of two group2Two courses */
/* are declared before.                      */
```

The goal is to check if the student has an advanced degree in Computer Science. The computer checks if the user satisfies group1All criteria or the group2One criteria. These checks can be done in parallel. The algorithm to check these conditions has parallel branches and parallel branches can be executed concurrently. When a problem involves multiple control programs (and multiple data too), then MIMD machines are ideally suited in solving such problems.

2.3 Comparison

MIMD machines are more flexible than SIMD machines because they do not impose any restriction on the order of execution. Also the interconnection network in MIMD

machines are more complex than those of SIMD machines. This flexibility of the interconnection is at the cost of speed of execution. However the selection of one of these architectures is based on the problem to be solved. If the problem is data-parallel in nature then SIMD machines are better suited. If the problem is control-parallel in nature then MIMD machines are better suited.

In the presence of "if-then else" instruction, MIMD machines are faster than SIMD machines. We will analyze the reason below.

2.3.1 If-then-else construct

Let us consider the execution of a program segment with an "if-then-else" statement by both SIMD and MIMD machines, as shown below

```
.
.
.
if (score > 50)
   update_gpa();
else
   print_score();
next_instruction
.
.
.
```

Consider Fig.3 in which each circle represents a processor and to the left of the "steps" we have the SIMD machines and to the right of it we have the "MIMD" machines. In step 1, all the processors test the boolean condition. In step 2, only the "true" processors execute the function "update_gpa" in SIMD machines while the "false" processors wait. Since MIMD machines do not impose any restriction on the synchronicity, some processors execute the function "update_gpa" while others execute the function "print_score". In step 3, in SIMD machines, the "false" computers

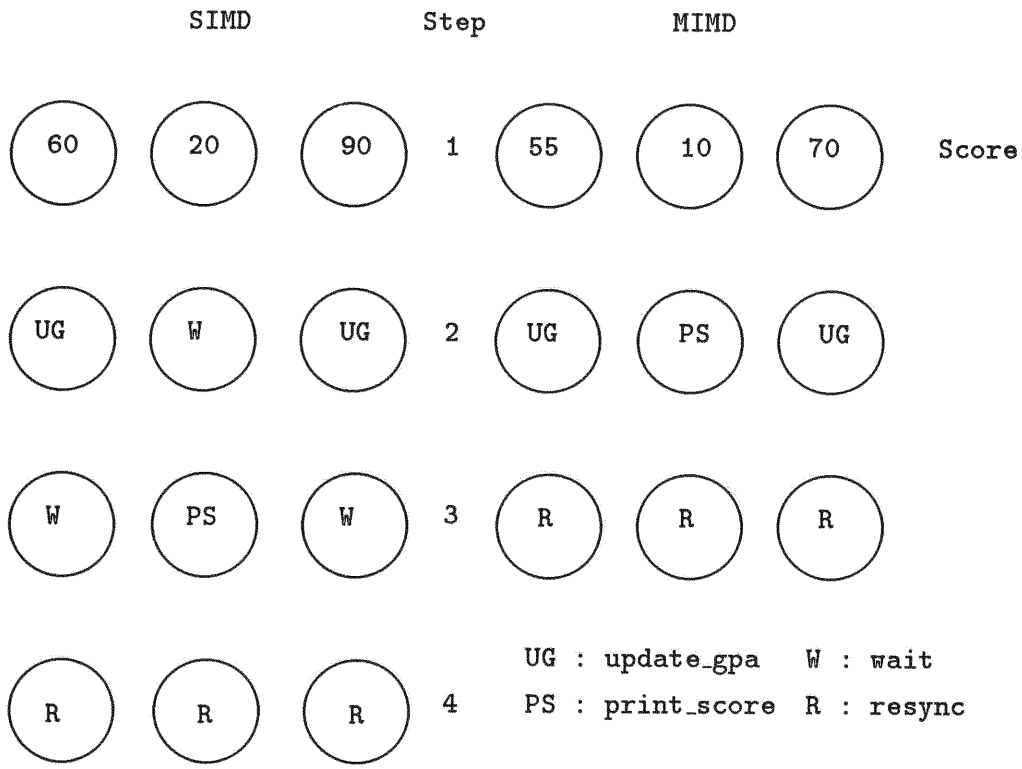

Figure 3: **If-Then-Else construct**

execute the function "print_score", while the "true" computers wait until the beginning of the next instruction. In MIMD machines all the processors execute the next instruction in step 3. In step 4, all the SIMD processors are resynchronized and they execute the next instruction. As can be seen, all the processors in a SIMD machine have to be synchronized at the end of each instruction, which slows down the overall execution time of the program. The same analysis holds true for the case statement too, and in this case SIMD machines are even slower.

MIMD machines, however use more memory than SIMD machines. This is because of multiple copies of the program, one in each of the processors. Also the interconnection network is more complex in the case of MIMD machines and hence the communication overhead is greater in MIMD machines. On the other hand debugging programs is easier on SIMD machines than on MIMD machines because execution can be reproduced in SIMD machines (because of their synchronicity). Execution is asynchronous in MIMD machines and hence they are difficult to reproduce which makes debugging harder.

MIMD machines seem to be more attractive than SIMD machines for multiple CPU computing. However the complications of reliable and fast interconnection network and the problem of synchronization have not been solved efficiently to make commercially viable MIMD machines. On the contrary there are SIMD machines with thousands of processors commercially available now and the Connection Machine is one such example. So SIMD machines seem to be an attractive solution for multiple CPU computing now. Let us study an application of multiple CPU machines used to solve real world problems.

Consider the following algorithm which is used to analyze the propagation of sound through water. The sound intensity at different points in the ocean are studied for

known sound intensities and range. These measurements are used to evaluate the intensity or range or the type of sound.

The methodology followed is as shown: The ocean is divided into grids and each element of the grid is assigned to a processor (The number of rows in this grid is assumed to be N). An initialization matrix (size NxN), is calculated based on empirical studies. A psi vector (size Nx1) is calculated based on the intensity of sound and its origin in the ocean. This psi vector is mapped to the leftmost column in the grid. The sound is then propagated to all the processor in the grid through a function called "Marching function". The value of sound intensity in each processor is modified based on the marching of sound which can be represented by the following function:

$$X_{i+1} = AX_i$$

where

X_i : psi vector in column i

A : the initialization matrix

X_{i+1} : psivector in column i+1

These calculations are repeated over a period of time on all the elements of the grid. So the number of matrix multiplication are thousands in number. Such multiplications which take O(NxN) time on sequential machines, can be executed in O(N) time on the Connection Machine. If there are M columns in the grid, then all the columns can execute their multiplication in parallel. So each iteration of matrix multiplication in a SIMD takes O(N) time. In a sequential machine, the same problem would have the computational complexity of the order O(NxNxM). Since the number of iterations is

high, it can be seen that SIMD machines are faster in such computations.

Based on the value of the sound in each processor in each iteration, graphs can be drawn to represent the propagation of sound through water. This process can be repeated for different initial psi vectors .

When the intensity and source of sound has to be computed, measurements are taken in certain points of the grid. By using the existing data and the measurements taken, it is possible to evaluate the source and the intensity of sound. [Lis90].

So SIMD machines are more suitable for data-parallel programs than control-parallel programs. So in general the nature of the problem dictates the selection of the architecture (SISD, SIMD or MIMD) used in its solution.

3 OBJECT-ORIENTED CONCEPTS

Increase in software size, difficulties in maintaining large software and an attempt to decrease the increasing software costs are among the various reasons that have led to advances in software technology. The evolution of software through the processes of design, development and implementation became more structured and the transition between stages smoother. One of the results of the evolution has been the development of software based on objects. The notion of what an object is, varies. Smalltalk [Gol84], C^{++} [Lip89], Actors [Agh86], Ada [Boo86], etc have their own concept of an object, but in general an object is an instantiation of an abstraction of a concept. To solve a problem, these objects perform computations, and may also interact with each other through message passing. An object can be assigned to a processor and each processor can combine with other processor in solving the problem. An object is self contained and has the properties of abstraction, modularity, encapsulation, information hiding, software reusability and inheritance and these can be implemented in each of the processors with hardware and software support. So multiple CPU computing seems to match the requirements of object-oriented programming. In the following section we present the object-oriented terms used in this thesis and the semantics associated with them.

3.1 Terminology

Since different languages have different interpretations of object-oriented terms, we now define our understanding of some object-oriented terms used in this thesis.

Object

An object is a combination of data and functions (procedures) that represents an instance of an abstraction. In conventional programming, an object can be viewed as a variable with associated functions.

Class

A class is a description of a collection of objects with the same data and functionality. A class can specify some data and functionality as private and others as public. By private we mean that the data or function can be accessed only by the object and not other objects. If data or function are public, then other objects can also access it. In this thesis, class, domain and module represent the same concept and hence can be used interchangeably unless specified otherwise. The data associated with an object is (are) also called as the instance variable(s) of a class.

Function and Message

A functionality can be defined for an object, called "function". This is the same as method or procedure in some object-oriented languages. A function is invoked when an object receives a message sent by another object. The sender object transmits the function name and if needed, arguments, as a message. The receiver object executes the function called and returns the result to the sender.

3.2 Software Engineering and Object-Oriented Programming

In this section we discuss how object-oriented programming support the basic principles of software engineering. The concepts discussed are abstraction, modularity, information hiding and software reuse through inheritance.

3.2.1 Abstraction

Abstraction is the technique by which a user can specify how to solve a problem at a certain high level without worrying about the lower level details. In object-oriented programming abstractions can be specified as classes and in C* they can be specified as domains.

3.2.2 Modularity

By using structured design, software can be divided into different addressable elements called the modules. These modules can be coupled to solve a problem. Good software should be modular, where different modules co-operate with each other to solve a problem. Modules can be mapped to classes in data parallel programming language .

3.2.3 Information hiding

By following modular design and development, one can obtain independent modules which cooperate with each other. When solving a problem, different modules may have to interact. In such cases the modules should depend only on the functionality of other modules and not on their internal details. This is supported by the concept of object communicating with other objects only through messages sent to its

(object's) methods. The internal details in a module should be local to that module and should be hidden from other modules. This property reinforces the ideas of modularity and abstraction and helps in maintaining and modifying modules with little or no changes to other modules. In a parallel machine this can be supported by the hardware which allows interprocess communication through message passing only and not through memory accesses. It can also be supported by the software if the data parallel programming language supports interprocess communication only through message passing.

3.2.4 Software reusability

Using existing modules to create new modules is the basic idea of this concept. Instead of "reinventing the wheel", by using software already developed, used and tested, we can create new software. By reusing software, not only can a user concentrate more on design of the software than its implementation, but also produce software with minimum effort. This can be supported by the concepts of classes and subclasses and the inheritance of data and methods from the ancestral classes in object-oriented programming . If the data parallel programming language supports the specification of classes and subclasses, then software can be reused in that language.

3.3 Inheritance

A technique to achieve reusability is called inheritance. Reusability is achieved by creating new modules based on the existing modules with few additions. The inheritance relationship is hierarchical in structure. All the functions offered by the existing modules do not have to be repeated in the new modules. Instead, the user

can concentrate on the newer concepts and functions only.

Inheritance is essential for software reusability in any object-oriented programming language or environment. A module can inherit from one or more modules. All the public variables (variables that can be seen by the descendant classes) and functions offered by the ancestral modules are available to the current module. The user can inherit from these modules and modify them to his/her needs. While the consistency of the ancestral modules is maintained (because the user cannot change them), extension to those modules is supported by the language. Consistency of a module is very essential because other modules could depend on the definition of the functions offered by the module and if they (functions) are not consistent, the whole system may not be consistent. Inheritance can not only be viewed as extension to a module, but also as a specialization of a module. The parent module can be a generalization while the child could be a special case of the parent. As an example, consider a module "integer" which offers the functions of addition, subtraction, division and multiplication. Another module called "small integer" can be defined as the child of the module "integer". "small integer" is obviously a special case of "integer" which inherits all the functions offered by "integer".

Graphical representation The inheritance relationship between modules can be represented as a directed acyclic graph with the nodes representing the different modules and the arcs representing the relationships between them. It should be noted that the graph has to be acyclic since the inheritance relationship is hierarchical in structure.

In Fig. 4, Person is a module which has some data and functionality associated with it. Among its submodules is the module Student. Student can be viewed as a

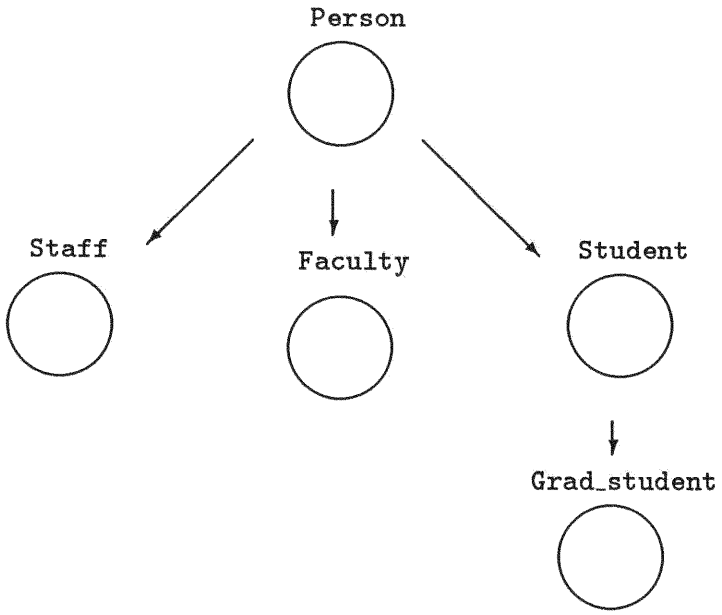

Figure 4: **Graphical Representation Of Inheritance**

special case of a Person. Student can add its own data and functions in addition to those inherited. Similarly Grad_student is a submodule of Student and hence, inherits all the data and functions of Student. Grad_student can be viewed as a Student with information about the thesis associated with it and also as an extended Student.

3.3.1 Multiple Inheritance

In this section we will study the concept of multiple inheritance and its advantages to understand the implications of introducing multiple inheritance in C* . Inheritance is the technique by which a class can inherit all the functionalities and data offered by the ancestral class and it is more so in the case of multiple inheritance. Multiple inheritance can be depicted as directed acyclic graph as shown in Fig. 5.

The Fig. 5 is similar to Fig. 4 with the addition of the class Faculty as a superclass of the class Grad_student. Grad_student is viewed as a Student who teaches. It inherits from the class Student and also from the class Faculty. If Grad_student is viewed based on the functionality it offers, then inheritance can be viewed as a specialization of a class. If Grad_student is viewed as a data type, then inheritance can be viewed as an extension of a class.

Multiple inheritance, while possessing many useful properties, has some inherent problems associated with it. We will now consider the advantages and the problems associated with multiple inheritance.

Advantages

Reuse of software Inheritance, multiple inheritance in particular, allows for extensive software reuse. The user has to define only the extensions to and specializa-

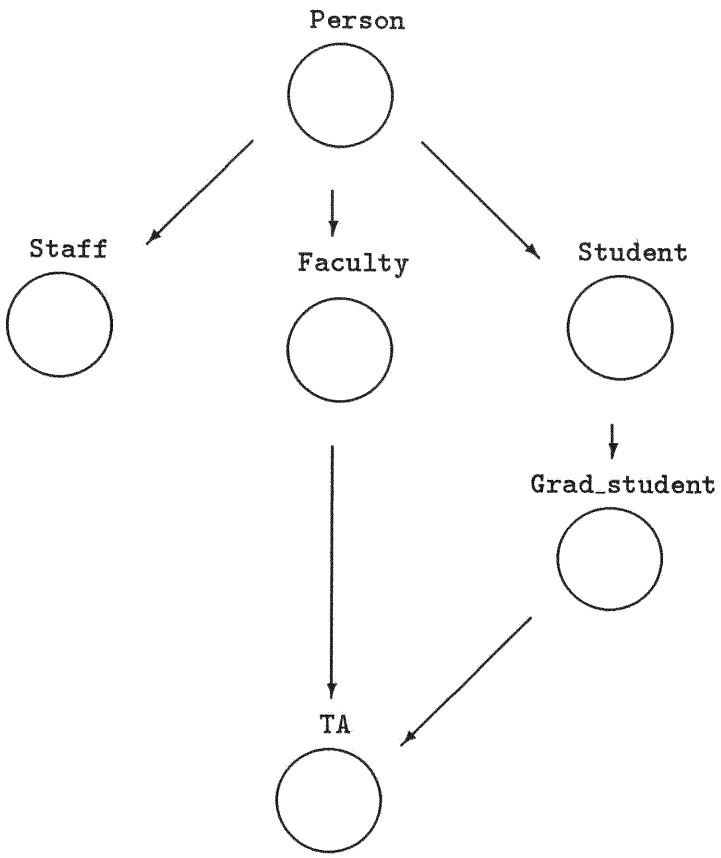

Figure 5: **Graphical Representation Of Multiple Inheritance**

tions of existing classes. This not only increases productivity, but also allows the user to concentrate only on the design and development of the changes. If the existing classes have been used and tested, inheritance helps in debugging software because bugs are isolated to a smaller part of the code.

Code size Multiple inheritance allows for greater amount of code to be reused. This decreases the size of the source code. The object code size could also decrease based on the linking mechanism. Linking of inherited classes can be either static or

dynamic. In the case of static linking, all the methods for certain specified class(es) are linked to form the executable file. Using dynamic linking, only those methods used in that application are linked during run-time. In C++ we support the static linking mechanism.

Dynamically linked programs are usually smaller in size (object code) than statically linked programs. However statically linked programs execute faster than dynamically linked programs. In parallel computers, like SIMD type of computers, the size of memory is typically less than in conventional computers. The advantage of shared code is considerable in such cases. In our example, the class Grad_student inherits from both Student and Faculty classes. Only a few additional data and functions can be defined while reusing existing code, and only one copy of Student and Faculty classes will be maintained.

Disadvantages

Name clashes A class and its ancestors can have the same names for data used and/ or the functions offered by them. The conflicts in the names have to be resolved and there are different approaches to solve the problem. One approach could be the prevention of any name clashes and force the user to name data and functions so that there are no clashes. Another approach could be that the user can rename data and functions if there are any clashes. Yet another approach could be to build a hierarchy of names and data and to use the most recent definition (the first definition reached when going from bottom to top and left to right) in the hierarchy as the default. The renaming approach is very useful because it not only resolves the naming conflicts, but also enhances the readability of the modules by specifying which data or function

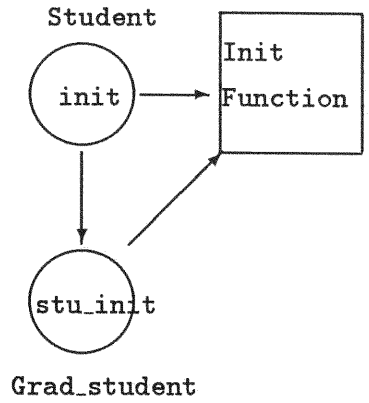

Figure 6: **Graphical Representation Of Renaming Functions**

is being used. The inheritance approach, while resolving the conflict, does not specify the data or function being used and the programs are less readable. So the user has to follow the hierarchy to comprehend which data or function is being used.

In our example both Faculty and Student can define a variable called "courseno". In the class Faculty, "courseno" could denote the course being taught while in the class Student, "courseno" could denote the course being attended. The "courseno" is a variable with multiple definitions in the class Grad_student and the conflict has to be resolved. The class Grad_student might have the need for both the ancestral "courseno" definitions and has to be specified without conflict in the class Grad_student.

Redefinition The functions inherited from the ancestors may be too general and hence could be defined again in the current module because of efficiency considerations. Redefinition could be used for resolving name conflicts and for readability. However, there is a distinct difference between renaming and redefining functions.

Consider the Fig. 6 in which a function "init" is defined for the class Student.

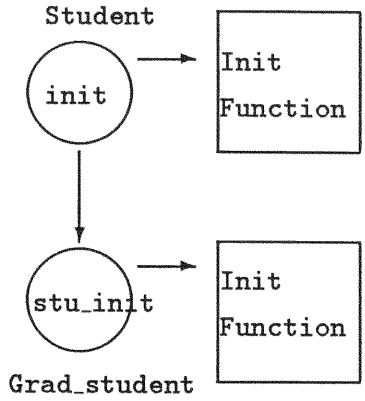

Figure 7: **Graphical Representation Of Redefining Functions**

Class Grad_student can either rename or redefine the function "init". When a function is *renamed* from "init" to "stu_init" in the Grad_student and "stu_init" is used in Grad_student, then the code specified for "init" (in class Student) is used. So there is only one copy of the function init.

When the function "init" is *redefined* as "stu_init" in Grad_student and "stu_init" is used in the Grad_student class, then the new definition specified for "stu_init" is used. There are two separate functions, one defined in the class Student and the other defined in the class Grad_student. This is depicted in Fig. 7. The definition of "init" in the class Student is different from the definition in the class Grad_student which is more suitable for that class. Hence, redefinition can be used as a method to achieve specialization.

Common ancestor Most of the languages resort to renaming and redefining to resolve conflicts. This however, does not resolve all the problems. Consider the following situation depicted pictorially in Figure 5. Module Grad_student inherits

from the modules Student and Faculty, which in turn inherit from the module Person. All the data and functions defined in module Person are available to the modules Student and Faculty. Module Grad_student inherits the data and functions defined in module Person twice, through the modules Student and Faculty. There will be conflicts in the names and hence have to be resolved. But the problem is because of the existence of a common ancestor and there should be no conflicts to be resolved. This problem must be solved by the compiler (interpreter) which should analyze and conclude that the conflicts are due to the common ancestor.

Scope rules Objects are uniquely identifiable. Within an object a special variable is defined to reference the object itself (e.g. *self*, *current*, *this*). In the context of inheritance, the meaning of such a self reference in a function becomes ambiguous. If a function containing a self reference is inherited by a subclass, then it is not clear whether it refers to the object at the subclass or the superclass level. The possible ambiguities are even greater in the presence of multiple inheritance. In addition, some languages allow the specification of a self reference at a superclass level (e.g. *super*). To resolve such ambiguities languages like C++ allow the user to specify the domain to which the function is to be associated with. An example is given below:

```
student::init();  /* specifies that init is to be associated */
                  /* with the class Student.                  */
Person::init(); /* specifies that init is to be associated */
                  /* with the class Person.                 */
```

So in the context of the child class Grad_student, the "init" function can be associated with the one defined in the class "student" or in class "Person" by explicit specification.

Type checking Type checking is another important issue in multiple inheritance. Let us suppose that "var1 := var2" is an assignment (like in Eiffel [Mey88]) in a function of a class where the types of "var1" and "var2" could be known only at execution time. If the function is inherited by another class, the types of "var1" and "var2" have to match. The rules for matching are more than the typical type checking rules used in conventional languages and the additional rules for matching can be specified by the language. The language can also specify when the checks for matching are carried out, either at compile time or run time.

Hence when we introduce multiple inheritance in C^* , all these conflicts have to be resolved in the extension of C^* to C^{++} .

4 PROGRAMMING ON THE CONNECTION MACHINE

The Connection Machine is an SIMD type computer system in which data-parallel programs can be executed. In this system each data element is assigned to a processor (either physical or virtual). This should reduce the execution time of the program and the decrease in time is usually proportional to the number of physical processors used.

4.1 Hardware Layout

The major components of the Connection Machine are the control unit, sequencer, instruction bus, processing elements, scalar memory bus, global result bus, interconnection network and I/O controllers [Gro89]. This is represented pictorially is Fig. 8.

4.1.1 Control Unit

The control unit is the front-end of the Connection Machine in which the program resides. All the serial code is executed in the front-end and the parallel instructions are passed to the sequencer. The front-end in the existing Connection Machines are either a VAX, a SUN, or a Symbolics system .

4.1.2 Sequencer

The sequencer receives the parallel instructions from the front-end. It interprets the instructions and produces a series of "nanoinstructions" which are broadcast to all

the processors through the instruction bus . The sequencer can also communicate with the back-end through the scalar memory and global result buses.

4.1.3 Processing Element

Each processing element consists of a processor and a memory associated with it. The ALU in each processor can operate on variable length operands. The memory is either 64K or 256K bits in existing Connection Machines. The processing elements, whose number in a Connection Machine can vary, communicate with each other through the interconnection network and with the sequencer through the instruction, scalar and global result buses.

4.1.4 Interconnection Network

The three mechanisms of interconnection between processors supported by the Connection Machine are

Router

This is the most general mechanism in which every processor can access the memory of every other processor. These memory accesses are simultaneous and could result in runtime errors (called collisions) if more than one processor tries to access the same memory location.

North East West South: NEWS

This is a faster and a structured mechanism in which processors are interconnected in a grid. NEWS supports upto 31 dimensions in the grid. There is no restriction on the number of processors in each dimension. However the product of the number of processors in each dimension should be equal to the number

of physical processor in the system. For example, if the number of processors in the Connection Machine is 64K, then 64x1024, 8x8x1024, 16x16x16 are some valid grid sizes. Special hardware makes NEWS communication faster than Router mechanism.

Scanning

This mechanism combines both communication and computation. Scanning could operate simultaneously on each row. The operations include addition, subtraction, multiplication and division. These operations are supported by special hardware related to the NEWS mechanism.

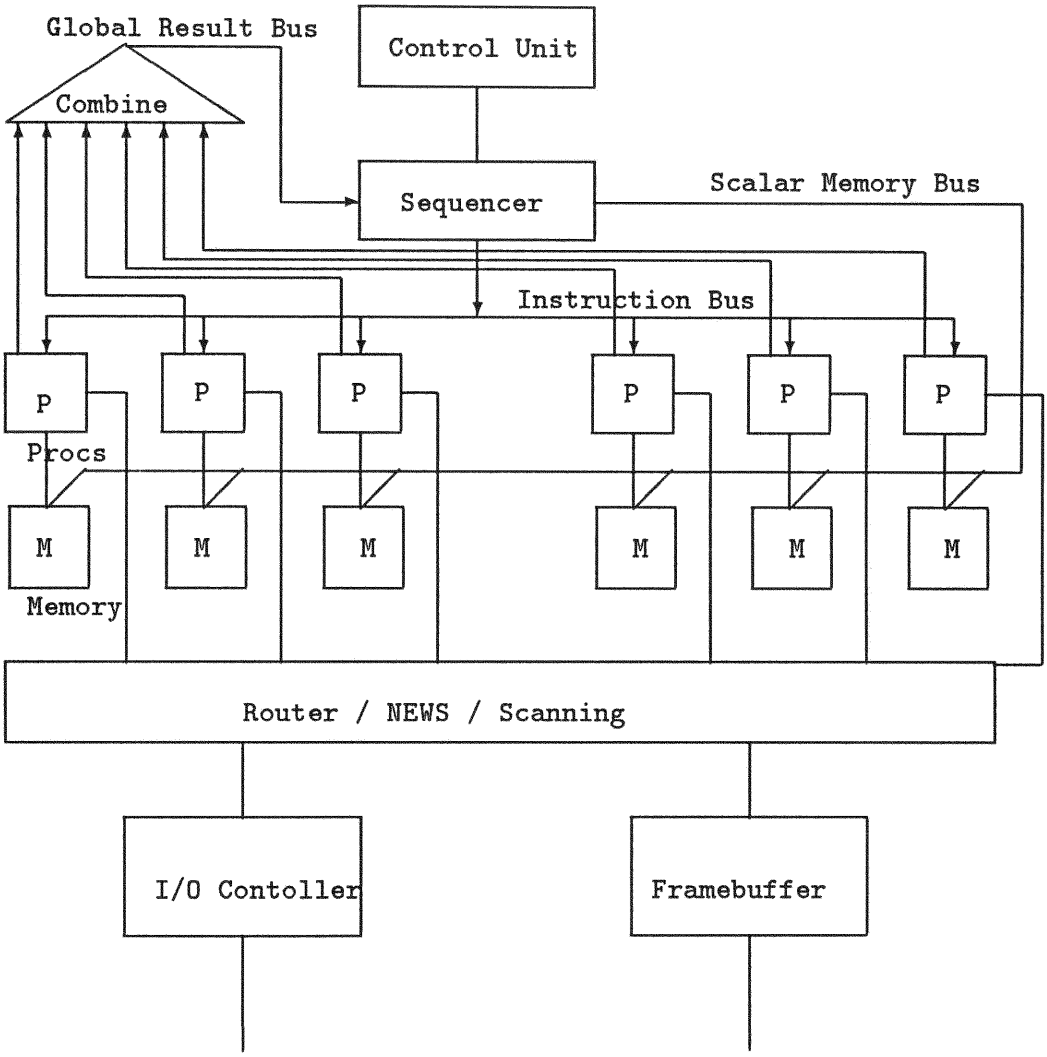

Figure 8: **Hardware Components of the Connection Machine**

4.2 C*

C* (pronounced C star) was designed with two important design factors[JG89]. One was to support data-parallel programming and the other was to be compatible with C. So C was extended to accommodate the specification of data-parallel programming. Some additional statement and operators were introduced to aid data-parallel programming.

C* introduces a data type to specify parallel data and functions called the "domain" and associated keywords to distinguish between scalar and parallel information (data as well as code). It also introduces a new statement to initiate parallel execution.

4.2.1 Domain Specification

The data type "*domain*" is used to specify parallel data and functions C* . These data and the functions are valid only when referred within the context of the domain.

The following is an example of the declaration of a domain (data and function associated with it) and the declarations of instances of the domain.

The domain specified models a student. Each student has a social security number, a list of courses taken (through all the semesters) and a list of courses taken in the current semester which is represented in C* as follows:

```
/* The following is the domain specification of a student.  */
/* Data variables are declared for soc_sec, GPA and courses */
/* taken. An index for the array defined is also used.      */

domain  student
{
   int ssn;
   int gpa;
   int current_grades[CURRENT];
```

```
    int current_courses[CURRENT];
    int INDEX;

    /* This function initializes the local variables. */
    void init()
    {
      INDEX = 0;
      ssn = 0;
      gpa = 0;
      for(i=0; i<CURRENT; i++)
      {
        current_grades[i] = 0;
        current_courses[i] = 0;
      }
    }

    /* This function updates the GPA based on semester GPA. */
    int update_gpa(int sem_gpa)
    {
       gpa += sem_gpa;
       return gpa;
    }

    /* This function calculates the semester GPA. */
    int calculate_sem_gpa()
    {
       int i, sem_gpa, sem_credit;

       sem_gpa = 0;
       sem_credit = 0;
       for(i=0; i<CURRENT; i++)
       {
          sem_gpa += current_grades[i] * current_courses[i];
          sem_credit += 1;
       }

       sem_gpa = sem_gpa / sem_credit;
       return sem_gpa;
    }
}
```

The domain student has five data members and three functions associated with it. More functions associated with this domain can be defined with the following syntax:

```
/* The scope of the function set_current_grades is    */
/* restricted to the domain student.                  */

void student::set_current_grades(int grade)
{
   current_grades[INDEX] = grade;
   INDEX += 1;
}
```

Once we have defined the domain, we should be able to specify the instances of the domain. Instances of the domain are declared by the following statement:

```
domain student twenty_students[20];
```

This statement declares twenty instances of student with each instance assigned to a processor (virtual or physical).

An experienced C^{++} programmer can note the similarities between C^{++} and C* . The development of C* from C was based on the extensions used by C^{++} . The declaration of a domain, specifying data and functions associated with the domain, creating instances of a domain and specifying the scope of domains and associated functions are all similar in C* and C^{++} . However C^{++} supports more object-oriented concepts than C* . C^{++} supports encapsulation of information in classes through the use **private, public** and **protected** variables [Lip89]. It also breaks the encapsulation through the use of **friends** specification. C^{++} also supports the dynamic binding of a function to an object through the use **virtual** keyword. C* does not support any of the previously mentioned principles [JG89] but is best suited to program the Connection Machine.

4.2.2 Parallel Statement

A statement is introduced in C*++ to direct the compiler to begin execution of parallel code. The syntax is described below:

```
.
.
.

/* total_gpa contains the sum of all the GPA's of all students */
mono int total_gpa;

total_gpa = 0;
[domain student].{
  init();
  set_current_grade(4);  /* Sets the grade to 4 pts */
  set_current_grade(3);  /* Sets the grade to 3 pts */
  total_gpa += calculate_sem_gpa(); /* Semester gpa is calc */
  /* Then all the semester GPAs are added. */
  /* The sum is added by the += operator */

  update_gpa(calculate_sem_gpa()); /* Update overall GPA */
}
.
.
.
```

[domain student]. is the compiler directive to begin parallel execution. The functions are executed only on all active instances of the domain student and the code they execute is called the "parallel code".

4.2.3 Additional Keywords

In C* , code is divided into serial and parallel. Serial code is executed by the front-end while the parallel code is executed by the active instances of the domain under consideration. Data is divided into scalar and parallel data, represented by two additional keywords " **mono** " and " **poly** ". Within the serial code, the default

type of data is "mono" while in the parallel code it is "poly". "Poly" variables used in serial code are accessed sequentially. In the example seen before, " total_gpa" is a scalar value and because it is used in parallel context it has to be declared as "mono" . In the case of "ssn", it is a parallel value declared in a parallel context and hence it is a "poly" value by default. It can also be declared in the domain Student as

poly int ssn;

4.3 Software Engineering in C*

In this section we will study how some software engineering and object-oriented concepts are supported in C* and why certain other concepts are not. The concepts considered are abstraction, modularity, encapsulation, software reuse through inheritance and information hiding.

Real world concepts can be abstracted as domains in C* . So C* allows the programmer to specify concepts as abstractions. Also modules can be represented as domains in C* . Different domains can be combined with each other in solving the problem and hence modular programs can be developed in C* . The concepts of abstraction and modularity are supported by C* since domains and functions associated with those domains can be specified in C* . These functions can be invoked by the function calls (message to the function).

In C* each instance of a domain (a processor) communicates with other instances of domain (processors) through the interconnection network. In this interconnection mechanism, any processor can access the memory of any other processor, which is contradictory to the concept of information hiding in object-oriented programming. It is the hardware of the Connection Machine which is wired this way and hence,

information hiding cannot be supported without changes to the hardware.

Software reuse is the major strength of inheritance. Dynamic binding of functions to domains is another advantage of inheritance. Because of dynamic binding, different instances of the same domain can be executing different functions simultaneously. This however cannot be supported by the Connection Machine because all instances of the same domain have to be executing the same instruction.

So C* while supporting abstraction and modularity, does not support the concepts of information hiding and inheritance. We will now study a mechanism to introduce partial inheritance in C* .

5 INTRODUCING INHERITANCE IN C*

In the previous section we have seen why C* does not support inheritance.

Software reuse is essential for fast and efficient development of programs. On the Connection Machine the high level language which supports some object-oriented concepts is C* . So if we could extend C* to include inheritance, then it is possible to combine the power of SIMD computing with software reuse.

So to extend C* to allow for software reuse, the language has to support the concepts of subclasses and specifications of the same. With the presence of subclasses, conflicts could arise. These conflicts can either be resolved by the user or by the compiler. In our approach we allow the user to specify subclasses and also allow the user to resolve the conflicts.

A translator from the extended language to C* has to be developed to compile and execute the programs in the extended language. In this thesis we develop a translator, called "csp", to translate programs written in the extended language to an equivalent C* program, which can then be compiled and executed on the Connection Machine.

5.1 C* : The Extended C*

We want to introduce "Inheritance" in C* , which necessitates the extension of the syntax and semantics of C* . In the design of C^{*++} , we wanted to modify existing structures in C* instead of creating new ones.

In the extended language, the user should be able to specify the classes from which to inherit. These classes are called "superclasses" or "superdomains". Since inheritance can result in naming conflicts, the user should be able to resolve the naming conflicts in data and functions. The user should also be able to selectively

inherit from the superdomains. Following these principles, C*++ has to be extended from C* to support the following:

1. To specify superdomain(s)

2. Selectively inherit functions from parent domains.

3. To rename the data inherited from parent domains.

To support the above points, few new reserved words were introduced. They are

1. NEWDOMAIN (to specify the C*++ domain).

2. SUPERCLASS (to specify the parent domain).

3. ALIAS (to rename the data inherited from the parent).

4. REDEFINE (to disinherit functions from the parent).

Let us now study the syntax of domain definition in C* and C*++ .

5.1.1 C*

The YACC-like grammar for the definition of a domain in C* is as follows:

```
domain      : DOMAIN      /* Reserved word domain */
              classname   /* Variable name for domain */
              block       /* Data & functions for domain */
            ;
```

We have already seen an example of a definition of a domain in C* .

5.1.2 Syntax of C*++

The major extension from C* to C*++ is in the definition of the domain. The syntax of C*++ can be specified in an YACC-like grammar as given below:

```
newdomain     :  NEWDOMAIN /* Reserved word newdomain */
                 classname /* Variable name for domain */
                 ancestor  /* List of ancestors & changes */
                 IS        /* Reserved word is */
                 block     /* Data & functions for domain */
              ;

classname     :  VARIABLE  /* Var name recognized by LEX */
              ;

ancestor      :  /* empty ancestor */
              |  SUPERDOMAIN class /*Reserved word superclass*/
                 /* followed by the classname and changes */
              ;

class         :  classname class1 /* Classname and changes */
              |  class class      /* Used for recursive def*/
              ;

class1        :  /* empty */
              |  {change}         /* Changes in parentheses*/
              ;

change        :  ALIAS aliaslist change1 /* Rename variable*/
              |  REDEFINE list  change1  /* Redefine funct */
              ;

change1       :  /* empty */
              |  change /* Used for recursion */
              ;

list          :  VARIABLE list1   /* Variable list in redefine*/
              ;

list1         :  ;                /* Used in recursion. */
              |  list         /* Recursion of list  */
              ;
```

```
aliaslist     :  VARIABLE VARIABLE aliaslist1
                 /* Alias oldname newname followed by recursion*/
              ;

aliaslist1    :  ;            /* Separate aliases by ; */
              |  ; aliaslist /* Recursion of aliases  */
              ;
```

Here NEWDOMAIN, IS, VARIABLE, REDEFINE, SUPERCLASS and ALIAS are all tokens. VARIABLE is any standard variable name while all other tokens are just lower case strings (like "newdomain" for "NEWDOMAIN"). The definition for the "block" in C^{*++} is the same as that C^{*} . The differences, as can be seen, are in the use of "ancestor" and "IS" symbols in the domain definition to allow for multiple inheritance definition.

The following is an example of a domain specification in C^{*++} . The domain Grad_student is specified as a child domain of the "student" domain specified earlier in the thesis.

```
newdomain grad_student
superdomain student {redefine init; alias gpa stu_gpa;}
is
{
  int thesis_code;
  int thesis_rating;

  init()
  {
    int i;

    i = 0;
    thesis_code = 6000;
    thesis_rating = 5;
    ssn = 100000000;
    stu_gpa = 3;
    for (i=0; i<CURRENT; i++)
    {
```

```
        current_grades[i] = 0;
        current_courses[i] = 0;

    }
  }
}
```

In the above example, the function "init" is not inherited from the student domain. And the data variable "gpa" is called as "stu_gpa" in the domain grad_student. All other data and functions of student are inherited without any changes.

5.2 Translation

In the previous sections we have seen how multiple inheritance can be specified in C^{*++} . However there is no compiler for C^{*++} . One approach to execute the code written in C^{*++} on the Connection Machine would be to translate the C^{*++} programs into equivalent C* programs. These transformed equivalent programs can then be compiled with the C* compiler and then executed on the Connection Machine. In this thesis we use such an approach. In this section we will study about the translation of C^{*++} to C* . The program specified in C^{*++} is translated into C* by the preprocessor "csp". The code generated is then compiled with C* compiler "cs". The translator (csp) uses the information about "superdomains", "aliases" and "redefines" specified by the user in the C^{*++} program to create an equivalent C* program.

In C^{*++} , for each new domain a list of superdomains can be specified. The translator maintains a list of these superdomains for each domain. In this new domain, data and functions inherited from the ancestral domains can be renamed. The new domain can also disinherit functions from the ancestral domains. So for every new domain, the translator maintains a list of data and functions renamed and the functions

disinherited for every ancestral domain. At the end of all definitions, the translator has information about every new domain defined and all the ancestral domains and the relevant information about renaming and redefinition. Once the translator has all this information, it then starts to build the domains which are C* compatible.

In C* , the data and the executable statements have to be separated. So for every new domain with ancestral classes, the data definitions from the ancestral domains have to be redefined in the new C* compatible domain (CC domain) followed by the executable statements. So in the expansion of the new domain to a CC domain, the data definition for each superdomain (with necessary aliasing) is redefined. Then the data defined in the C^{*++} domain is copied into the CC domain. At this point the data definition for the new expanded CC domain is complete. Then the executable statements from each of the superdomains are copied onto the CC domain (all the aliased variables have their new names in the CC domain). However redefined functions are not copied into it. Finally the functions of the new domain defined in C^{*++} are copied onto the CC domain. At this point the definition of the new expanded CC domain is complete. The same process is carried out on all the new domains till there are none. The rest of the C^{*++} program is copied without any changes to the CC domain. This approach is depicted if Fig. 9.

The translator "csp" parses the C^{*++} program for building information about the ancestors for every new domain and the relevant renaming and redefining information. The translator also checks for syntax and detects most of the syntactic errors. Once a syntax error is detected, the translator gives a suitable error message and stops execution. The translator does not carry out any semantic checking and all the semantic errors are detected by the C* compiler.

The translation process, depicted in Fig.10, shows the translation from C^{*++} pro-

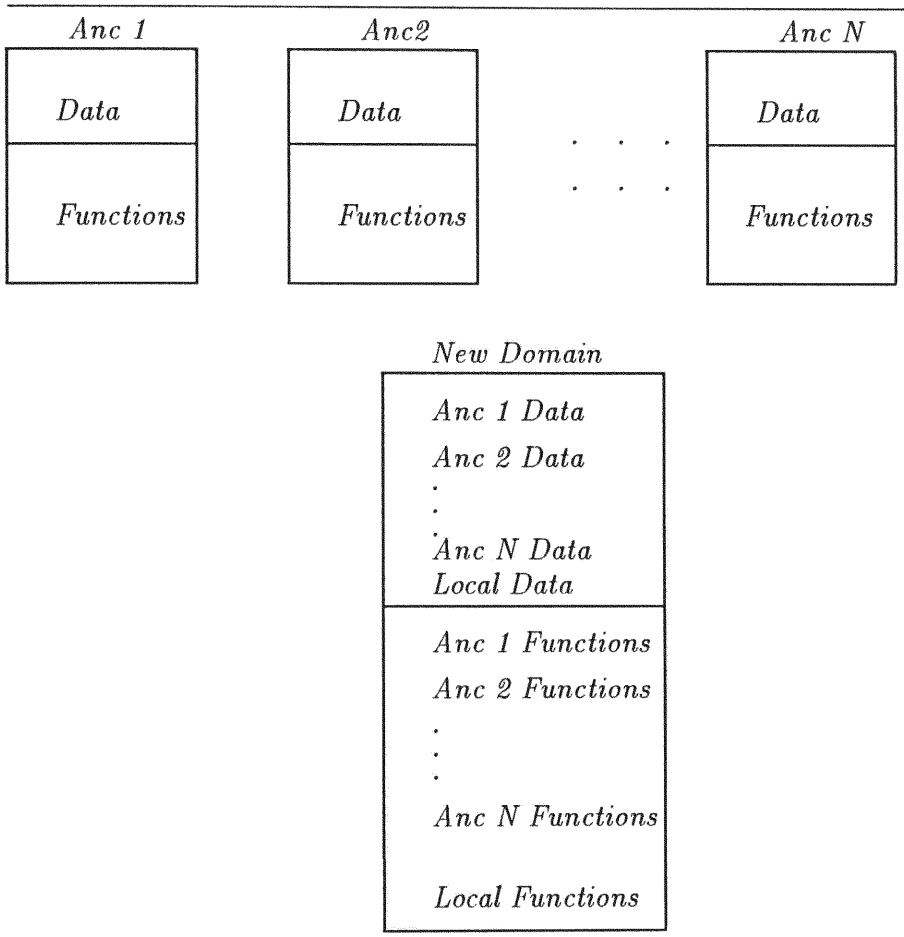

Figure 9: **Graphical Representation of the Translation**

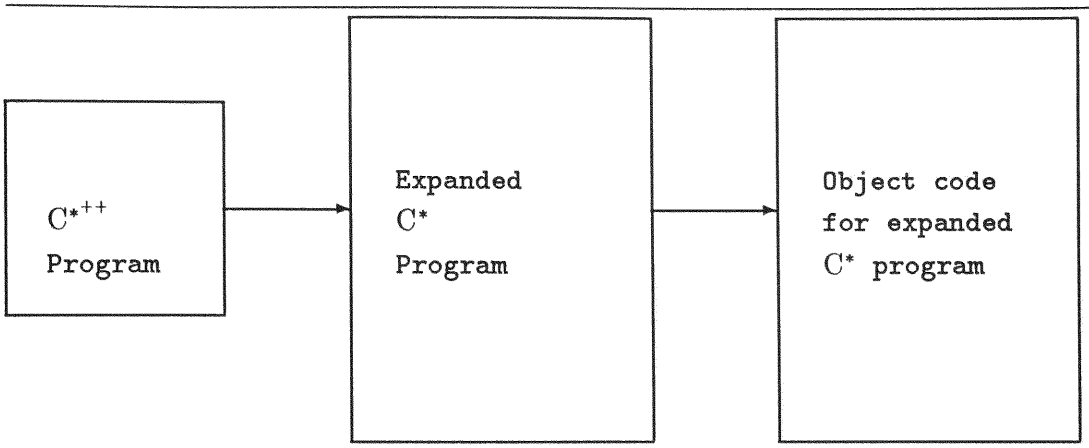

Figure 10: **Translation Phases**

gram to an expanded C* program which is then compiled into object code. A more suitable approach would be to translate from C*++ to object code which should reduce the size of the object code. This translation however cannot support dynamic binding of a function to an object (a back-end processor) on the Connection Machine. This constraint is because of the single instruction restriction imposed by the architecture of the Connection Machine.

We illustrate the benefits of using C*++ over C* to develop software on the Connection Machine through an example.

5.2.1 Example

We use the program of the propagation of sound through water as our example. The ocean can be divided into a grid and each element in the grid can be assigned to a processor. Each processor contains information about the volume or the intensity of sound at that point. The value of sound in each processor is dependent on the value of sound in the neighboring processors in the left (say). We can create a sound

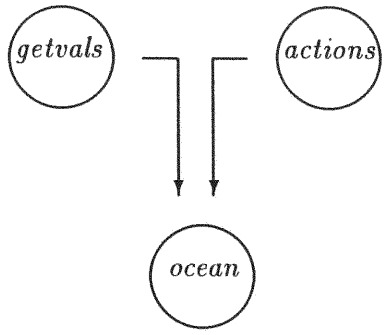

Figure 11: **Graphical Representation of Marching of Sound**

in the leftmost column in the grid (This can be achieved by modifying the value of sound in all the processors in the leftmost column) and study the propagation of it through water. Since the volume of sound in each processor is dependent on its left neighbors and since we have modified the volume of sound in the leftmost column, the volume of sound in each processor changes. The parameters which affect the volume of sound are the functions on which the propagation is based on and the number of neighboring processors which affect it. As a result of following a modular approach in developing software, we can create two domains to reflect our analysis. One domain could implement the methods to get values from the neighboring processors while another domain could evaluate the intensity of sound at a point. We can then create a new domain which would inherit from both these domains and implements additional methods (if needed) to analyze the propagation of sound through water.

This can be implemented in C^{*++} as three different domains. This is depicted in the Fig. 11. The first domain specifies the methods associated with getting values from the neighboring processors while the second domain specifies methods regarding the actions (or functions) used to simulate the marching of sound through water.

As can be seen, software is developed by following the principles of abstraction and modularity. We can then create a new domain, which inherits from the two domains defined, to fully analyze the propagation of sound. Again, we follow the principle of software reuse through inheritance. So program development in C^{*++} easily follows the principles of software engineering and object-oriented programming .

We now define the various domains in C^{*++} . The domain "getvals" defines the methods for getting values from the neighboring processors.

```
/* This domain defines all the get indices */
/* Defines few vars and funcs to get vals  */
/* from the left, left top, left bottom    */
/* and all in the left col.                */
DOMAIN getvals procs[NO_PROCS];
NEWDOMAIN getvals IS
{
   INT leftind, left_top_ind;
   INT left_bot_ind, selfid;
   /* These are pointers to the indices of the elements on */
   /* the left, left top and the left bottom. there is also a */
   /* self pointer */

   /* Function initializes all vars */
   VOID init()
   {
     leftind      = -1;
     left_top_ind = -1;
     left_bot_ind = -1;
     selfid       = this - &procs[0];
   }

   /* Gets val from the left element in grid */
   VOID get_left()
   {
     INT index;

     index = selfid - 1;
     IF (index >= 0 && (((index + 1) % COLS) != 0))
        leftind = index;
     ELSE
```

```
        leftind = -1;
    }

    /* Gets val from the left top element in grid */
    VOID get_left_top()
    {
      INT index;

      index = selfid - COLS - 1;
      IF (index >= 0 && (((index + 1) % COLS) != 0))
         left_top_ind = index;
      ELSE
         left_top_ind = -1;
    }

    /* Gets val from the left bottom element in grid */
    VOID get_left_bottom()
    {
      INT index;

      index = selfid + COLS - 1;
      IF (index < (ROWS * COLS) && (((index + 1) % COLS) != 0))
         left_bot_ind = index;
      ELSE
         left_bot_ind = -1;
    }


};
```

We can now define the domain which defines the methods of calculating the marching function. In this domain, it is assumed that all the necessary values from neighboring processors are available. The following is the definition in C^{*++} .

```
DOMAIN actions proc_acs[NO_PROCS];
/* This domain defines the function to evaluate the */
/* neighbors or all in the left col.                */
NEWDOMAIN actions is
{
    INT value, leftval, left_top_val, left_bot_val, leftall;
    INT lind, ltind, lbind;
```

```
  /* initializes the value 0 */
  VOID init()
  {
     value = 0;
  }

  /* The value of sound at a point is 3/4 its old value */
  /* and the average of the value of the elements on the */
  /* left, left top and left bottom  only */
  /* Evaluates the local val based on neighbors only */
  VOID eval_neigh()
  {
    value = 3 * value / 4;
    IF (lind >= 0 && (lind < (ROWS *COLS)))
      leftval      = proc_acs[lind].value;
    IF (ltind >= 0 && (ltind < (ROWS *COLS)))
      left_top_val = proc_acs[ltind].value;
    IF (lbind >= 0 && (lbind < (ROWS * COLS)))
      left_bot_val = proc_acs[lbind].value;

    value += (leftval + left_top_val + left_bot_val) / 3;
  }

  /* Value at a point is 3/4 its old value plus the average */
  /* of all the values in the left column */
  VOID eval_all()
  {

    INT index , column, i;

    leftall = 0;
    column  = ((this - &proc_acs[0]) % COLS) - 1;
    for (i=0; i<ROWS; i++)
    {
      index = (i * COLS) + column;
      IF (lind >= 0)
        IF (index < (ROWS * COLS) )
         leftall += proc_acs[index].value;
    }
    value = 3 * value / 4;
    value += leftall / ROWS;
  }
};
```

We can now define a new domain which inherits from both the domains defined. It combines the values obtained from the neighboring processors and combines with the marching function to analyze the propagation of sound. It also defines the main function. In the main function, the domains are executed and the values are printed in each step of the iteration.

```
DOMAIN ocean underwater[NO_PROCS];
/* Declares NO_PROCS instances of the domain ocean */
/* This domain inherits from getvals and actions */
/* It aliases the procs var in getvals           */
/* It aliases all the local data in actions      */
/* This is because we want to use the vals obtained*/
/* from the domain getvals                         */
/* It defines two func to eval the sound           */
NEWDOMAIN ocean  SUPERCLASS
getvals {  ALIAS procs underwater;}
actions {  ALIAS value vos; REDEFINE init;
           ALIAS proc_acs underwater;
}
IS
{
   INT var1;

   /* This function matches the indices */
   /* between the 2 parent domains.     */
   VOID match_indices()
   {
     lind = leftind;
     ltind = left_top_ind;
     lbind = left_bot_ind;
   }

   /* Evaluates based on the neighbors */
   VOID restricted_eval(INT loop)
   {
     INT i;

     FOR(i=0; i< loop; i++)
     {
       eval_neigh();
     }
```

```
    }

    /* Evaluates based on the left column */
    VOID all_eval(INT loop)
    {
      INT i;

      FOR(i=0; i< loop; i++)
      {
         eval_all();
      }
    }
};


/* This is the main loop, Some vals are assigned */
/* to the leftmost column. Then the vals are     */
/* printed. eval is done 10 times & vals are     */
/* printed again.                                */
main()
{
    INT i, j, tmp;

    [DOMAIN ocean].{
      FOR(i=0; i< ROWS; i++)
         underwater[i*COLS].vos = i * 1000;
    }

    FOR(i=0; i<ROWS; i++)
    {
      FOR(j=0;j<COLS;j++)
         printf("  %d ", underwater[i*COLS+j].vos);
      printf("\n");
    }

    [DOMAIN ocean].{
      init();
      get_left();
      get_left_top();
      get_left_bottom();
      match_indices();
    }
    FOR (tmp=0; tmp<LOOPCOUNT; tmp++)
    {
```

```
      [DOMAIN ocean].{
        restricted_eval(1);
      }
      FOR(i=0; i<ROWS; i++)
      {
        FOR(j=0;j<COLS;j++)
           printf("  %d ", underwater[i*COLS+j].vos);
        printf("\n");
      }
      printf("\n\n");
   }
}
```

We defined only the minimum number of functions for the domain "ocean". We can also define different propagation functions as superclasses of the domain "ocean" and inherit from all of them. Then we can separate different functionality into different domains which follows the principles of abstraction and modularity. This helps in the processes of design and development. Since minimum amount of new software has to be written, the process of software development is faster in time and better in quality. The processes of debugging and testing also become faster. To illustrate these advantages we present the expanded version of the domain "ocean".

```
DOMAIN ocean underwater[NO_PROCS];
/* This DOMAIN inherits from getvals and actions */
/* It aliases the procs var in getvals           */
/* It aliases all the local data in actions      */
/* This is because we want to use the vals obtained*/
/* from the DOMAIN getvals                         */
/* It defines two func to eval the sound           */

 /*  ocean  CSP DOMAIN BEGINS */
DOMAIN ocean
{


   INT leftind, left_top_ind;
```

```
INT left_bot_ind, selfid;

INT vos, leftval, left_top_val, left_bot_val, leftall;
INT lind, ltind, lbind;

INT var1;

VOID init()  /* BEGIN CSP init */

{
  leftind      = -1;
  left_top_ind = -1;
  left_bot_ind = -1;
  selfid       = this - &underwater[0];
}  /* END CSP FUNCTION */


/* Gets val from the left element in grid */
VOID get_left()  /* BEGIN CSP get_left */

{
  INT index;

  index = selfid - 1;
  IF (index >= 0 && (((index + 1) % COLS) != 0))
     leftind = index;
  ELSE
     leftind = -1;
}  /* END CSP FUNCTION */


/* Gets val from the left top element in grid */
VOID get_left_top()  /* BEGIN CSP get_left_top */

{
  INT index;

  index = selfid - COLS - 1;
  IF (index >= 0 && (((index + 1) % COLS) != 0))
     left_top_ind = index;
  ELSE
     left_top_ind = -1;
}  /* END CSP FUNCTION */
```

```
/* Gets val from the left bottom element in grid */
VOID get_left_bottom()  /* BEGIN CSP get_left_bottom */

{
  INT index;

  index = selfid + COLS - 1;
  IF (index < (ROWS * COLS) && (((index + 1) % COLS) != 0))
     left_bot_ind = index;
  ELSE
     left_bot_ind = -1;
}  /* END CSP FUNCTION */




/* Evaluates the local val based on neighbors only */
VOID eval_neigh()  /* BEGIN CSP eval_neigh */

{
  vos = 3 * vos / 4;
  IF (lind >= 0 && (lind < (ROWS *COLS)))
    leftval      = underwater[lind].vos;
  IF (ltind >= 0 && (ltind < (ROWS *COLS)))
    left_top_val = underwater[ltind].vos;
  IF (lbind >= 0 && (lbind < (ROWS * COLS)))
    left_bot_val = underwater[lbind].vos;

  vos += (leftval + left_top_val + left_bot_val) / 3;
}  /* END CSP FUNCTION */


/* Evaluates the local val based on the val in the */
/* left column.                                    */
VOID eval_all()  /* BEGIN CSP eval_all */

{

  INT index , column, i;

  leftall = 0;
  column  = ((this - &underwater[0]) % COLS) - 1;
```

```
  FOR (i=0; i<ROWS; i++)
  {
    index = (i * COLS) + column;
    IF (lind >= 0)
      IF (index < (ROWS * COLS) )
       leftall += underwater[index].vos;
  }
  vos = 3 * vos / 4;
  vos += leftall / ROWS;
}  /* END CSP FUNCTION */

VOID match_indices()  /* BEGIN CSP match_indices */

{
  lind = leftind;
  ltind = left_top_ind;
  lbind = left_bot_ind;
}  /* END CSP FUNCTION */


/* Evaluates based on the neighbors */
VOID restricted_eval(INT loop)  /* BEGIN CSP restricted_eval */

{
  INT i;

  FOR(i=0; i< loop; i++)
  {
    eval_neigh();
  }
}  /* END CSP FUNCTION */


/* Evaluates based on the left column */
VOID all_eval(INT loop)  /* BEGIN CSP all_eval */

{
  INT i;

  FOR(i=0; i< loop; i++)
  {
     eval_all();
  }
}  /* END CSP FUNCTION */
```

```
}

 /*  ocean  DOMAIN ENDS  */




;


/* This is the main loop, Some vals are assigned */
/* to the leftmost column. Then the vals are     */
/* printed. eval is done 10 times & vals are     */
/* printed again.                                */
main()
{
   INT i, j, tmp;

   [DOMAIN ocean].{
     FOR(i=0; i< ROWS; i++)
        underwater[i*COLS].vos = i * 1000;
   }

   FOR(i=0; i<ROWS; i++)
   {
     FOR(j=0;j<COLS;j++)
        printf("  %d ", underwater[i*COLS+j].vos);
     printf("\n");
   }

   [DOMAIN ocean].{
     init();
     get_left();
     get_left_top();
     get_left_bottom();
     match_indices();
   }
   FOR (tmp=0; tmp<COLS; tmp++)
   {
     [DOMAIN ocean].{
       restricted_eval(1);
```

```
    }
    FOR(i=0; i<ROWS; i++)
    {
      FOR(j=0;j<COLS;j++)
         printf("  %d ", underwater[i*COLS+j].vos);
      printf("\n");
    }
  }
}
```

5.2.2 Sample Run

The above C*++ program was translated using the translator "csp". The command used was as follows:

% csp < filename

where "filename" is the C*++ program. The output is produced in the file "cspgen" in the working directory. The generated program was then transferred to the front-end of the Connection Machine. The converted C* program was then compiled on the front-end of the Connection Machine by using the following command:

% cs filename.cs

where "filename.cs" is the C* program generated by the translator "csp".

After the compilation the object code was executed on the Connection Machine. This is done by first "attaching" to the Connection Machine through the command

% cmattach -b

(-b is an option for the cmattach command). Once attached to the Connection Machine, it is possible to execute the object code. The example program was run on the Connection Machine with 32 processors (4x8) for 15 iterations. The initial value in the leftmost column are 0, 1000, 2000 and 3000. The value in all the other elements is zero. In each step of the execution, the value at any element is based on

the algorithm specified before. The output of a sample run is presented below.

0	0	0	0	0	0	0	0
1000	0	0	0	0	0	0	0
2000	0	0	0	0	0	0	0
3000	0	0	0	0	0	0	0
0	250	0	0	0	0	0	0
750	750	0	0	0	0	0	0
1500	1500	0	0	0	0	0	0
2250	1250	0	0	0	0	0	0
0	374	249	0	0	0	0	0
562	1124	624	0	0	0	0	0
1125	2249	874	0	0	0	0	0
1687	1874	687	0	0	0	0	0
0	420	560	218	0	0	0	0
421	1264	1404	436	0	0	0	0
843	2529	1966	546	0	0	0	0
1265	2107	1545	390	0	0	0	0
0	420	841	654	163	0	0	0
315	1263	2106	1309	299	0	0	0
632	2527	2948	1637	342	0	0	0
948	2106	2316	1169	233	0	0	0
0	393	1050	1226	612	115	0	0
236	1183	2631	2454	1123	200	0	0
474	2368	3684	3069	1284	218	0	0
711	1974	2895	2192	875	143	0	0
0	353	1180	1839	1378	519	78	0
177	1064	2958	3681	2528	904	133	0
355	2131	4144	4603	2891	983	140	0
533	1776	3256	3288	1971	646	90	0

0	308	1239	2413	2412	1365	413	52
132	930	3104	4830	4426	2377	700	87
266	1863	4350	6041	5060	2584	738	90
399	1553	3418	4316	3450	1699	474	57

0	264	1238	2894	3619	2732	1244	317
99	796	3103	5795	6639	4756	2106	527
199	1596	4348	7247	7591	5171	2217	544
299	1330	3416	5178	5176	3401	1425	344

0	222	1193	3255	4886	4613	2805	1074
74	671	2991	6518	8962	8029	4743	1786
149	1346	4191	8151	10247	8729	4993	1844
224	1121	3293	5824	6988	5741	3210	1168

0	184	1117	3486	6107	6920	5263	2691
55	558	2802	6981	11201	12044	8899	4473
111	1120	3927	8731	12808	13095	9368	4619
168	933	3085	6238	8734	8613	6024	2926

0	151	1022	3593	7196	9516	8688	5558
41	459	2566	7196	13199	16561	14688	9236
83	923	3597	9001	15093	18006	15463	9536
126	768	2826	6430	10292	11844	9944	6042

0	123	918	3590	8094	12235	13035	10012
30	374	2307	7192	14846	21291	22036	16636
62	754	3234	8996	16975	23149	23199	17175
94	628	2541	6427	11576	15229	14920	10882

0	99	812	3498	8765	14910	18157	16276
22	302	2042	7008	16078	25946	30695	27044
46	611	2863	8767	18384	28210	32315	27919
70	509	2250	6263	12537	18558	20784	17690

0	79	709	3336	9199	17392	23830	24419
16	242	1783	6685	16876	30265	40287	40574

```
34    492    2502    8363    19297    32906    42414    41887
52    409    1966    5975    13159    21648    27279    26541


 0     63     611    3124     9404    19562    29786    34343
12    193    1540    6261    17252    34040    50355    57062
25    394    2161    7834    19727    37011    53014    58909
39    327    1699    5597    13453    24349    34097    37328
```

As we can see, the sound propagates from left to right and the volume of sound changes for each iteration. The same program can be executed for different marching functions and for different number of iterations.

The classes specified here can be reused in other cases. For example, we could specify a different "action" domain while maintaining the same "getvals" domain. In such a case only ONE new domain has to specified while the other domains can be reused. Similarly the "getvals" domain can be changed while maintaining the same "actions" domain. These domains can also be used in other applications like the analysis of flow of heat in sheet metal. By reusing domains, minimum amount of new software has to be rewritten and this is the main advantage of programming in C^{*++} than in C^{*} .

6 CONCLUSION

In this thesis we have discussed the need for multiple CPU computing. We have also seen how object-oriented programming closely matches the concepts of multiple CPU computing.

C* on the Connection Machine is an example of such a matching between object-oriented programming and multiple CPU computing. However C* does not support the concepts of information hiding and software reuse. We have seen the advantages of software reuse, especially in a parallel programming environment. This thesis introduces extensions to C* to include multiple inheritance in the language to make use of advantages of software reuse. Such an extension to C* is our language C*$^{++}$. It is our belief that C*$^{++}$ is a very useful language with substantial advantages over other languages to program the Connection Machine. It makes the process of designing newer domains easier. Since the amount of software to be written is smaller, the production time of quality software is lesser. New software to be written is smaller and this decreases the chances of bugs. The debugging process also gets easier because the bugs could be confined to a smaller part of the code.

To execute C*$^{++}$ programs on the Connection Machine we also introduced a translator from C*$^{++}$ to C*$^{++}$ in this thesis. The translator however makes the user responsible for understanding information about the data and functions inherited. The user is responsible for resolving the conflicts in data and functions in the classes. So a knowledge about the domains inherited is needed and hence information about ancestral domains is not totally hidden from the descendant domains. And the object size is not proportional to the size C*$^{++}$ program and is based on the size of the translated C* program.

C^{*++} , though an extension to C^* , is not a complete extension. More features can be added to C^{*++} and as a part of the future work, the current translator could be made more user-friendly. The current translator makes the user responsible for resolving the conflicts. In resolving those conflicts, the translator could work interactively with the user. Further advances are possible when the Connection Machine is made multiuser (as being currently examined). In such a case, different instances of the same domain can execute different functions simultaneously which could make the processing even faster. C^{*++} would be even more powerful if different instances of different domains can execute simultaneously. It would then be feasible to create objects dynamically and also to support dynamic binding of functions to objects on such an architecture. This architecture is similar to the partitioned SIMD machine (it is an SIMD machine which is divided into many sub-SIMD machines. Each sub-SIMD machine executes its own instruction in parallel and all such sub-SIMD machines execute concurrently). It is our belief that C^{*++} would be very efficient to program such machines too.

References

[Agh86] Gul Agha. *Actors.* The MIT Press, 1986.

[Boo86] G. Booch. Object-oriented development. *IEEE Transactions on Software Engineering*, SE-12(2):211–221, February 1986.

[Gol84] Adele Goldberg. *SMALLTALK-80.* Addison Wesley, 1984.

[Gro89] Connection Machine System Group. Connection machine technical summary. Technical report, Thinking Machines Corporation, 1989.

[JG89] J.Rose and G.Steele. C* : An extended c language for data parallel programming. Technical report, Thinking Machines Corporation, 1989.

[Lip89] Stanley B. Lippman. *C++ Primer.* Addison Wesley, 1989.

[Lis90] Christine Lisetti. *Private Conversation.* 1990.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988.

[MW88] David Maier and David Warren. *Computing with Logic.* Benjamin Cummings Publishing Company, 1988.

[Sto80] Harold S. Stone. *Introduction to Computer Architecture.* SRA Computer Science Series, 1980.

[YT87] Akinori Yonezawa and Mario Tokoro, editors. *Object-Oriented Concurrent Programming.* The MIT Press, 1987.

7 VITA

Title of Thesis

DATA PARALLEL PROGRAMMING WITH MULTIPLE INHERITANCE ON THE CONNECTION MACHINE

Full Name

Sanjay Girimaji

Place and Date of Birth

Bangalore, India; April 15, 1963

Colleges and Universities

Indian Institute of Science, Bangalore, India (1983-1986)

B.E. in Computer Science.

Florida International University, Miami, FL (1988-present)

M.S. in Computer Science.

Professional Organisation

Association of Computing Machinery

Publication

ARDOCS: Application of Hypertext using Object-Oriented Programming, ACM South East Regional Conference '89, Atlanta.

Major Department

Computer Science

Date_____ Signed_____

8 APPENDIX

The source programs (the LEX and the YACC specifications) are submitted separately. The C^{*++} programs for the simulation of the propagation of sound through water are also submitted separately.