DISTRIBUTED EXECUTION ENVIRONMENTS

FOR THE CODE 2.0

PARALLEL PROGRAMMING SYSTEM

APPROVED BY
SUPERVISING COMMITTEE:

To my parents

DISTRIBUTED EXECUTION ENVIRONMENTS
FOR THE CODE 2.0
PARALLEL PROGRAMMING SYSTEM

by

RAJEEV MANDAYAM VOKKARNE, B. E.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF ARTS

THE UNIVERSITY OF TEXAS AT AUSTIN

May 1995

Acknowledgments

I would like to thank my advisor Professor J. C. Browne for his constant encouragement and guidance. He was a great source of ideas as well as inspiration.

Also, I would like to express my gratitude to Dr. Peter Newton who helped me understand the CODE system and outlined a basic framework for my thesis implementation. He has also been of great help at every stage of my thesis work.

I would like to thank Dr. John Werth for his time and suggestions.

I would also like to thank Dr. Syed Irfan Hyder for valuable discussions and suggestions.

Finally, I would like thank my parents for being very supportive of me.

DISTRIBUTED EXECUTION ENVIRONMENTS

FOR THE CODE 2.0

PARALLEL PROGRAMMING SYSTEM

Rajeev Mandayam Vokkarne, M.A.

The University of Texas at Austin, 1995

Supervisor: James C Browne

Writing parallel programs which are both efficient and portable has been a major barrier to effective utilization of parallel computer architectures. One means of obtaining portable parallel programs is to express the parallelism in a declarative abstract manner. The conventional wisdom is that the difficulty of translation of abstract specifications to executable code leads to loss of efficiency in execution. This thesis demonstrates that programs written in the CODE 2.0 representation where parallel structure is defined in declarative abstract forms can be straightforwardly compiled to give efficient execution on the distributed execution environment defined by the Parallel Virtual Machine (PVM) system.

The CODE 2.0 model of programming casts parallel programs as dynamic hierarchical dependence graphs where the nodes are sequential computations and the arcs define the dependencies among the nodes. Both partitioned and shared name spaces are supported. This abstract representation of parallel structure is independent of implementation architecture. The challenge is to compile this abstract parallel structure to an efficient executable program. CODE 2.0 was originally implemented on the Sequent Symmetry

shared memory multiprocessor and was shown to give executable code which was competitive with good hand coded programs in this environment.

This thesis demonstrates that CODE 2.0 programs can be compiled for efficient execution on a distributed memory execution environment with a modest amount of effort. The environment chosen for this demonstration was PVM. PVM was chosen because it is available on a variety of distributed memory parallel computer architectures. Development of the translator from CODE 2.0 to the PVM execution environment required only a modest amount of effort. Translations to other distributed execution environments can probably be accomplished with a few man-weeks of effort. The efficiency of the executable is demonstrated by comparing the measured execution time of several parallel programs to hand-coded versions of the same algorithms.

Table of Contents

List of Tables

List of Figures

Chapter 1. Introduction

New parallel computer architectures have the capability of performing very high speed computation. However, they usually offer tools and primitives which are architecture and implementation-specific. Writing parallel programs which are both efficient and portable is often a contradictory goal. It is often the case that programs that run very efficiently under one parallel architecture are not portable to another architecture. Architectures for parallel computers are still changing rapidly. Moving to a new parallel architecture often entails rewriting all existing programs. This offsets the benefits of moving to a newer and better architecture.

To help ease the burden of moving across architectures, the programming of these machines must be decoupled from the underlying architecture. Architecture-independent declarative abstractions are a means of achieving this goal. However, in order to make it worthwhile to use them, these abstractions must provide a programming model which provides the flexibility and expressiveness of architecture-dependent primitives while not being biased towards any particular architecture. The programming model must be easy to use and should support the creation of an efficient executable for any reasonable target architecture. Also, the programming model must not be biased towards either shared-memory machines or distributed-memory machines. However, it is a common belief that such generalized abstractions do not support creation of efficient runtime structures.

The CODE 2.0[New93a] abstract parallel programming system has been shown to yield programs with execution efficiency comparable to hand-coding for shared-memory multiprocessors. This thesis extends these results by demonstrating that programs with efficiency comparable to hand-coded ones can be obtained for CODE 2.0 programs for a popular distributed memory programming environment, the Parallel Virtual Machine (PVM) [Gei94]. This compiler from CODE 2.0 to PVM was created by the individual efforts of the author of this thesis in a few months of work.

1.1 CODE 2.0

CODE 2.0 (CODE) is a graphical retargetable parallel programming system. It facilitates a compositional approach to programming. Sequential units of computation are composed to form a parallel program where dependencies are specified by means of arcs. This abstract declarative language is independent of any architecture. It takes an integrated textual/visual approach to parallel programming. The user writes parallel programs by drawing an graph which represents the relationships between the various units of computation. The structure of the graph captures major elements of the parallel structure of the program. The graph serves as a template which is used as a framework for creating dynamic structures at runtime.

CODE has been implemented using a object-oriented design. It uses a class hierarchy to represent the abstract declarative structure. This is extremely helpful when writing translators for CODE. For each target architecture, a translate method is added for every class in the hierarchy. This means that architecture specific optimizations may be made without affecting code produced for other architecture. This is of utmost help when trying to produce efficient code for diverse architectures. The translate methods at the root of the abstract syntax tree invoke translate methods further down the hierarchy. In this manner, the entire tree is traversed and translated into executable code.

1.2 PVM

Parallel Virtual Machine (PVM) is a software system that permits a network of heterogeneous network of parallel and serial computers to appear as a single concurrent computational resource. PVM consists of two parts: a daemon process that any user can install on a machine, and a user library that contains routines for initiating processes on other machines for communicating between processes and changing the configuration of machines.

PVM is supported by a large and diverse set of distributed memory architectures. It is extensively used in academia and industry. These were the factors in selecting it as the target architecture in the current implementation.

Chapter 2 provides a more detailed overview of CODE and PVM.

A CODE translator already exists for the shared-memory architecture of the Sequent Symmetry. It has been shown to generate programs comparable in efficiency to hand-coded programs on the Sequent [New93a]. We show that CODE can also be used to produce an efficient executable for a distributed execution environment like PVM.

1.3 Distributed Execution Environment

This thesis deals with the problem of compiling CODE graphs into an efficient executable for the PVM environment. CODE's abstract declarative structure represents atomic actions as nodes and dependencies among actions as arcs. This Graphical User Interface representation is internally represented as an abstract syntax tree. The PVM translator works by traversing this tree while producing source code.

CODE 2.0's PVM translator produces source code that runs on a set of heterogeneous hosts networked using PVM. One PVM task executes on each host. These tasks communicate using asynchronous communication primitives.

Distributed computation usually involves some overheads due to the use of messages. In this implementation, these overheads are minimized by using messages only when absolutely required. Each of the basic elements in a CODE graph is mapped to a unique PVM task. This mapping is available on every PVM task. Messages are used only when data transfer occurs between elements residing on different PVM tasks. In cases of local transfer, the local data structures are manipulated to produce the same effect.

However, it is possible that the hosts on which CODE's PVM tasks are spawned are themselves heavily loaded. In such cases, no attempt is made at dynamic load balancing between tasks. That will be tackled by a separate load-balancer which works with PVM.

The translator produces a C program which uses PVM primitives which must be compiled on the target architecture to be run.

Chapter 3 outlines the design of the distributed execution environment for CODE.

Chapter 4 describes the implementation of the distributed execution environment of CODE for PVM.

1.4 Experimental Results

The output of CODE's PVM translator compares favorably with hand-coded programs in terms of speeds of execution. Chapter 5 details the results of an extensive set of experiments.

1.5 Related Work

There have been many attempts at providing similar high-level abstractions for parallel programming. PVM itself is an attempt at presenting a homogeneous message-passing interface to a heterogeneous network of computers. HeNCE[Beg91] is a graphical parallel programming tool which can be used to create parallel programs on a set of networked machines. Linda[Sci92] is another parallel programming language which tries to augment sequential programming through the use of primitives which are architecture-independent.

Chapter 6 explains in greater detail some of the related work in this area.

Chapter 7 presents the conclusions and indicates possible future work.

Chapter 2. Overview of CODE and PVM

This chapter will provide an overview of the CODE programming environment and of PVM.

2.1 CODE

The following is an extract from the CODE 2.0 Users Manual[-New93b].

CODE is a graphical parallel programming language. The fundamental idea is that users can create parallel programs with it by drawing and annotating graphs. A graph is a type of diagram that consists of nodes (represented by icons in CODE) and arcs that interconnect the nodes. Such a diagram shows a computation's communication structure. Nodes represent computations (or shared variables) and arcs represent the flow of data from one computation to another. Parallel programs are created by composing nodes into graphs by direct construction. Users draw the nodes and then interconnect them with arcs.

There are several steps in creating a program with CODE. First, the user draws the graph that shows the parallel program's communication structure. Then he or she annotates this graph by filling in a set of forms that describe properties of the nodes, arcs, etc. in the graph. For example, the user must specify which sequential computation a node performs. Finally, the user selects a target machine from a list of machine types for which translators have been prepared, and CODE translates the annotated graph into a program that can be compiled and run on the chosen target machine.

2.1.1 A Simple Example Program

When a user first runs CODE, an empty window appears. He or She then draws a graph in this window using a mouse. CODE graphs can contain several different node types which have different meanings and uses. Each different node type is represented by a different icon. The most important type of node is the Unit of Computation (UC) node. These represent sequential computations that can be composed into parallel programs by drawing arcs that show data flow among them. Figure 2-1 shows a complete CODE program that has been created using only UC nodes and data flow arcs.

This program integrates a function over an interval by dividing the interval in half and integrating over each piece separately. The results are summed to form the final answer. The program consists of four UC nodes. The arcs represent data that is created by one UC and is needed before another UC can begin its sequential computation. When the program runs, it first executes (or fires) the UC called “Split Interval” which divides the interval in half and sends the endpoints of the sub-intervals to both of the “Integ Half” nodes.

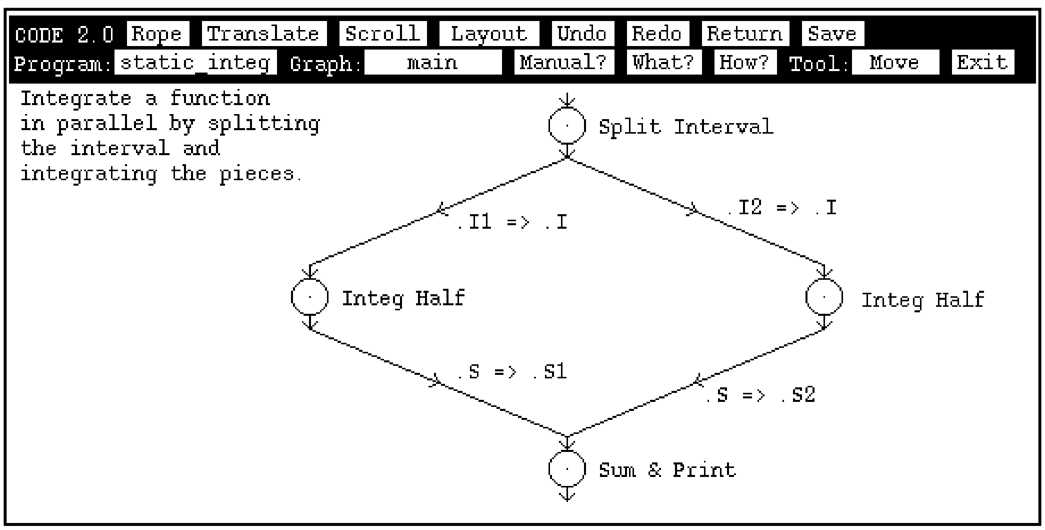

Figure 2-1. A CODE Program.

These nodes integrate, using a sequential procedure, a fixed function over the interval sent to them. They can run in parallel with each other since there is no arc from one to the other. Node “Sum & Print” waits for results from the two integrations and then adds them together and prints this sum. The arcs represent unbounded FIFO queues of data that are sent from one UC node to another. Nodes send and receive data on ports that are defined within them. Arcs bind ports together. These bindings are shown next to the arcs in Figure 2-

1. For example, node “Split Interval” sends data out on port “I1” that is received by the “Integ Half” node on the left on port “I.”

Drawing the graph is only the first part of creating a CODE program. Users also have to annotate it to complete its specification. These annotations define many aspects of the program-- what sequential computation a UC node will perform, under what conditions it can fire, what data types are defined for the program, etc. Annotations are performed by filling out attribute forms associated with the entity being defined. Figure 2-2 shows the attribute form associated with a UC node.

Notice the fields for “Terminate node?” and “Start node?”. The programmer must designate exactly one UC node in the program to be the Start Node and exactly one node to be the Terminate node. When a CODE program is run, the system first executes the Start Node to get the computation started. The firing of the Terminate Node signals the end of the computation to the CODE system. The system will not end the computation until the Terminate Node fires.

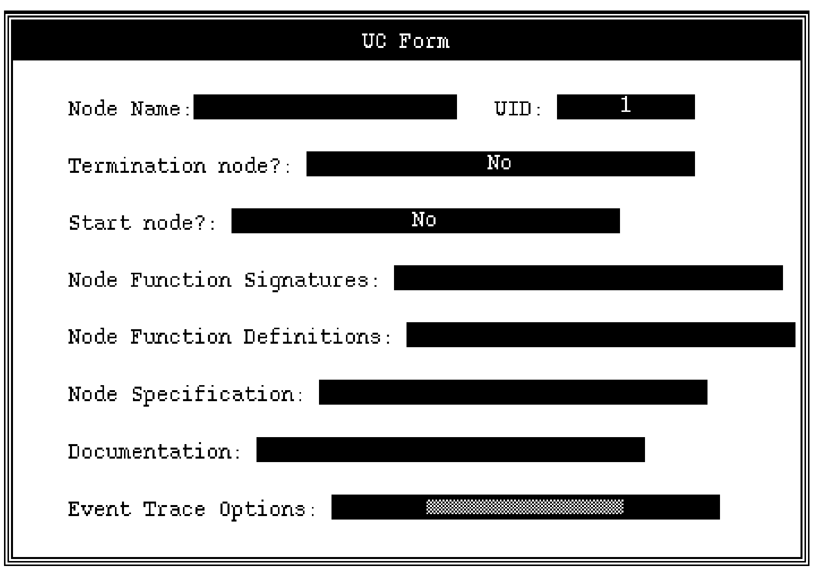

Figure 2-2. UC Node Attribute Form.

Recall that a UC node represents a sequential computation. These computations are specified by the user in the “Node Specification” field of the UC attribute form. This specification also includes the mapping between the data on the incoming and outgoing arcs and the local data used by the UC’s computation, and it also defines the conditions under which the UC node is allowed to fire.

The node specification is text-based and mostly declarative. It is divided into units called “stanzas” each of which has a different function. Stanzas consist of a name followed by some text enclosed in curly braces { }. The specification for the UC node named “Split Interval” is shown below.

```
// ***** Node Split Interval ***********
output_ports { IntegInfo I1; IntegInfo I2; }
vars { real a; real b; int n; IntegInfo i1; IntegInfo i2; }
comp {
   a = ReadReal();
   b = ReadReal();
   n = ReadInt();
   i1.a = a;
   i1.b = (b - a)/2.0;
   i1.n = n/2;
   i2.a = i1.b;
   i2.b = b;
   i2.n = n - i1.n;
}
routing_rules {
   TRUE => I1 <- i1; I2 <- i2;
}
```

The first stanza is the “output_ports” stanza. Output ports are names that are bound to arcs that leave the node, and input ports are names that are bound to arcs that enter nodes. This binding is specified as an arc attribute and will be discussed later. For now, it is enough to realize that two arcs that it refers to by names I1 and I2 leave Split Interval. The type name of the data that will be placed on these arcs is IntegInfo, a structure that contains the endpoints of an interval and the number of points to use in the integration.

The “vars” stanza of a node defines local variables that are defined inside the node and are used by its sequential computation.

The “comp” stanza defines the sequential computation the node performs when it fires. Split Interval defines two intervals which it stores in structure variables I1 and I2.

The language used in the comp stanza resembles a subset of C, but is not C. It is in fact, rather limited. It is expected that all substantial sequential computations will be encapsulated in ordinary sequential functions (usually written in C) that are defined outside of CODE (although CODE has facilities to help the user in managing, placing, and compiling them). Notice the calls to ReadInt and ReadReal. These are C functions, written separately.

UC nodes are the fundamental building blocks in CODE. UC nodes run in parallel with UC nodes. Hence, CODE programs express parallelism at a coarse level of granularity, roughly at the level of a subroutine call, since UC nodes are expected to perform fairly significant computations, often involving calls to external C functions.

Since Split Interval is the Start Node, its computation will be run immediately when the program is executed. After its sequential computation is complete, its routing rules are evaluated. Routing rules determine what values will be placed on what outgoing arcs and have the general form shown below.

```
Guard, Guard, ..., Guard => Binding; Binding; ...; Binding;
```

Guards are Boolean expressions and Bindings are “Arc Outputs” that use a “<-” operator to place a value onto an arc. Other forms of bindings are allowed and will be discussed elsewhere. All bindings are performed from left to right on all routing rules whose guards all evaluate to TRUE. Node Split Interval places struct i1 onto arc I1 and struct i2 onto arc I2. These define the intervals that the Integ Half nodes will integrate.

Both of the Integ Half nodes have identical specifications, hence only one of them must be discussed, and the only new stanza is the Firing Rules Stanza.

```
// ******* Both Node Integ Half are identical *********
input_ports { IntegInfo I; }
output_ports { real S; }
vars { IntegInfo i; real val; }
firing_rules {
   I -> i =>
}
comp {
   val = simp(i.a, i.b, i.n);  // Integrate using Simpson's
                               // rule.
}
routing_rules {
   TRUE => S <- val;
}
```

Firing rules serve two purposes. They define the conditions under which a UC node is allowed to fire, and they describe what arcs have values removed from them and placed into local variables for use by the node's sequential computation. Firing rules have the form shown below.

```
Guard, Guard, ..., Guard =>
```

Guards represent "Arc Inputs" that extract information from arcs using the "->" operator.

Actually, firing rules can be somewhat more complicated than this. Assignment statements can appear on the right of the "=>" and guards can also be Boolean expressions. Firing rules in the general case have a very rich syntax and can express quite elaborate firing conditions and bindings. This will be discussed elsewhere since such elaborate firing rules are needed only occasionally.

Arc inputs represent both a condition and a binding. The notation "I -> i" represents the condition or guard "there is a value on arc I" and the binding "remove a value from arc I and place it into variable i." A UC node may execute whenever all of the guards on any of its firing rules are TRUE. Such a rule is said to be “satisfied”. When the node fires, the bindings associated with the satisfied rule are performed. Hence, Integ Half may fire whenever there is a value on arc I. It extracts a value from I and places it into local variable i for use by the node’s computation.

It is possible for a node to have many firing rules. As indicated above, the node may fire when any of them are satisfied. If more than one rule is satisfied, the CODE system chooses one of the satisfied rules arbitrarily and performs its bindings before firing the node.

So, the nodes Integ Half wait for a value on arc I, perform an integration described by the interval received, and then pass the result out on arc S.

Node Sum & Print has a firing rule that requires it to wait for values on both arcs S1 and S2 before it can fire. When it fires, it prints the sum of the values received and, since it is the Terminate Node, signals the end of the computation.

```
// ******* Node Sum & Print **********
input_ports { real S1; real S2; }
vars { real s1; real s2; }
firing_rules {
   S1 -> s1, S2 -> s2 =>
}
comp {
   PrintReal(s1+s2);
}
```

The last issue in the Integration program is the annotation of arcs. Each arc’s annotation is shown in Figure 2-1. (Such display is not automatic. The annotations have been written as comments on the graph.)

Arc annotations are called “arc topology specifications”. They serve to bind names between nodes. As mentioned above, nodes use ports as local names for arcs. Arc topology specifications bind port names together. For example, the specification

```
.I1 => .I
```

indicates that output port I1 (declared in UC node Split Interval) is to be bound to input port I (declared in node Integ Half on the left). When you draw an arc between UC nodes, you must specify what pair of ports the arc binds.

It is reasonable to think of UC nodes as being analogous to integrated circuits. The port names of the UC serve the same purpose as the pin names on the IC. You place UCs into a graph in any way you like and connect them with arcs rather than wires. The arc topology specification describes what “pins” have been connected.

2.1.2 CODE Elements Are Actually Templates

It is time to confess that the description of CODE so far, although completely correct, has been oversimplified. The notations as they have been described are inadequate as the basis for a programming system because they are static. As described so far, the graph drawn represents a completely fixed computation and communication structure. Many real-world algorithms do not fit into this view. Aspects of their structure depend on runtime information. As a simple case, one may wish to prepare a program that will utilize as many processors as happen to be available at a particular moment to perform a certain task. Perhaps the desired structure is as shown in Figure 2-3 where there are N replicated nodes in the center, where N is a runtime determined value. Structures that are dictated by runtime parameters are called “dynamic”.

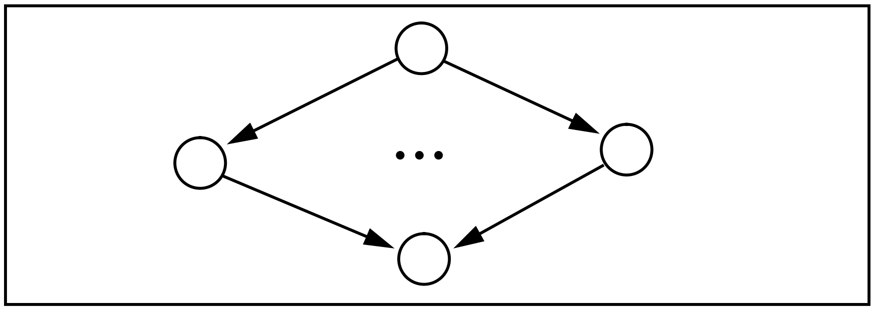

Figure 2-3. A Dynamic Computation Structure.

CODE directly supports the specification of dynamic structures. Rather than being static, every node or arc users draw in CODE can be instantiated any number of times at runtime. The instantiations are named by integer valued indices. Hence, it is more accurate to say that you are creating a *template* for a node rather than a node itself when you draw and annotate a UC node.

The simplest way to use the template is to use a single instantiation of it that has no indices. This is what was done in the Integration program.

The Integration program is limited to two-way parallelism. There are only two Integ Half nodes to run in parallel. An N-way parallel program can easily be envisioned that would use N "Integ" nodes each of which integrates a fraction of the interval of size (b - a)/N. A CODE program that implements this scheme is shown in Figure 2-4. The node Integ is instantiated N times using the same communication structure shown in Figure 2-3.

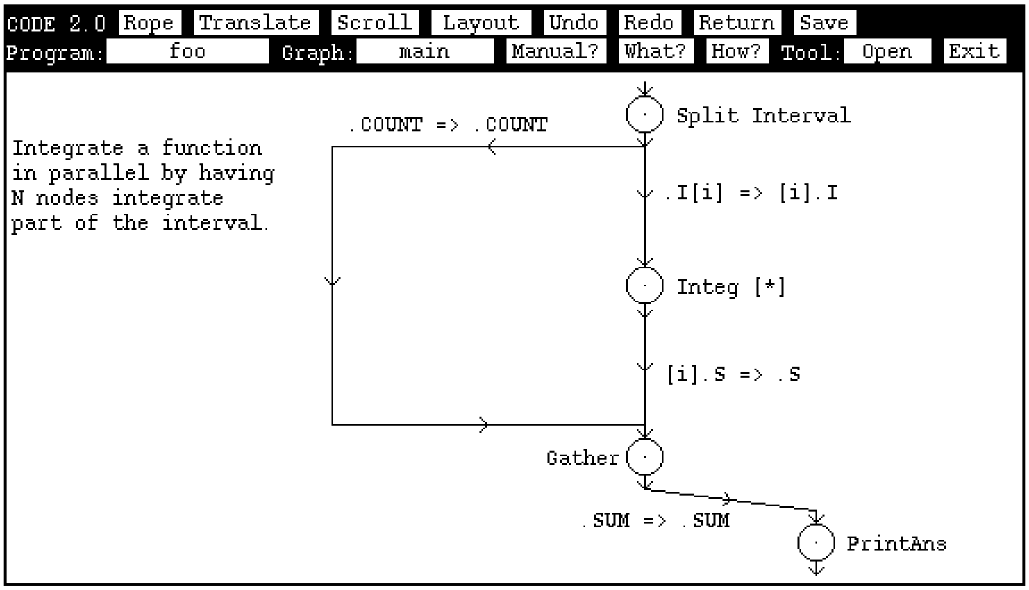

Figure 2-4. Another Integration Program.

The “[*]” by the name is a comment reminding the viewer that the node is instantiated multiple times. In fact, UC node names serve only as comments and, hence, are optional attributes. Actually, all of the text that appears in Figure 2-4 is either an optional name or a comment drawn on the graph. Such text increases the readability of the graph.

The multiple instantiation is specified by annotations that will be described in later chapters. The important idea to remember now is that all CODE elements (nodes, arcs, etc.) you draw are in reality templates that may be instantiated many times. The graph depicts static aspects of the program. Annotations capture dynamic aspects.

2.1.3 Other CODE Nodes

There are other node types than UC nodes in CODE. Many of these serve to hierarchically structure programs, exactly as subprograms and Call statements do in conventional languages such as C or Fortran. Figure 2-5 shows all of CODE’s node types.

CODE graphs play the role of subprograms. In general, CODE programs consist of many graphs that interact by means of Call nodes. The Integration program is simply a single graph program.

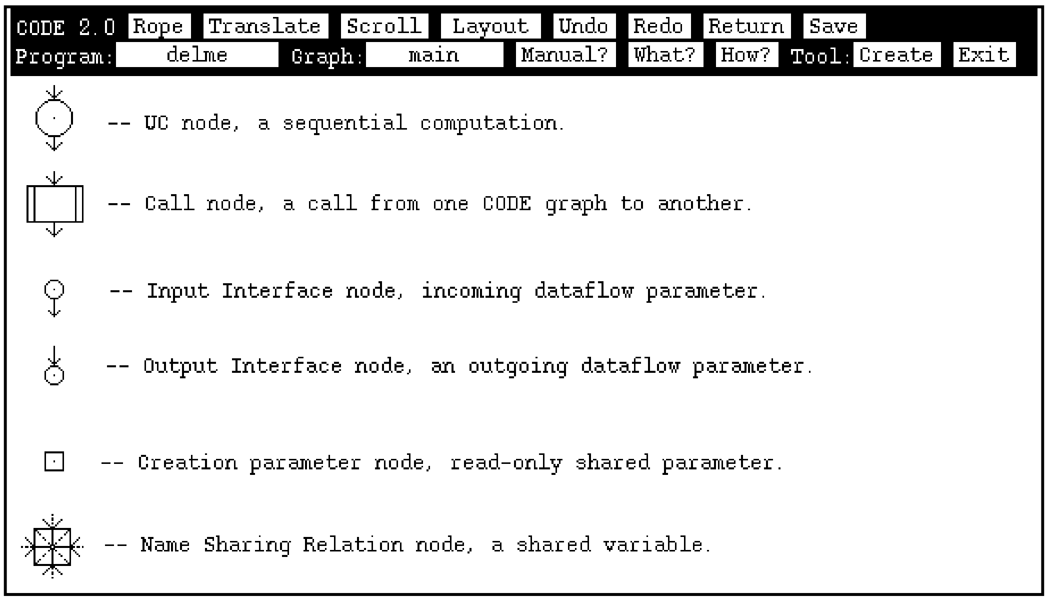

Figure 2-5. All CODE Node Types.

Just as with subprograms in conventional languages, CODE graphs have formal parameters. They are specified by Interface nodes and Creation Parameter nodes. A graph's Interface and Creation Parameter nodes form its interface. Actual parameters are arcs that enter or leave Call nodes. These arcs are bound (by means of arc topology specifications) to Interface or Creation Parameter nodes within the graph that is called.

Call Nodes and Hierarchical Structuring

No useful programming language can be without facilities for hierarchically structuring programs. Conventional programming languages such as C and Fortran use Call statements and procedures for this purpose. A program is made up of a number of procedures that interact via Call statement. CODE has facilities that are entirely analogous. A CODE program is made up of a set of graph instances that interact via Call nodes. Graphs are like procedures in that they are the basic unit of hierarchical structuring. Call nodes are like Call statements.

Creating a Graph

When CODE is first run and given the name of a new graph (.grf) file, it creates the file and a graph called "main" within it. It is intended (but not required) that your Start node be in this graph. You can create another graph which can be called from "main" or any other graph that you create. Graphs can even be recursive. To create a graph, click the create cursor on the Graph button and enter a name for the new graph. The name must be a legal CODE identifier.

Let us consider the elements of a CODE graph. Just as Fortran procedures have formal parameter lists to define their interface, CODE graphs have Interface and Creation Parameters nodes to define theirs. This is perhaps best explained by an example. Suppose we wish to create a graph that could be called multiple times to multiply a series of vectors by a fixed matrix to produce a vector result. Figure 2-6 shows such a graph.

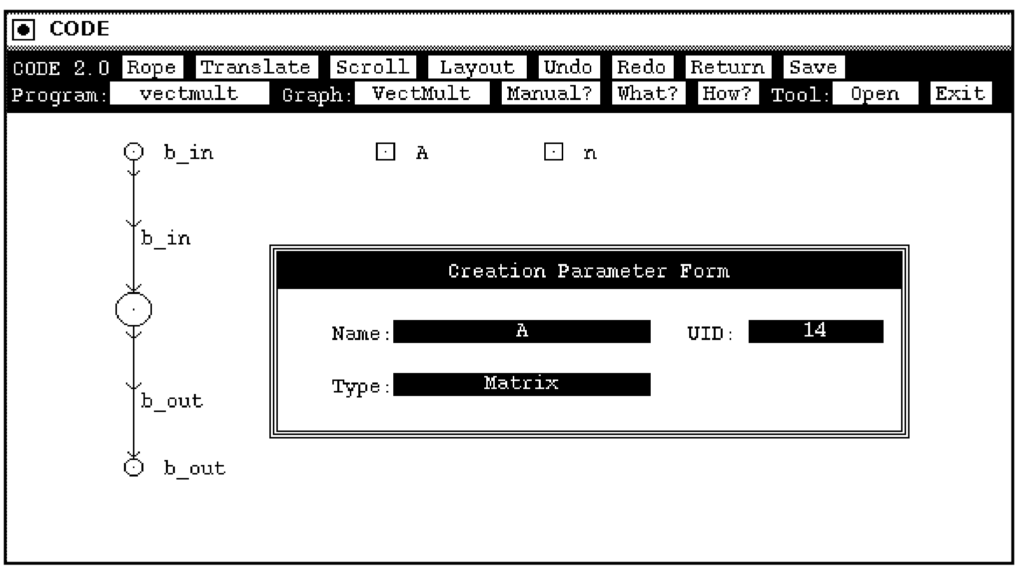

Figure 2-6. VectMult Graph.

This graph has three input formal parameters and a single output formal parameter. All formal parameters to graphs are either interface nodes or creation parameters. These nodes must be named (with unique names that are legal CODE identifiers) since their names form the graph's interface.

Parameter	Type	Node Type
b_in	Vector	Input Interface node
A	Matrix	Creation Parameter node
n	int	Creation Parameter node
b_out	Vector	Output Interface node

Let us first discuss the Interface nodes. They have only two attributes, a name and a type name. These nodes exist because arcs in the calling graph must be bound to ports or shared variables in the called graph, but one would not want such things to form a graph's interface because they are too "local"-- they are defined within nodes. Interface nodes serve as aliases for ports in UC nodes or shared variables in Name Sharing Relation nodes.

For example, the UC node in our example graph has an input port called "b_in," and we wish the calling graph to send data to it. So, we create an interface node and give it a name. The example uses name "b_in." It is the same as the port name in this case, but need not be. Next, an arc is drawn from the Input Interface node to the UC. Its attribute form contains a "Port to Connect to" field. It's value should be the name of the port the Interface node aliases, "b_in" in this example. All of these annotations are shown in figure 2-6.

The Output Interface node works in the same way. It aliases a port in the UC that happens to be called "b_out."

Creation Parameter nodes represent a special kind of shared variable. They receive exactly one value from the calling graph-- at the moment that the graph is created. This variable can then be read from anywhere within the graph containing the Creation Parameter node.

Our example has a Creation Parameter called “A.” Its attribute form is shown in figure 8.1. It has bound to it the matrix that will be used in all of the vector-matrix multiplications. Variable “A” is of type Matrix and can be read from any point in the graph (including the UC node that does the multiplication) just as though it were a local variable. It is illegal to write to “A” however. Also, no arcs may be drawn to or from Creation Parameter nodes. There is no need in any case.

Name Sharing Relations

Name Sharing Relation nodes are a mechanism for declaring variables that will be shared among a set of UC nodes. Theses nodes contain definitions and initialization of shared variables but do not in themselves specify which UC nodes will access which variables and in which way. That is done by shared variable declarations in UC nodes and by arcs that bind UC shared variables to shared variables in a Name Sharing Relation node. UC nodes that access them must declare themselves to be either readers or writers of the variable

Once a shared variable has been declared in a UC, it must be bound to a shared variable in a Name Sharing Relation node by means of an arc with an arc topology specification. The specification is much like that for a data flow arc, but no port indices are permitted.

One should notice that a UC shared variable is much like a port. It is a name that is to be bound to something outside of the UC node itself. The arc topology specification provides this binding.

An example may be useful. Figure 2-7shows the graph for a simple program. In it, nodes A and B have firing rules and data flow arcs that cause them to fire repeatedly. Each time they fire, they increment a shared integer.

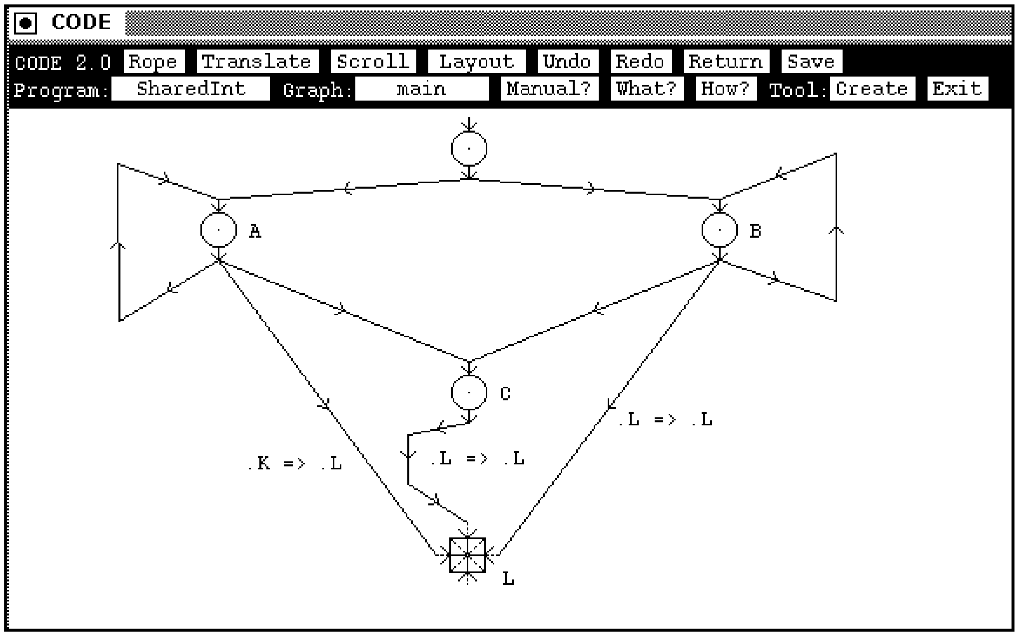

Figure 2-7. Name Sharing Relation Example.

After A and B fire the specified number of times, they send dummy values to C which cause it to fire. It prints the value of the shared integer. A and B are writers of the integer. C is a reader. We will ignore all aspects of this program but the use of the shared variable.

Node A's specification is shown below. Notice that it uses name "K" for the shared variable and declares itself to be a writer of it.

```
input_ports { int X; }
output_ports { int Y; int AGAIN; }
shared_vars {
 int K writer;
}
vars { int v; }
firing_rules {
 X -> v => v = v - 1;
}
comp {
 K = K + 1; // Increment shared int
}
routing_rules {
```

```
 v == 0 => Y <- v; &&
 v > 0 => AGAIN <- v;
}
```

Node B is the same except that it uses name L.

The specification of the Name Sharing Relation node follows.

```
shared_vars {
      int L;
}
init_comp {
      L = 0;
}
```

Node A's shared variable name K is bound to shared variable L by means of the arc drawn from UC node A to the Name Sharing Relation node. The arc topology specification is as shown below.

```
.K => .L
```

One always draws arcs to Name Sharing Relation nodes, even if the arc must "cross" Call boundaries.

There are some limitations on the use of shared variables within a UC node. They may be used only within the UC's comp and routing_rules stanzas, and it is of course illegal to write to a shared variable that has been declared read-only.

In addition no two shared variables within one UC node may be bound to the same shared variable in a Name Sharing Relation node-- even when different instances (meaning nodes with indices) of the Name Sharing Node are involved.

2.1.4 Translating and Running CODE Programs

Once a CODE program has been drawn and annotated, the CODE system can translate it into a program that can be compiled and run on a parallel machine. This process will be described in a later chapter but the general idea is that the user clicks on the translate button at the top of the CODE window and picks a target machine type, such as the Sequent Symmetry or a cluster architecture like PVM. CODE will produce a program in C that realizes the computation and communications structures expressed by the graph. The user may then compile and run this program on the parallel machine.

Users are offered choices on how CODE will optimize and instrument the program. For example, one can ask CODE to automatically have the program be instrumented to measure how long each UC takes to fire and how often it fires.

That concludes the extract from the CODE 2.0 User manual.

2.2 PVM

This section will provide a brief overview of the Parallel Virtual Machine (PVM) as relevant to this thesis. It contains material taken from the PVM 3 User's Guide and Reference Manual[Gei94].

PVM 3 is a software system that permits a network of heterogeneous UNIX computers to be used as a single large parallel computer. In other words, a user defined collection of serial, parallel and vector computers appear as one large distributed-memory computer. This logical distributed-memory computer is termed a virtual machine. The list of computers supported include SUN Sparcstations, IBM /RS6000 among a large number of other architectures.

PVM supplies primitives to start up tasks on the virtual machine and allows the tasks to communicate and synchronize with each other. A task is defined as a unit of computation in PVM analogous to a UNIX process. By sending and receiving messages multiple tasks can co-operate to solve a problem in parallel. PVM handles all data conversion that may be required if two computers use different floating points representations. Also, PVM allows the virtual machine to be interconnected by a variety of different networks.

Appendix A describes PVM primitives which are used in the implementation of this thesis.

Chapter 3. Design

This chapter will provide an overview of the distributed execution environment along with a discussion of the key issues involved in its design.

3.1 Overview

UC's are the actual computational elements of CODE. The dynamic parallel execution environment of CODE is due to individual UC instances executing concurrently. Thereby, in the execution environment, UC instances are distributed across the PVM cluster architecture. The granularity of UC's does not warrant that one process be spawned per UC instance. Instead, more than one UC instance is executed by a single process.

Since processes execute serially on a uniprocessor, in general there is no gain in running more than one process on each uniprocessor. Thus, only one process runs on each machine in the PVM configuration. These processes will be referred to as PVM tasks. Each PVM task runs its own scheduler which maintains a queue of UC's waiting to execute. The different PVM tasks run asynchronously. Each PVM task continually polls its queue and keeps executing UC's until program termination. All interactions (data flow) in the distributed environment have two modes - local and non-local. The distributed environment exploits its knowledge in cases where the interaction is local thereby minimizing the number of messages generated.

3.2 Distribution of Work

CODE supports a dynamic execution environment in which there is lazy creation of instances of CODE elements at runtime. To distribute work, a mapping routine is used to dynamically assign these instances to PVM tasks at the time of their creation. The mapping depends on the number of PVM tasks and the number of instances created during a particular run. A UC instance created by the execution of a PVM task may be assigned to a different PVM task, even a PVM task on a different host system.

3.3 Address Tables

The CODE model of programming envisions the user drawing/writing a CODE program which is translated into a set of templates, one for each CODE computational element used in the program. This set of templates is used as the framework for creating instances of computational elements in the dynamic runtime environment.

At compile time, it is not known how many instances of a template will actually be created. While it is possible, in theory to statically assign unique IDs to all instances of a template, the list of IDs would be infinite. Instead, a mapping function is used to encode this mapping. An address table which stores addresses of all runtime instances is used for name resolution. In a distributed environment, the options are to have a central address table which contains the addresses of all instances present across the entire runtime environment or to maintain a distributed address table. The former would involve centralized address lookups which would lead to the address table being a bottleneck for the entire system. Hence, a distributed address table will be used. This means that each PVM task will only have the addresses of all the instances of CODE elements that have been created on itself.

3.4 Name Resolution

Integration of all the distributed address tables would provide a complete tree representing the chain of creation of instances. Such a tree will be called a dynamic instance tree. In a dynamic instance tree, instances higher up in the hierarchy invoke ones further down. The tree also represents the various levels of scoping in the runtime environment. A complete tree is not available at runtime to the PVM tasks since the address table is distributed across all PVM tasks.

Name resolution in CODE's distributed environments can be explained by means of the following CODE program which consists of two graphs. Consider Figures 3-1 and 3-2. Suppose at runtime there are two instances G[1] and G[2] of the graph G. This implies that there are two instances of X - G[1]/X and G[2]/X as well as two instances of Y-G[1]/Y and G[2]/Y. Now, suppose that G[1]/X and G[2]/X are mapped to PVM task M and G[1]/Y, G[2]/Y are mapped to PVM task N. Once instance G[1]/X completes execution, it sends data to G[1]/Y. Now, two instances of Y - G[1]/Y and G[2]/Y reside on PVM task N. Therefore, in order to uniquely identify G[1]/Y, G[1]/X must the send the entire string “G[1]/Y” to task N along with the data.

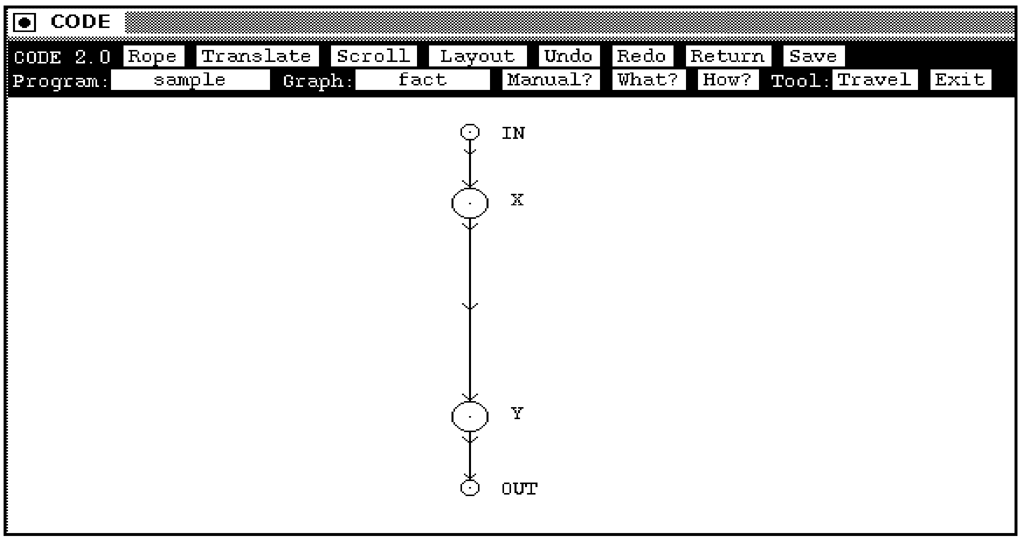

Figure 3-2. Graph Fact of Sample CODE Program.

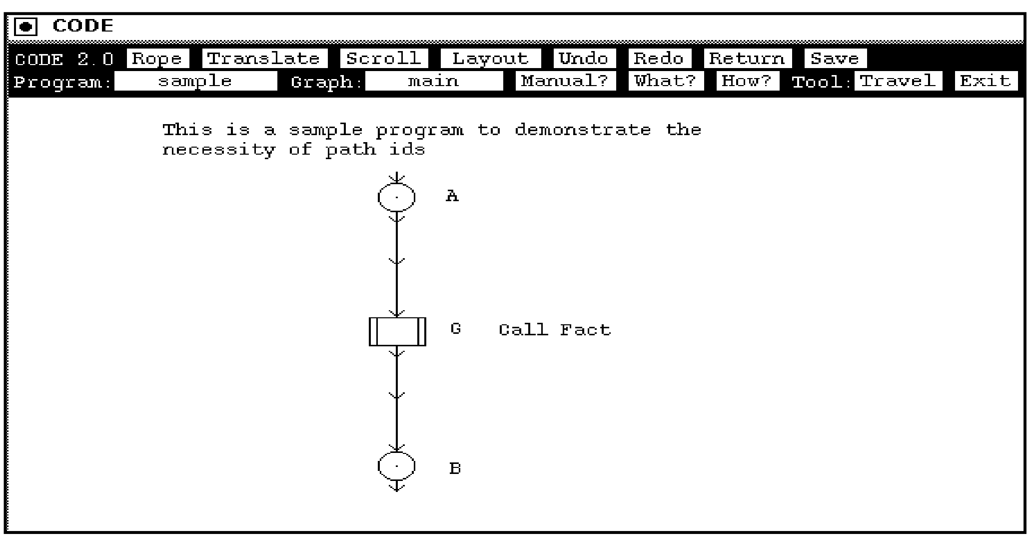

Figure 3-1. Main Graph of Sample CODE Program.

Extending this to more levels, we must specify all the parent graphs of a instance in order to uniquely identify it. The value of this string is the full path of the instance from the root in the dynamic instance tree. This string will be referred to as "Path ID" or more simply "Path" and will be used to uniquely identify instances in CODE's distributed environment.

3.5 Creation Parameter Nodes

Creation Parameter nodes are akin to global constants within the scope of a CODE graph. These nodes are initialized once at the time of creation of the graph and remain constant after that. CODE graphs are instantiated on all PVM tasks where at least one of their children (UC, NSRel or graph) needs to be instantiated. The distributed execution environment makes no effort to keep track of all PVM tasks on which a particular CODE graph has been instantiated. Instead, whenever a Creation Parameter node has to be instantiated, the graph to which it belongs is instantiated on all PVM tasks and is initialized on all of them.

3.6 Name Sharing Relations

The issues involved in the implementation of Name Sharing Relations are as follows.

3.6.1 Distribution of Name Sharing Relations

While Name Sharing Relations are not actual computational units, they are a resource shared by many UC's. Accessing shared variables in NSRel's involves some overhead. Therefore, Name Sharing Relations are also distributed across PVM tasks using the same mapping routine used for other CODE elements. Again, the NSRel to PVM task mapping is available to all PVM tasks locally.

3.6.2 Distributed Access of Shared Variables

Shared variables in NSRels are accessed by various UC instances. In order to ensure the integrity of each access, some form of locking mechanism must be set in place. The locking mechanism must work irrespective of whether the NSRel involved is local or remote.

3.6.3 Deadlock Avoidance

The other issue is the fundamental problem of deadlock avoidance. This is solved in the following manner. Each UC has a list of shared variables that it must lock, before it starts computation. Locks in the list are always acquired in a particular order. This order is based on a unique id which is used by each shared variable. Since all UC's acquire locks in the same order, there is no possibility of a deadlock developing due to a circular wait for locks.

3.6.4 Locking Mechanism for Shared Variables

The locking mechanism followed for shared variables is best explained by means of an example. Consider the CODE graph in Figure 3-3.

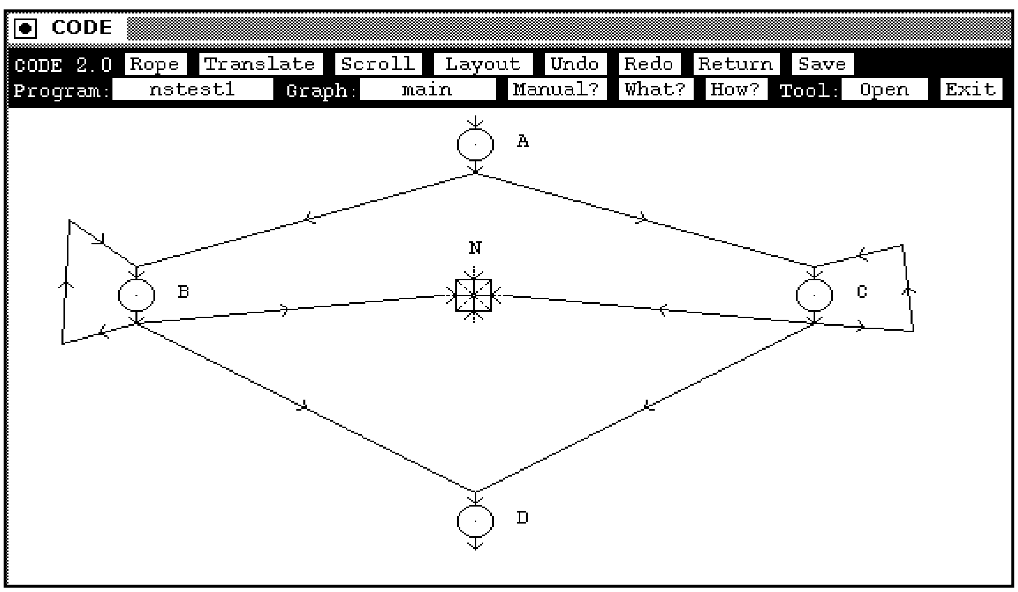

Figure 3-3. A simple CODE graph.

Suppose the runtime mapping of UC's and NSRel's for the CODE graph in Figure 3-3 is as shown in Figure 3-4.

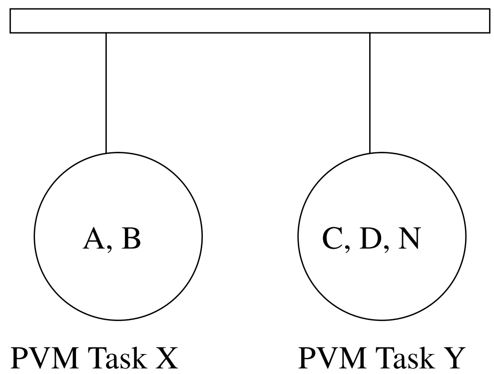

Figure 3-4. Runtime Mapping of CODE elements of Fig 3-3.

Further suppose that there is a shared variable I within NSRel N and that both UC's B and C have shared variables which are writers of I.

Consider the case when UC B accesses shared variable I in NSRel N. Before B fires, it must acquire a lock on shared variable I. It invokes the mapping routine and discovers that I is on a different PVM task, namely Y. Thereby, B sends a PVM message to PVM task Y requesting a lock on shared variable I.

Now, PVM task Y checks to see if shared variable I is already locked by some UC or not. If not, it locks I and sends a PVM message to task X which indicates that UC B has been allowed write access to shared variable I. It also sends the current state(value) of the shared variable I. UC B upon receiving this message updates its local version of the shared variable I. Then it proceeds with its computation. At the end of the computation, UC B sends a PVM message to Task Y requesting the release of the lock on shared variable I along with the updated state of shared variable I. Task Y receives this message, updates the shared variable I and then releases its lock.

However, if I has already been locked by some other UC, then Task Y places UC B on a queue of UC's which are waiting to get a lock on shared variable I. When, the UC which currently has a lock on I releases it, all the UC's in the queue are executed again so that they may try to get a lock on I. Now, some of the UC's in the queue might be from a different PVM task. In that case, a PVM message is sent to those PVM tasks which ensures that the respective UC's are executed again.

Now, consider the case when UC C accesses shared variable I on NSRel N. It must acquire an access lock on I before its starts computation. It invokes its mapping routine and discovers that I is on the same PVM task as it is. It then checks the structure representing I to see if it has already been locked. If so, it gets on the queue of tasks waiting for access to I. Otherwise, it sets a flag in I's structure indicating that it is locked for write access and proceeds with its computation. After completion it unlocks I and enqueues the list of UC's waiting for access to I for execution.

3.7 Data flow and Messaging

All runtime interactions in CODE involve some form of data flow. Local interactions (communication between CODE elements on the same PVM task) take place through data structure manipulation. While this may involve access to shared structures, there is no need for the use of explicit synchronization primitives. This is so because, according to the CODE model of programming UC's execute atomically. Each PVM task has its own scheduler which runs only one UC at a time.

All non-local interactions in CODE's distributed environment involve some form of PVM messages. The availability of the mapping routine on every PVM task eliminates the need for "store and forward" type of messages. All messages in the runtime environment make only one hop.

Also, most interactions in CODE such as those between two UC's or between a UC and a CreP involve one-way communication. However, interactions with NSRels's involve two-way communication.

The various forms of PVM messages in CODE's distributed environment may be classified as follows.

3.7.1 Data transfer messages from UC to UC/Crep

These messages are used for data flow arcs in CODE. When a UC sends data to another UC which is on a different PVM task, this message is the means of data transfer. This message contains information that is needed to uniquely identify the destination UC and the data being transferred which can be an int, real, char, array or struct. In case of complex data types, a copy of the entire structure is transferred.

3.7.2 Data transfer on return from graphs

These messages are very similar to the previous data transfer messages in functionality. However, they are invoked on returns from CODE graphs.

3.7.3 Request access to non-local shared variable

This type of message is used to request a lock on a non-local shared variable. It contains information that would uniquely identify the sending UC and the shared variable involved. Also, it contains the type of lock - Reader or Writer.

3.7.4 Access approval for non-local shared variable

This message is sent by an NSRel when it locks one of its shared variables. This message contains the unique identity of the UC that requested a lock along with a snapshot of the current state (value) of the shared variable being locked.

3.7.5 Release access to non-local shared variable

This message is sent by a UC after it has completed its computation, when it wants to release all locks that it holds. This message contains the unique identity of the NSRel containing the shared variable being released and also a snapshot of the updated shared variable. This message is received by the NSRel which then updates its shared variable.

3.7.6 Request to enqueue UC instance for firing

This message is used to enqueue UC's which were queued up for access to shared variables for firing. When a UC sends a request for access to a shared variable, if the variable is already locked by some other UC, then this UC is put on a queue. When the shared variable is unlocked, then all UC's in the queue are enqueued for firing so that they can again try to get access to the shared variable.

3.7.7 Request to terminate run

This message is sent by the UC which runs the terminate node. Upon receiving this message all PVM tasks terminate execution and end their run.

That concludes the discussion of the design issues for the distributed execution environment.

Chapter 4. Implementation

This chapter describes in detail the implementation of the distributed execution environment for CODE.

The CODE model of programming has been designed to support multiple target architectures. When a user enters a CODE program, it is stored in Graphical User Interface data structures. Since, one of the objectives of CODE has been to decouple the GUI from the translators, an intermediate form of representation is used. This form of representation is called the Abstract Syntax Tree (AST).

Translation of CODE programs for a target architecture is a five-stage process consisting of translation from the GUI representation to the AST, AST decoration, AST optimization, Translation and Linking with the runtime library. The first two stages are architecture-independent and are not discussed here [New93a]. Also, no AST optimizations have been made as part of this thesis. The discussion here will pertain to translation and linking only.

Translating involves running the translation methods of all nodes in the AST for the PVM architecture. This produces a set of C programs which use PVM primitives. These programs must be compiled on the target architecture. Runtime library routines are discussed along with translation methods.

The last phase deals with the creation of an executable for the PVM architecture. It involves linking the files produced by the PVM translators with the CODE runtime library for PVM and finally the PVM library itself.

4.1 Abstract Syntax Tree

The Abstract Syntax Tree is an hierarchy of classes used to represent the user-defined parallel program in an architecture-independent manner.This representation is the starting point for the translators of CODE which will convert this abstract parallel structure into architecture-specific executable code. In order to facilitate architecture-specific optimizations, each of these classes contain a translate method for each target architecture of CODE. These methods emit architecture-specific code. Figure 4-1 shows the AST template.

At the root of the AST is the class Program. It represents the entire CODE program. The class Graph is among its children. A graph represents one complete screen of a CODE program. A CODE program consists of one or more graphs. Each CODE graph consists of Units of Computation(UC), Name Sharing Relations (NSRel), Arcs, Creation Parameter Nodes (CreP) among other CODE elements. All these are sub-classes of class Graph.

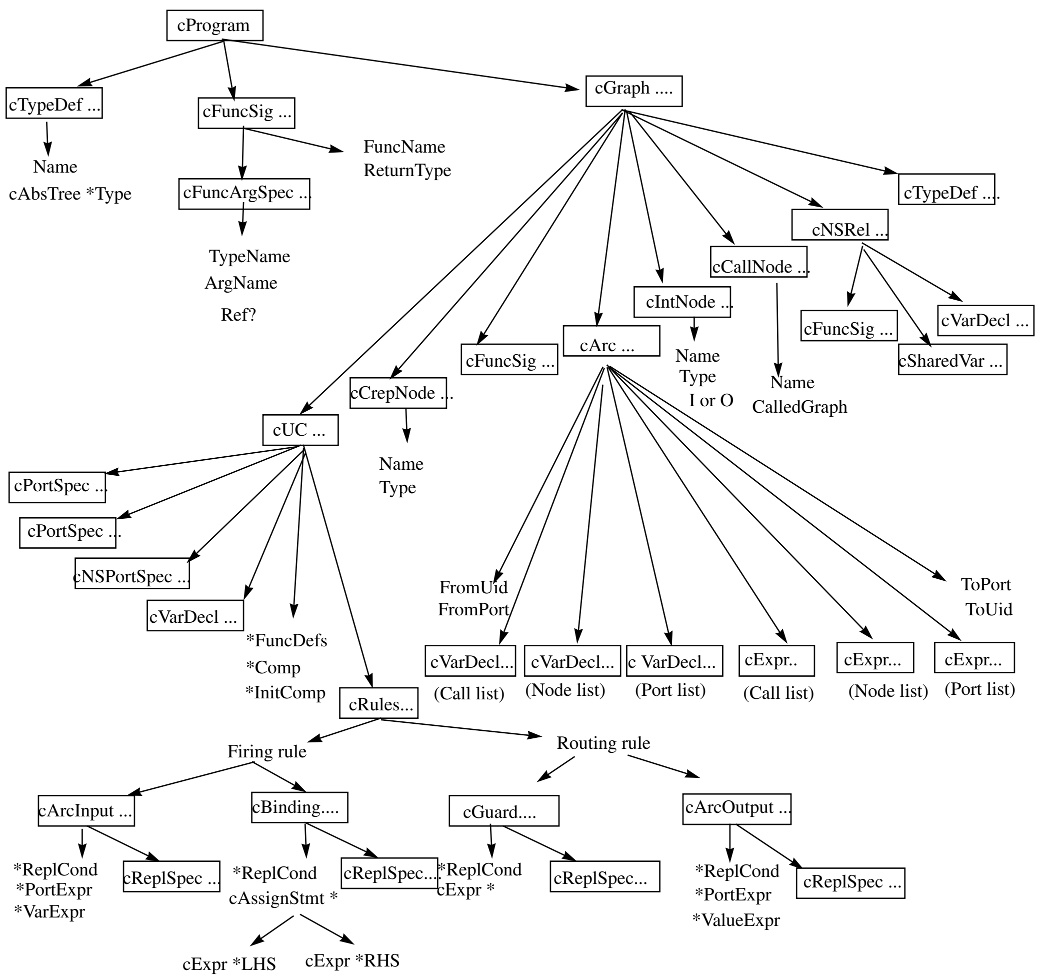

Figure 4-1. Abstract Syntax Tree.

When a CODE program is translated, the translate method for that architecture in the class Program is invoked which in turn invokes appropriate translate methods in its sub-classes in a depth-first manner. These translate methods ultimately generate calls to PVM and the CODE runtime library primitives.

The translate methods produce a set of C source files which along with a runtime library and the PVM library form the distributed execution environment of CODE.

4.2 Structure of Output

The output of the PVM translator is a set of C files. They are as follows

- A Makefile to allow the user to compile the parallel program that CODE produces using the UNIX "make" facility (Makefile).
- A file of translated global type definitions that users can include in separate files of sequential routines. (_c2_globtype.h)
- A C (and a ".h") file that implements the main driver routines for the parallel program, the message handling routines, as well as node and graph creation routines.(_c2_main.c & _c2_main.h)
- one C file for each graph (main.c etc.).

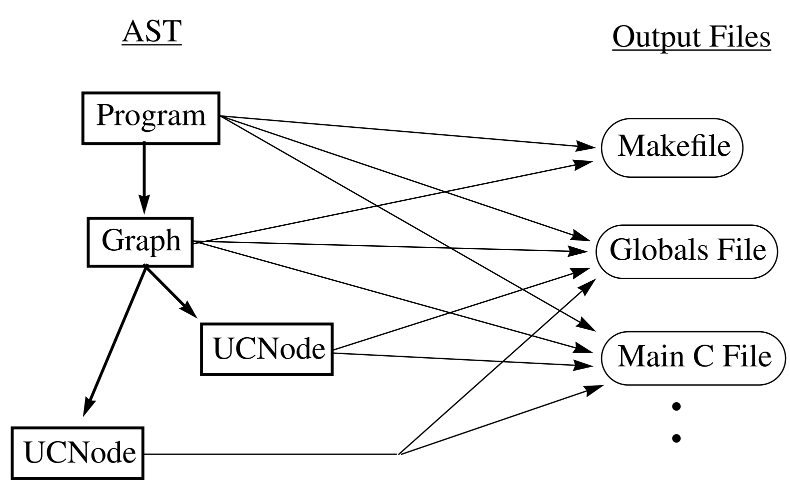

Figure 4-2. AST nodes place text into output files ([New93a]).

The PVM translation methods for the AST nodes write into these files as shown in Figure 4-2.

4.3 Structured Files

Structured files are an hierarchical abstraction used to represent the parallel PVM program produced by the PVM translator. As can be seen from Figure 4-3 the translate methods of various members of the AST write to the various output files. This introduces a coordination problem in which these translate methods must interleave code generation among themselves. This problem is explained in detail in Chapter 7 of [New93a]. Structured files are used to solve this problem. Structured files are also explained in detail in Chapter 7 of [New93a]. The following brief explanation of structured files is from that chapter.

A structured file defines in template form the structure of a set of UNIX files that will contain the code that make up the translated CODE 2.0 program. As a simple example, a program produced by CODE's translators might hypothetically consist of a set of global variable declarations which must be placed at the front of the file and a set of function definitions which must follow the variables. Each function consists of a header part and a body part. A structured file to represent this situation would have the form shown in Figure 4-3. Notice that the various parts of the file are named. The "[*]" indicates that the FuncPart is replicated. Each replication must be supplied with a unique identifier.

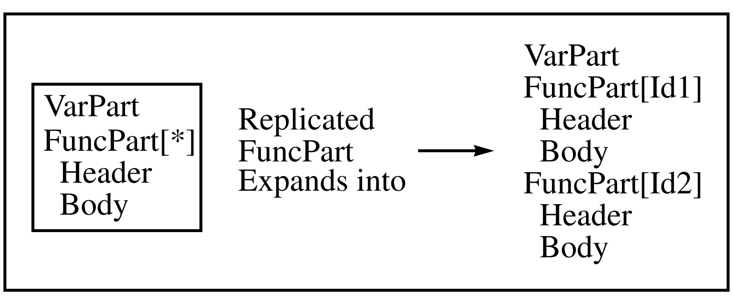

Figure 4-3. Example Structured File.

After all of the translation methods have written to the structured file, it is dumped to a plain UNIX text file to serve as input to a native compiler.

4.4 Runtime Representation of CODE elements

In this section, the runtime representation of the various CODE elements will be described in detail. The CODE elements UC, Graph and Name Sharing Relation are represented as two C structures allocated on the heap. These structures represent fixed type information and variable type information respectively. The fixed type information is hard-wired in the runtime library, while, the variable type information is generated at translate time.

.

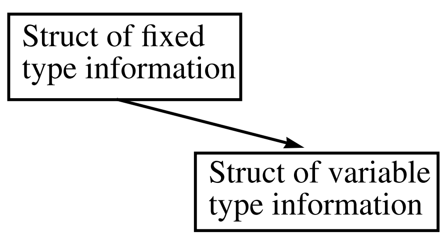

Figure 4-4. Runtime Representation of UC, Graph & NSRel.

4.4.1 Unit of Computation (UC)

Each UC has a fixed information structure which is represented as a C struct. This structure is defined as follows.

```
struct _c2_sNodeBase {
      int QueueStatus;
      int NSState;
      _c2_LockSet *HeadLock;   /* List of NS locks needed */
      _c2_LockSet *NextLock;   /* Next NS Lock needed */
      int UID;
      char Path[Path_ID_Length];/* The Path ID */
      int Crepped;             /* Creation params bound */
      _c2_GraphBase *MyGraph;  /* Containing graph */
      _c2_Index Index;

      void (*InitProc)();
      void (*CompProc)();
```

```
        void *LocalData;     /* Node specific information */
};
```

In the above structure, three fields provide varying degrees of identity to each UC. They are *UID*, *Index* and *Path*. Each *UID* uniquely identifies a UC template from all other UC's. However, the CODE runtime environment is dynamic and there maybe multiple instances of a UC template. In such a situation, the *UID* along with the *Index* provides for unique identity of a UC instance. UC instances may be created with up to seven dimensions. Still, the *UID* and the *Index* are unique only within the context of a CODE graph/subgraph. However, as explained in Chapter 3, there is a need in the PVM backend to have a unique identity for each UC instance across the entire runtime environment (for instance, to uniquely identify different UC instances of a UC template within multiple instances of the parent graph). Such a unique identity is provided by *Path* which is the Path ID as described earlier [Section 3.4].

Also, the structure contains a pointer to its parent graph. This among other things helps provide access to creation parameters. The flag *Crepped* indicates UC node dependency on creation parameters and is set to true when creation parameters are bound. The structure also contains two function pointers. One of them, *InitProc*, is executed the first time a UC instance executes. The other, *CompProc*, is run each time the UC instance is fired. The field *QueueStatus* is used in conjunction with CODE's runtime scheduler.

The last field in the structure, *LocalData* is a pointer to the variable type information present in each UC. This mostly contains user specified local variables of the UC and data flow queues for incoming ports.

There are also three fields of the structure which contain information pertaining to name sharing relations. They are *NSState*, *HeadLock* and *NextLock*. *NSState* reflects the current state of the UC with regard to acquiring name-sharing relation locks. *HeadLock* is a pointer to a list of name-sharing relation locks that a UC instance must acquire before it fires. *NextLock* is a pointer to the next name-sharing relation lock that the UC must acquire. Both of them point to structures of the following type.

```
struct _c2_sLockSet {
 int IsRemote;/*Shared Variable's NSRel local or remote?*/
 int NSRelUID; /* UID of Shared Variable's NSRel */
 char *NSRelGraphPath;/* PathID of Shared Variable's
                         NSRel's Parent Graph */
 _c2_Index NSRelIndex;/* Indices of Shared Variable's NSRel
                                                         */
 int RequestSent; /* Flag to indicate if a remote lock
                     request has already been made */

 _c2_NSRelBase *NSRelAddr;/* NSRel address */
 int *RCount;     /* number of current readers */
 _c2_NSLink **RQ; /* Queue of UC Nodes needing read locks
                     to shared variable */
 int *WCount;     /* number of current writers */
 _c2_NSLink **WQ; /* Queue of UC Nodes needing write locks
                     to shared variable *
 int ReqType;     /* Reader or Writer */
 void *SharedAddr;/* Address of Shared Variable in NSRel */
 void *LocalAddr; /* Local Address of Shared Variable */
 enum _c2_ValueType TypeTag;/* Shared Variable Type */
 struct _c2_sLockSet *Next;/* Pointer to Next Lock */
 int UID;         /* UID of shared variable */
} _c2_LockSet;
```

The above structure represents the information maintained by each UC for every shared variable that it uses. The first field, *IsRemote*, identifies if the name sharing relation which contains this shared variable is on a different PVM task or on the local one. The rest of the fields are used to maintain various information that is required while obtaining and releasing locks. The fields *NSRelUID*, *NSRelGraphPath*, *NSRelIndex* and *RequestSent* are useful only if the relevant name-sharing relation is on a different PVM task.

Field *NSState* is also used in this process. It can take on one of the following values at any time.

- _c2_NOTINIT - UC Node's list of required locks has not been created.
- _c2_NOBIND - The list has been created, but no locks have been acquired.

- _c2_INPROG - At least one lock has been acquired, but more are needed.

An example of a structure of variable-type information follows. The fields of type *_c2_SeqVarQ* are data flow queues for data sent on incoming arcs. There is one queue for each input port. The fields with names that do not begin with "_c2" are all local variables.The field *_c2_Dummy* is used to prevent a situation in which a C structure containing no fields is created.

```
struct _c2_nv3 { /* UC = ADD2 */
 char _c2_Dummy;
 _c2_SeqVarQ Y2;
 Vect V2;
 int i;
 int sum2;
};
```

4.4.2 Graph / Call Node

Graphs provide a level of name scoping in CODE environment. Graphs are invoked by using Call Nodes. They consist of two C structures allocated on the heap. The first structure represents fixed type information and the second represents variable type information. The first contains a pointer to the second. The structure containing fixed-type information is as follows.

```
.struct _c2_sGraphBase {
 int UID;
 _c2_Index Index;
 char Path[PATH_ID_LENGTH]; /* The Path ID */

 _c2_GraphBase *Parent; /* Pointer to calling Graph */
 int CrepsToGo;        /* Count of unbound Creation Params */
 _c2_AddrMap *Map;     /* Address Table */

 void *LocalData; /* Pointer to struct of creation params */
};
```

The first three fields *UID*, *Index* and *Path* provide identity to each graph in a manner very similar to corresponding fields of UCs. Each graph contains a pointer to the structure representing its parent, *Parent*. Also, each graph contains a pointer to an address map, *Map*. Each address map is a linked list of pointers to CODE elements (on the heap) instantiated within the scope of the enclosing graph. These CODE elements may include UC's, Name Sharing Relations and other Graphs. Thus, in CODE the addresses of instances are determined by traversing an hierarchy of address tables. Traversal is possible due to the pointer each graph has to its parent. The field *CrepsToGo* is a integer representing the number of Creation Parameters that are as yet unbound. All creation parameter nodes in a graph must be bound before any of the UC's in the graph are fired. This is required in order to ensure that UC's within a graph which make use of a creation parameter nodes are not fired before the creation parameter nodes are bound.

The last field of the structure points to the variable-type information of each graph. An example of the variable-type information follows.

```
struct _c2_gv14 { /* Graph = DoBarnes */
 char _c2_Dummy;
 int NUMPART;
 int _c2_NUMPART;
 int NUMIT;
 int _c2_NUMIT;
};
```

In the above structure, each field without a "_c2_" prefix is a creation parameter. For each creation parameter field there is another field with the same name alongwith a "_c2_" prefix. These fields indicate whether the corresponding creation parameter is bound or not.

4.4.3 Name Sharing Relation (NSRel)

Name sharing relations are represented in a similar manner. The fixed type information is as follows.

```
struct _c2_sNSRelBase {
 int UID;
 _c2_Index Index;
 char Path[PATH_ID_LENGTH]; /* The Path ID */
 int Crepped;
 _c2_GraphBase *MyGraph;
```

```
 void (*InitProc)();

 void *LocalData; /* Pointer to local variables struct */

};
```

Again *UID*, *Index* and *Path* provide identity to the Name Sharing Relation. The field *Crepped* indicates whether creation parameters used by this NSRel have been bound or not. *MyGraph* is a pointer to the structure representing the enclosing Graph. Function pointer *InitProc* points to a function which initializes shared variables within the NSRel. It is executed only once at the time of NSRel creation. *LocalData* points to the variable type information which contains all the shared variables of this NSRel. An example of the variable type information follows.

```
struct _c2_nv5 { /* NSRel = x1 */
 char _c2_Dummy;
int x1; /* shared variable */
int _c2_x1_RCount;/* No of current readers */
int _c2_x1_WCount;/* No of current writers */
_c2_NSLink *_c2_x1_WQ;/* list of UC's that want write access
                         locks */
_c2_NSLink *_c2_x1_RQ;/* list of UC's that want read access
                         locks */
int _c2_x1_UID; /* UID of shared variable */
};
```

In the above structure, the only shared variable is *x1*. All the others are status fields. The field *_c2_x1_RCount* represents the number of UC nodes which have currently been provided read access to *x1*. Field *_c2_x1_WCount* represents the number of UC nodes which have currently been provided write access to *x1*. This may take values 0 or 1 only. i.e. there may be many concurrent readers or one exclusive writer for each shared variable. Writers are provided non-preemptive priority over readers. The fields *_c2_x1_WQ* and *_c2_x1_RQ* are pointers to lists of UC nodes queued up to obtain write and read access locks to *x1*.

4.4.4 Creation Parameter Node (CreP)

As already mentioned Creation Parameter Nodes are part of the data structure of Graphs.

4.4.5 Interface Nodes

There are two types of interface nodes - input and output. These are used to map other CODE elements such as UCs, NSRels and Creation Parameter nodes. They are used at translate time. However, at runtime, they are not represented. The other CODE elements are directly mapped.

4.5 Mapping Routine

The routine used to map CODE elements to PVM tasks is actually a hashing algorithm. It takes as input the PathID of each CODE element and hashes it to a PVM task. The hash formula is (Sum of Numbers in PathID) modulo (Number of PVM tasks). For example, if the PathID were to be "0/2/4[10][1]/3", then the individual numbers in the PathId are summed up (0+2+4+10+1+3) = 20. Then, the hash value would be (20 modulo (Number of PVM tasks)).

4.6 PVM Scheduler

The PVM runtime environment has its own non-preemptive scheduler, a version of which runs on each of the PVM tasks. Each PVM task maintains a ready queue of UC Nodes that must be run. The PVM task removes the UC node which is at the head of the list and runs its *CompProc*. UC Nodes are placed on and taken off this queue depending on their state. Their state is maintained in a variable called *QueueStatus*. UCs may be in any one of the following four states.

- _c2_IDLE - Node is not on ready queue and is not running
- _c2_ONQUEUE - Node is on ready queue, waiting to be run
- _c2_RUNNING - Node is currently running, not on ready queue

- _c2_NEEDSRUN - Node is not on ready queue and is running but its state has changed in such a way that it needs to be evaluated again when the current evaluation is complete.

QueueStatus takes on one of the above values at any point in time.

4.7 UC Computation

The *CompProc* of each UC follows the structure outlined below.

```
  /* Check to see if Name Sharing Lock List is created */

  If (NSState == NONINIT)
    create list of required locks
    NSState = NOBIND;
  Endif
  If (NSState == INPROG) goto GetLocks;

  /* Check firing rules */

  If FiringRule1 is not satisfied, goto EndFRule1;
  Perform bindings for FiringRule1;
  Goto StartComp;
EndFRule1:
  If FiringRule2 is not satisfied, goto EndFRule2;
  Perform bindings for FiringRule2;
  Goto StartComp;
EndFRule2:
      •
      •
  Return without doing any computation;

GetLocks:

  /* Get All Name Sharing Relation Locks */
  /* and then Run node's sequential computation */

StartComp:
  If GetAllLocks is TRUE NSState = NOBIND
  Else
    NSState = INPROG
    Return without doing any computation
  Endif

  Perform node's sequential computation;

  /* Evaluate routing rules */
```

```
  If RoutingRule1 is not satisfied, goto EndRRule1;
  Perform bindings for RoutingRule1;
EndRRule1:
  If RoutingRule2 is not satisfied, goto EndRRule1;
  Perform bindings for RoutingRule2;
EndRRule2:
     •
     •
Release All Shared Variable Locks
```

4.8 Data flow

Data flow in CODE (from an implementation stand-point) takes three forms.

4.8.1 Data flow from UC to UC

Due to the dynamic nature of the CODE model, data flow from one UC instance to another involves a runtime mapping of the output port of the sending UC instance to the input port of the receiving UC instance. Also, due to the lazy creation feature of CODE, receiving UC's are instantiated the first time they receive data.

The receiving node's indices are determined by the arc topology specification applied to the sending node's indices (graph, node, and port in the general case). For example, suppose the sending node (node A with index [4]) has the following routing rule.

```
TRUE => Y[3] <- n;
```

So, the sending node's node index is [4] and its port index is [3]. See Figure 4-5.

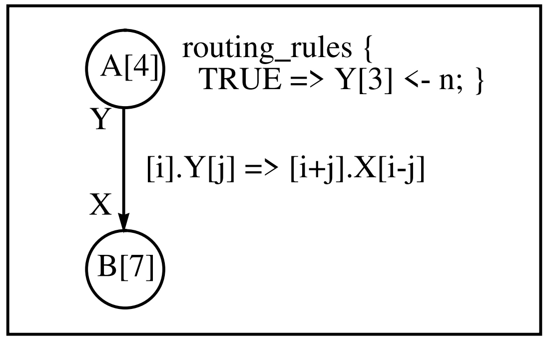

Figure 4-5. Example Data flow Send.

The arc topology specification determines that the receiving node is node B with index [7] since $7 = 4 + 3$. The receiving port is X[1] since $1 = 4 - 3$.

Once the mapping is done, the receiving UC must be instantiated if it has not already been instantiated. Then, the actual data transfer must take place.

However, before a UC can be instantiated, it must be determined whether the instantiation will be local (same PVM task) or non-local (remote PVM task). A mapping routine is called to determine on which PVM task the receiving UC instance must be created. If it is local, then it is instantiated and data is transferred into the data flow queue of the receiving UC's data structure. If it is non-local, then a PVM message is prepared by packing information needed to uniquely identify the UC instance alongwith a copy of the entire data that must be transferred. This PVM message is sent to the PVM task whose id is returned by the mapping routine. The remote PVM task receives the message and correspondingly instantiates (if necessary) the receiving UC instance and transfers the data into its data flow queue.

The arc topology specification defines a mapping that relates the send node's graph, node, and port indices to the receiving node's graph, node, and port indices. A cache is used to avoid recomputing the mapping for every send. Variable “addr” is the address of the structure that implements the receiving node and “taskid” is the id of the PVM task the receiving UC must be instantiated on. The entire algorithm follows.

```
// Input is sending node graph, node, and port indices.
if (input indices are not in cache) then
  Apply arc topology mapping to get receiving node indices.
  taskid = MapToPvmTask(receiving node path id info)
else
  taskid = value from cache
endif

if (myid != taskid) then
  pack receiving node path id info into pvm message
  pack data into pvm message
  send pvm message to taskid
else
  if (addr not in cache) then
   addr = GetAddr(receiving node ID, receiving node indices)
   Place (input indices, addr, taskid) tuple in cache.
  else
   addr = value from cache
  endif
endif

Add data to the receiving queue.
Place receiving node on ready queue for firing rule test.
```

In case the UC has to be instantiated on a different PVM task, a PVM message is sent to that task. The function that unpacks messages from other PVM tasks and takes appropriate action is called Message_Handler. A sample function which implements the above algorithm follows.

```
void _c2_g25(_c2_NodeBase *_c2_na, _c2_Index *_c2_pi, int
_c2_value) /* Port P*/
{
 int task_id;

 struct _c2_gv0 *_c2_g;
 _c2_GraphBase *_c2_gr;
 _c2_NodeBase *_c2_TmpNode;
 _c2_Index _c2_origpi;
 _c2_Index _c2_gi;
 _c2_Index _c2_ni;
```

```
 int _c2_AddToCache = 0;

 _c2_gr = _c2_na->MyGraph;
 _c2_ni = _c2_na->Index;
 _c2_origpi = *_c2_pi;
 _c2_g = (struct _c2_gv0 *) (_c2_gr->LocalData);

 /* Check Cache */

 if (!_c2_FindCache(&(((struct _c2_nv3 *)(_c2_na-
>Local Data))->_c2_g25Cache), &_c2_origpi, _c2_pi,
 &_c2_gi, &_c2_ni, &_c2_TmpNode, &task_id))
 {
   _c2_AddToCache = 1;

   /* perform mapping */

   {
     _c2_gi.NumInd = 0;
     _c2_ni.NumInd = 0;
     (*_c2_pi).NumInd = 0;
   }

   /* Map receiving UC to PVM task */

  task_id = _c2_MapToPvmTask(_c2_gr, &_c2_gi, &_c2_ni, 4,
4);
 }

 if (task_id != _c2_me)
 {
   /* prepare PVM message for remote instantiation*/

   pvm_initsend (PVMDATATYPE);
   pvm_pkstr (_c2_gr->Path); /* Parent graph Path ID */
   pvm_pkint (&_c2_gi.NumInd, 1, 1); /* Graph Indices */
   pvm_pkint (&_c2_gi.Ind[0], _c2_gi.NumInd, 1);
   pvm_pkint (&_c2_ni.NumInd, 1, 1); /* Node Indices */
   pvm_pkint (&_c2_ni.Ind[0], _c2_ni.NumInd, 1);
   pvm_pkint (&_c2_pi->NumInd, 1, 1);/* Port Indices */
   pvm_pkint (&_c2_pi->Ind[0], _c2_pi->NumInd, 1);
   j = 25;
   pvm_pkint (&j, 1, 1); /* g routine no.25 */
   pvm_pkint (&_c2_value, 1, 1); /* data */
   pvm_send (_c2_tids[task_id], 1);
 }
else
```

```
   /* Local Instantiation */

  _c2_g25_local(_c2_gr, &_c2_gi, &_c2_ni, _c2_pi, &_c2_Tmp-
Node, _c2_AddToCache, _c2_value);

if (_c2_AddToCache)     /* Add to Cache */
 _c2_EnCache(&(((struct _c2_nv3 *)(_c2_na->LocalData))-
>_c2_g25Cache), &_c2_origpi, _c2_pi, &_c2_gi, &_c2_ni,
_c2_TmpNode, task_id);
}
```

The function _c2_g25_local contains code that instantiates the receiving UC and transfers data into its data flow queue. It follows.

```
void _c2_g25_local(_c2_GraphBase *_c2_gr, _c2_Index
*_c2_gi, _c2_Index *_c2_ni, _c2_Index *_c2_pi, _c2_NodeBase
**_c2_NodeAddr, int _c2_AddToCache, int _c2_value)
{
 _c2_SeqVar **_c2_QVar;
 _c2_NodeBase *_c2_TmpNode;
 _c2_Value _c2_TmpValue;

 if (_c2_AddToCache)
   /* Instantiate if necessary */

  _c2_TmpNode = (_c2_NodeBase *) _c2_GetAddr(_c2_gr,
 4,_c2_ni);
 else
  _c2_TmpNode = *_c2_NodeAddr;

 _c2_QVar = ((struct _c2_nv4 *)(_c2_TmpNode->LocalData))-
>Q2;

 /* Insert data into data flow queue */

 _c2_TmpValue.type = _c2_IsAnInt;
 _c2_TmpValue.u.i = _c2_value;
 _c2_Insert(_c2_QVar, _c2_pi, &_c2_TmpValue);

 /* enqueue receiving UC for firing rule test */

 _c2_EnQueue(_c2_TmpNode);
 if (_c2_AddToCache)
  *_c2_NodeAddr = _c2_TmpNode;
}
```

Function GetAddr invoked above is responsible for returning the address on the heap where the structures representing instances of UC's, Graphs and Name-Sharing Relations reside. As a side-effect, it instantiates them if they have not already been instantiated. In order to achieve this, it makes use of the address map which is part of each graph's data structure. While the function GetAddr itself is not CODE-specific (it is part of the runtime library and does not vary from program to program), it calls a function MakeObj which is CODE specific. This function MakeObj creates runtime instances of UC's, Graph's and NSRel's. It uses the template type ID (UID) to determine which template to instantiate. It uses helper functions MakeGraph, MakeNode and MakeNSRel, to initialize the fixed type information.

4.8.2 Data flow from UC to CreP

Creation Parameter Nodes complicate matters in a distributed environment. This is due to the fact that they must be initialized on every PVM task which carries an instance of their enclosing graph. An instance of a graph will be present on a PVM task if any of its children is (or will be). Thereby, It is not possible to determine apriori if a particular PVM task will carry that graph instance or not. Thus, whenever a creation parameter node is created, it is broadcast to all PVM tasks and initialized on all of them.

The algorithm followed to achieve this is very similar to that of data flow from UC to UC. The only difference being that creation parameter nodes are not mapped, but instantiated on all PVM tasks. A sample routine which does this is given below.

```
void _c2_g111 (_c2_NodeBase *_c2_na, _c2_Index *_c2_pi, int
_c2_value) /* Port NUMPROC */
{
 int task_id;

 struct _c2_gv0 *_c2_g;
 _c2_GraphBase *_c2_gr;
 _c2_NodeBase *_c2_TmpNode;
 _c2_Index _c2_origpi;
 _c2_Index _c2_gi;
```

```
 _c2_Index _c2_ni;
 int _c2_AddToCache = 0;

 _c2_gr = _c2_na->MyGraph;
 _c2_ni = _c2_na->Index;
 _c2_origpi = *_c2_pi;
 _c2_g = (struct _c2_gv0 *) (_c2_gr->LocalData);

   /* Check Cache */

 if (!_c2_FindCache (&(((struct _c2_nv1 *) (_c2_na->Local-
Data))->_c2_g111Cache), &_c2_origpi, _c2_pi, &_c2_gi,
&_c2_ni, &_c2_TmpNode, &task_id))
 {
   _c2_AddToCache = 1;

   /* Perform mapping */

   {
     _c2_gi.NumInd = 0;
     _c2_ni.NumInd = 0;
     (*_c2_pi).NumInd = 0;
   }

   /* Dont add to Cache. This function executed only once */

   _c2_AddToCache = 0;
 }
 {
   /* Prepare PVM message to broadcast CreP */

   int j, k;
   pvm_initsend (PVMDATATYPE);
   pvm_pkstr (_c2_gr->Path); /* Parent Graph Path Id */
   pvm_pkint (&_c2_gi.NumInd, 1, 1);/* Graph Indices */
   pvm_pkint (&_c2_gi.Ind[0], _c2_gi.NumInd, 1);
   pvm_pkint (&_c2_ni.NumInd, 1, 1);/* Node Indices */
   pvm_pkint (&_c2_ni.Ind[0], _c2_ni.NumInd, 1);
   pvm_pkint (&_c2_pi->NumInd, 1, 1);/* Port Indices */
   pvm_pkint (&_c2_pi->Ind[0], _c2_pi->NumInd, 1);
   j = 111;
   pvm_pkint (&j, 1, 1);/* _c2_g routine no. 111 */
   pvm_pkint (&_c2_value, 1, 1); /* CreP Value */

   /* SEND PVM MESSAGE TO ALL PVM TASKS OTHER THAN ITSELF */

   for (k = 0; k < _c2_nhost; k++)
```

```
    if (k != _c2_me)
      pvm_send (_c2_tids[k], 1);
 }
  /* Initialize local instance of creation parameter node */

 _c2_g111_local (_c2_gr, &_c2_gi, &_c2_ni, _c2_pi, &_c2_Tmp-
Node, _c2_AddToCache, _c2_value);

 /* Will not be added to Cache */

 if (_c2_AddToCache)
 _c2_EnCache (&(((struct _c2_nv1 *) (_c2_na->LocalData))-
>_c2_g111Cache), &_c2_origpi, _c2_pi, &_c2_gi, &_c2_ni,
_c2_TmpNode, task_id);
}
```

The function _c2_g111_local which performs the actual initialization is given below.

```
void_c2_g111_local (_c2_GraphBase *_c2_gr, _c2_Index
*_c2_gi, _c2_Index *_c2_ni, _c2_Index *_c2_pi, _c2_NodeBase
*_c2_NodeAddr, int _c2_AddToCache, int _c2_value)
{
 _c2_SeqVar **_c2_QVar;
 _c2_NodeBase *_c2_TmpNode;
 _c2_Value _c2_TmpValue;

 /* Get addr of Parent Graph (create if necessary) */

 _c2_gr = (_c2_GraphBase *) _c2_GetAddr (_c2_gr, 8, _c2_gi);
 if (((struct _c2_gv14 *) (_c2_gr->LocalData))->_c2_NUMPROC
== 1)
 {
   /* Error !! Initialized Twice !!*/
   _c2_KillComp (_c2_CREPTWICE);
 }
 else
   /* Set CreP Flag to true */

  ((struct _c2_gv14 *) (_c2_gr->LocalData))->_c2_NUMPROC=1;

   /* Initialize CreP */

 ((struct _c2_gv14 *) (_c2_gr->LocalData))->NUMPROC =
(int)_c2_value;

   /* Enqueue waiting UC's for firing */
```

```
 _c2_InitWaitingNodes (_c2_gr);
 if (_c2_AddToCache)            /* Dont Add to Cache */
 *_c2_NodeAddr = _c2_TmpNode;
}
```

Two routines from CODE's PVM runtime library are used in the above function. InitWaitingNodes is a function that checks if all creation parameter nodes of the graph have been bound. If so, it will enqueue all the UC's belonging to that graph that were waiting for the creation parameter nodes to be bound, for firing. The function KillComp is invoked on the error condition in which a creation parameter node is initialized twice.

4.8.3 Data flow between UC and NSRel

Before any form of data flow occurs between UC's and NSRel's, a list of locks which the UC should acquire must be created. In the runtime environment, one C function is generated to create each lock that must be acquired by a UC. Each UC invokes these routines the first time it runs. Of course, these routines are created only if the UC uses shared variables. A sample routine is given below.

```
void _c2_g80 (_c2_NodeBase *_c2_na, _c2_Index *_c2_pi)
{
 int i;
 struct _c2_gv14 *_c2_g;
 _c2_GraphBase *_c2_gr;
 _c2_NSRelBase *_c2_TmpNSRel;
 _c2_Index _c2_gi;
 _c2_Index _c2_ni;
 void *_c2_LocalAddr;
 int _c2_LocalReqType;
 _c2_LockSet *_c2_NewLock = (_c2_LockSet *) _c2_shmalloc(-
sizeof (_c2_LockSet));

 _c2_gr = _c2_na->MyGraph;
 _c2_ni = _c2_na->Index;
 _c2_g = (struct _c2_gv14 *) (_c2_gr->LocalData);

 /* Initialize pointer to Local Address of Shared Variable.
This is the address of the UC's own copy of the shared vari-
able */
```

```
 _c2_LocalAddr = (void *) &((struct _c2_nv25 *) (_c2_na-
>LocalData))->Tree;

 /* This UC is a READER of this Shared Variable */

 _c2_LocalReqType = _c2_READER;

  /* Perform Mapping */

 {
   _c2_gi.NumInd = 0;
   _c2_ni.NumInd = 0;
   (*_c2_pi).NumInd = 0;
 }
 {
   /* This block of code maps the NSRel containing
      this shared variable to a PVM task */

   int tempsum;
   int j;

   tempsum = _c2_PvmPathSum (_c2_gr->Path);
   for (j = 0; j < _c2_ni.NumInd; j++)
     tempsum += _c2_ni.Ind[j];
   tempsum += 23;
   i = tempsum % _c2_nhost;
 }

 /* Is the NSRel containing this shared Variable local ? */

 if (i == _c2_me)
 { /* Yes, Instantiate it */

   _c2_TmpNSRel = (_c2_NSRelBase *) _c2_GetAddr (_c2_gr, 23,
&_c2_ni);

   /* Initialize support fields */

   _c2_NewLock->NSRelAddr = _c2_TmpNSRel;
   _c2_NewLock->RCount = &((struct _c2_nv23 *) (_c2_TmpN-
SRel->LocalData))>_c2_Tree_RCount;
   _c2_NewLock->WCount = &((struct _c2_nv23 *) (_c2_TmpN-
SRel->LocalData))>_c2_Tree_WCount;
   _c2_NewLock->RQ = &((struct _c2_nv23 *) (_c2_TmpNSRel-
>LocalData))>_c2_Tree_RQ;
   _c2_NewLock->WQ = &((struct _c2_nv23 *) (_c2_TmpNSRel-
```

```
>LocalData))>_c2_Tree_WQ;
   _c2_NewLock->SharedAddr = (void *) &((struct _c2_nv23 *)
(_c2_TmpNSRel->LocalData))->Tree;
   _c2_NewLock->UID = ((struct _c2_nv23 *) (_c2_TmpNSRel-
>LocalData))>_c2_Tree_UID;

   /* NSRel is local */

   _c2_NewLock->IsRemote = 0;
 }
 else
 { /* No, wait until lock is requested to instantiate */

   _c2_NewLock->IsRemote = 1;
   _c2_NewLock->RequestSent = 0;
 }
 _c2_NewLock->ReqType = _c2_LocalReqType;
 _c2_NewLock->LocalAddr = _c2_LocalAddr;
 _c2_NewLock->NSRelUID = 23;
 _c2_NewLock->NSRelGraphPath = _c2_gr->Path;
 _c2_NewLock->NSRelIndex = _c2_ni;
 _c2_NewLock->UID = _c2_23_Tree_SharVarUID ();
 _c2_NewLock->TypeTag = _c2_IsAStructPtr;
 _c2_InsertLock (&_c2_na->HeadLock, _c2_NewLock);
}
```

After the list of locks is created, the UC begins acquiring these locks in a specific order as described in Section 3.6.3 in order to avoid deadlock. In order to acquire locks the UC invokes a CODE runtime library function called GetAllLocks. This function is given below.

```
 /* Acquire all locks in list from NextLock to end. If fail
to get a lock, leave NextLock pointing at it to try again
next time.Return 1 if end of list reached (all locks
acquired), return 0 if fail to get one-- hence more locks
needed */

int _c2_GetAllLocks(_c2_NodeBase *_c2_na, _c2_LockSet
**NextLock)
{
 _c2_LockSet *cur;

 /* Get all locks in node’s list */

 cur = *NextLock;
```

```
 while (cur != 0) {

 /* Is the NSRel on the same pvm task?*/

 if (!cur->IsRemote)
 { /* NSRel is local, try to acquire lock */

   if (_c2_GetLock(cur->NSRelAddr, cur->RCount, cur-
>RQ,cur->WCount, cur->WQ, cur->ReqType, _c2_na))
   {
     /* Got lock. Map Addr of Shared Variable to Local UC’s
        Data Structure */

     switch (cur->TypeTag) {
      case _c2_IsAnInt:
       *((int *)(cur->LocalAddr)) = *((int *)(cur->Share-
dAddr));
       break;
      case _c2_IsADouble:
       *((double *)(cur->LocalAddr)) = *((double *)(cur-
>SharedAddr));
       break;
      case _c2_IsAnArrayPtr:
      case _c2_IsAStructPtr:
       *((void **)(cur->LocalAddr)) = *((void **)(cur-
>SharedAddr));
       break;
      case _c2_IsAChar:
       *((char *)(cur->LocalAddr)) = *((char *)(cur->Share-
dAddr));
      break;
    }
   }
   else
   {
     *NextLock = cur;
     return 0;              /* Failed to get this lock */
   }
   cur = cur->Next;
 }
 else /* or on a different Pvm task ? */
 {
   /* Send a PVM message to different PVM task requesting
       a lock on this shared variable */

    if (cur->RequestSent == 0)
    {
```

```
        int tempsum;
        int i;
        int TaskID;

        /* Map NSRel to PVM task */

        tempsum = _c2_PvmPathSum(cur->NSRelGraphPath);
        for (i = 0; i < cur->NSRelIndex.NumInd; i++)
          tempsum += cur->NSRelIndex.Ind[i];
        tempsum += cur->NSRelUID;
        TaskID = tempsum % _c2_nhost;

        /* Prepare PVM Message */

        pvm_initsend(PVMDATATYPE);
        /* NSRel ParentGraph Path ID */
        pvm_pkstr(cur->NSRelGraphPath);
        pvm_pkint(&cur->NSRelUID, 1, 1);/* NSRel UID */
        pvm_pkint(&cur->NSRelIndex.NumInd, 1, 1);/* Indices*/
        pvm_pkint(&cur->NSRelIndex.Ind[0], cur->NSRelIndex.-
NumInd, 1);
        pvm_pkstr(_c2_na->Path);/* Path ID of UC */
        pvm_pkint(&cur->ReqType, 1, 1);/* Reader or Writer */
        pvm_pkint(&cur->UID, 1, 1);/* UID of Shared Variable*/
        pvm_pkint(&_c2_me, 1, 1);/* My PVM Task ID */
        pvm_send(_c2_tids[TaskID], 3);
        *NextLock = cur;
        cur->RequestSent = 1;
       }
       return 0;
      }
    }
  return 1;
}
```

As the above function shows, this routine tries to acquire all locks until it runs into a lock that has already been acquired by some other UC or a lock on a shared variable which is on a different PVM task. In the latter case, the function sends a PVM message containing the identity of the shared variable and the UC requesting the lock. In both cases, the function returns false indicating that all locks have not been acquired. The other PVM task receives this message through its Message_Handler function and returns an approval message if that shared variable is not locked. That message is received by the local

Message_Handler which again enqueues the UC for execution. However, if the other PVM task finds the shared variable already locked, then it puts the UC on the queue and sends the local PVM task a message when the shared variable is unlocked. In that case, the UC starts all over again and sends a PVM message requesting the lock.

Once the execution of the UC completes, all locks must be released. Routine RelAllLocks from the runtime library performs this function. It releases all the locks in the same order as the one in which they were acquired. If a shared variable is local, it updates its value and releases the lock. If it is non-local, a PVM message is sent to that PVM task which contains the identity of the shared variable and its new value.The other PVM task updates the shared variable and unlocks it.

4.9 Message Handling

The function Message_Handler is responsible for handling all the PVM messages in CODE’s distributed environment. It is program-specific and is generated at compile time. Its structure is outlined below.

```
Perform a non-blocking receive

If (any message has arrived)
 {
  Switch( Message Tag)
  {
   case 1:
    /* Data transfer message from a UC to a UC/Crep

    Unpack Message
    Instantiate receiving UC if necessary
    Transfer Data to Data flow queue of receiving UC
    Enqueue UC for firing rule test
    break;

   case 2:
    /* Data transfer message on return from CODE graph */

    Unpack message
    Instantiate receiving UC if necessary
```

```
 Transfer Data to Data flow queue of receiving UC
 Enqueue UC for firing rule test
 break;

case 3:
 /* Request for lock on shared variable */

 Unpack Identity of NSRel containing Shared Variable
 Instantiate NSRel if necessary
 Unpack Identity of UC requesting lock and type of lock
 Try to acquire lock
 If (successful)
    pack identity of UC requesting lock
    pack the current value of shared variable
    send PVM message(type 4)to PVM task on which the
       UC resides
 Endif
 break;

case 4:
 /* Approval of request for lock on shared variable */

 unpack the identity of the UC.
 get addr of UC
 update its list of locks to reflect that the
  last requested lock has been acquired
 update UC’s structure to take the shared variable’s new
 value

 enqueue UC for execution (or to acquire further locks)
 break;

case 5:
 /* Request to release lock on shared variable */

 unpack identity of shared variable
 get addr of shared variable
 update the shared variable with new value
 enqueue for execution all UC’s queued for access to
  shared variable
 break;

case 6:
 /* Request to enqueue UC for execution */

 unpack identity of UC
 get addr of UC
```

```
    enqueue UC for execution
    break;

   case 99:
    /* terminate run */

    stop execution and end run
    break
  }

  check if a message has arrived
}
```

4.10 Linking and Running

Suppose the user's program is called "prog.grf". CODE's PVM translator creates a directory called "prog.pvm" and places within it all program text that must be compiled for PVM. It also creates a Makefile which permits the user to use the UNIX "make" facility to automatically compile these files and create an executable parallel program.

The Makefile contains instructions to link in all necessary runtime libraries, including the CODE Runtime Library and the PVM library. Their structure is as outlined in Figure 4-6.

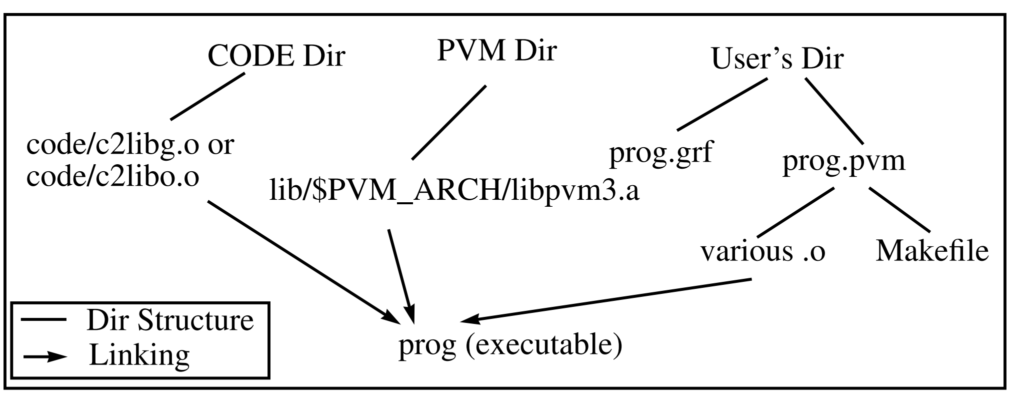

Figure 4-6. Files Linked for PVM.

That concludes the detailed description of the implementation of the distributed execution environment of CODE for PVM.

Chapter 5. Experimental Results

In this chapter, results of an extensive set of experiments which were conducted to evaluate the distributed execution environment will be presented.

5.1 Goal

The goal of these experiments is to evaluate the effectiveness of the distributed execution environment and to compare its performance with those of hand-coded native programs that run on the same execution environment. Performance is measured by comparing the execution times of CODE generated PVM programs with those of well-structured hand-coded PVM programs. The focus is on the relative performance of CODE generated versus hand-coded PVM programs rather than absolute performance measures.

5.2 Set Up

A set of standard programs were chosen to serve as benchmarks for our distributed execution environment. These include the Block Triangular Solver [Don86],the Life [Gar70] program and the Barnes Hut [Bar86, Cha92] algorithm.These programs use different patterns of communication.

We also used a synthetic benchmark in order to evaluate the following features of our system.

- efficiency of communication
- effectiveness of placement

The synthetic benchmark system varies the ratio of computation and data transfer.

5.3 Execution Environment

The execution environment for these experiments is a network of RS6000 workstations coupled by a FDDI network. This is a fairly commonly used execution environment for PVM and is thus a representative host for these experiments.

5.4 Example Programs

5.4.1 The Block Triangular Solver

This program solves a set of simultaneous linear equations Ax = B for a known lower triangular matrix. It uses a parallel algorithm developed by Jack Dongarra and Danny Sorenson[Don86].

Values obtained by running CODE generated programs as well as hand-coded programs on matrices of size 1200x1200 are as follows.

Table 5-1. Block Triangular Solver Timings.

Processors	CODE pgm (secs)	Hand-coded PVM pgm (secs)
1	0.23	1.44
2	1.39	3.32
3	3.18	2.79
4	2.15	2.57
5	2.34	2.37
6	2.55	2.30
8	2.71	2.20
10	2.57	2.09

In these experiments the number of blocks is set to the number of processors, so that the granularity of computations decreases as the number of processors increase. This is one reason why speedups do not follow the expected log N behavior. Another reason is that the amount of communication is increasing. This is because every time that part of the system which has been solved by a processor must be broadcast to all the other processors.

As may be observed, the timings of both CODE and hand-coded programs are comparable. In the case of one processor, the CODE program is faster than the corresponding hand-coded program. This is because the hand-coded program has been written in a generalized fashion for any number of processors. Thus, it does not take advantage of the fact that on a single processor, there is no need for explicit message-passing. In the case of two processors, again the CODE program is faster than the hand-coded program. This is due to good placement of units of computation which ensures that some of the data do not have to be explicitly transmitted between processors.

5.4.2 The Life program

The life program is based on the Game of Life [Gar70]. The Game of Life consists of a rectangular grid of cells, each surrounded by eight neighbors. Each cell can be on("alive") or off("dead") during each iteration ("generation") of the program. The three rules which govern the next generation of cells in the grid are as follows.

- survivals - each living cell with two or three living neighbors survives.
- deaths - each living cell with four or more living neighbors dies from overpopulation. Each cell with fewer than two live neighbors dies of isolation.
- births - each dead cell with exactly three living neighbors will produce a living cell.

Table 5-2. Life Program Timings.

Number of processors	CODE pgm (secs)	Hand-coded PVM pgm (secs)
1	12.62	13.18
2	8.63	8.69
3	6.38	6.89
4	5.59	5.65
5	5.06	4.98

Table 5-2. Life Program Timings.

Number of processors	CODE pgm (secs)	Hand-coded PVM pgm (secs)
6	4.67	4.49
8	4.24	3.95
10	4.08	3.73
12	3.88	3.43

Table 5-2 was generated by running CODE and hand-coded PVM programs on 1200x1200 matrices for 10 generations. As can be observed from the above table, not only do CODE and hand-coded programs execute in comparable timings, they also show very similar patterns of speedup. Again, note that the granularity of the computations decreases as the number of processors increase.

5.4.3 The Barnes-Hut Algorithm

The Barnes-Hut algorithm [Bar86, Cha92] computes the gravitational interactions among N particles and also computes their positions over time. Its order of execution is O(N log N).

Values obtained by running CODE generated programs as well as hand-coded programs on 80 particles for 40 iterations are as follows.

Table 5-3. Barnes-Hut Algorithm Timings.

Processors	CODE pgm (secs)	Hand-coded PVM pgm (secs)
1	2.74	3.48
2	19.18	2.68
3	6.65	2.48

Table 5-3. Barnes-Hut Algorithm Timings.

Processors	CODE pgm (secs)	Hand-coded PVM pgm (secs)
4	21.14	2.93
5	21.09	3.63
6	20.45	3.85
8	22.42	4.44
10	24.21	5.07

This is an example where the CODE generated program performs poorly compared to the hand-coded one. There are a couple of reasons for this. One reason is that the CODE program uses a shared variable. The shared variable implementation by the PVM translator involves a lot of overhead especially when the shared variable and the UC manipulating it are not placed on the same PVM task. In this case, every time a UC using the shared tree executes it has to get a fresh copy of the tree from the remote PVM task on which the tree resides.Thus, it may be observed for the cases when the number of processors is 2, 4, 5, 6, 8, 10 that the CODE program is very slow. However, for three processors, while not comparable to hand-coded programs, the timing is far better compared to the others. This is partly due to good placement of the UC of interest and the shared tree. Another reason is that the CODE program and the corresponding hand-coded program use different data structures, which in this case results in the amount of communication increasing by three times in the CODE program compared to the hand-coded one. One of the reasons for using different data structures is that CODE does not provide explicit support for pointers.

5.5 Synthetic Benchmark

To create a synthetic benchmark, a CODE program consisting of two layers of UC's was written. The first layer of UC's perform some computation and then send messages to different UC's in the second layer. The second layer UC's send messages back to corresponding UC's in the first layer.

The factors under consideration were computation and communication. The first set of runs was conducted by keeping the computation fixed at 1,200,000 floating-point additions and varying the amount of communication (message size). The number of iterations was fixed at 100.

Table 5-4. Varying Communication with fixed Computation.

Number of Processors	Communication size 12000 bytes (secs)	Communication Size 120000 bytes (secs)	Communication Size 1200000 bytes (secs)
1	10.82	10.84	10.84
2	11.09	11.10	11.10
3	6.09	9.84	55.60
4	5.63	9.20	37.92
5	4.32	7.07	27.85
6	4.00	6.05	23.23
8	3.25	4.66	17.86
10	2.81	3.61	14.40
12	2.32	3.26	14.61

From the above table, it is observed that when the computation is two orders of magnitude higher than communication, the speedup is much higher than when computation is one order of magnitude higher than communication. When the two are of the same magnitude there is no speedup at all.

The second set of runs were made by keeping the communication size fixed at 120000 bytes and varying the amount of computation. The number of iterations were again fixed at 100.

Table 5-5. Varying Computation with fixed Communication.

Number of Processors	Computation 600000 add statements (secs)	Computation 1200000 add statements (secs)	Computation 2400000 add statements (secs)
1	5.47	10.84	21.56
2	5.79	11.10	21.81
3	6.85	9.84	15.06
4	6.57	9.20	14.30
5	5.16	7.07	10.97
6	4.49	6.05	9.66
8	3.34	4.66	7.27
10	2.76	3.61	5.97
12	2.52	3.26	5.05

Again, it is observed that higher speedups are obtained by increasing computation over communication and vice-versa.

Further, a comparison between CODE's synthetic program and the corresponding hand-coded program is provided for one set of values - where computation is 1200000 floating-point additions, communication size is 1200000 bytes and the number of iterations is 100.

Table 5-6. Synthetic Benchmark Timings.

Number of Processors	CODE pgm (secs)	Hand-coded PVM pgm (secs)
1	10.84	25.58
2	11.10	48.00
3	55.60	35.79
4	37.92	27.67
5	27.85	23.28
6	23.23	19.55
8	17.86	15.20
10	14.40	13.14
12	14.61	11.86

The average execution time of the benchmark is similar in pattern to the previous cases. The CODE and hand-coded programs have similar execution times except for very small number of processors.

We can conclude that the performance of CODE-generated PVM program is comparable to hand-coded PVM program except where placement issues are not treated in a comparable manner. Placement is a separate issue which will be discussed in later work.

Chapter 6. Related Work

There have been many attempts to abstract parallel and distributed computing. Some have used graphical approaches while others have used the idea of enhancing existing languages by providing new primitives. In this chapter an example of each will be examined, HeNCE[Beg91] and Linda[S-ci92] respectively. Newton[New93a] gives a through survey of related work. Since, this recent survey is available, we give only a brief sketch of the two systems.

6.1 HeNCE

HeNCE is a graphical parallel programming environment. It is very similar to CODE in terms of goal, function, philosophy as well as the look and feel. The two have similarities and dissimilarities, which are outlined below. Much of what is outlined below is from [Bro94].

Both CODE and HeNCE support graphical/textual interfaces. Both allow calls to external sequential routines, enforce type checking and perform automatic storage management.

CODE provides a complete and very expressive set of firing rules. While this is an asset for an experienced CODE programmer, it may be daunting to a new user. However, in HeNCE firing rules are fixed and implicit. Nodes are permitted to fire when all their predecessors have fired.

HeNCE does not allow hierarchical structuring of programs. i.e., HeNCE graphs cannot call other HeNCE graphs. This restricts the expressiveness of the programming model. CODE graphs can invoke other CODE graphs.

HeNCE graphs must be acyclic while CODE graphs may be cyclic. Also, nodes in HeNCE are not decoupled from arcs. All data flow is implicit. But, in CODE each node is decoupled from arcs leaving it and data flow is explicit. All this ensures that CODE can represent more complex dynamic communication patterns than HeNCE.

Most importantly, HeNCE produces parallel code only for the PVM system. But, CODE is a retargetable language. It produces code for PVM as well as other targets such as the Sequent Symmetry shared memory system. The design of CODE is such that its translators operate much closer to the hardware of the target system. This enables them to produce more efficient code. Also, architecture specific optimizations may be made in CODE. HeNCE depends on PVM to make architecture specific optimizations. No efficient implementation of HeNCE was ever attempted. It starts a UNIX process every time it runs a node. This does not mean that an efficient implementation is impossible. However, performance comparisons between CODE and HeNCE would not be valid.

The HeNCE system allows the programmer to define a cost matrix. This helps it to make intelligent choices about running the program. The CODE system has no equivalent feature.

One can summarize the differences between CODE and HeNCE by saying that CODE is more capable and at least as efficient. But HeNCE is more concise and simpler for beginning programmers.

6.2 Linda

Linda centers on a logical model of memory closely related to relational data bases. While a conventional memory's storage unit is the physical byte, Linda memory's storage unit is the logical tuple, where a tuple is an ordered set of values. While elements of a conventional memory are accessed by addresses, elements in Linda are accessed by logical name, where a tuple's name is any selection of its values.

Linda supports the notion of a globally shared associative memory space referred to as tuple space. In reality, portions on this memory space will reside on different processors, but will appear as one global memory space to all component Linda processes. Messages in Linda are never exchanged between two processes explicitly. Instead, a process with data to communicate adds to the tuple space and a process that needs data seeks it, likewise, in tuple space. There are six operations defined over tuple space - out(), in(), inp(), rd(), rdp() and eval(). out() places a tuple in the tuple space, in() removes a tuple from the tuple space, inp() is a non-blocking form of in(), rd() reads the value of the tuple but leaves it in place, rdp() is a non-blocking form of rd() and eval() creates a new tuple.

Thus, it may be observed that the Linda model of programming is substantially different from that of CODE. However, it has been shown that Linda is inefficient over distributed memory architectures. This is because Linda[Sci92] basically is a shared memory model. The runtime execution model remains the same over all the architectures on which it runs.

6.3 Others

There are a variety of other systems for parallel programming. They include Paralex[Bab92], PPSE[Lew90] and P4[But92]. They are discussed in detail in the literature.

Chapter 7. Conclusion and Future Work

In this thesis, the design and implementation of a distributed execution environment in PVM for the CODE 2.0 graphical parallel programming system has been presented. This demonstrates the effectiveness of the CODE system in supporting multiple targets. Also, the fact that the CODE system works equally well with both shared and distributed memory architectures is demonstrated. Further, it makes the CODE system portable to large set of diverse architectures.

The effectiveness of the distributed execution environment has been demonstrated by the execution of several standard benchmarks. The CODE generated programs have been shown to run with competitive speed to hand-coded native programs in the same environment.

While the algorithm for distribution of work is reasonable in the current implementation, it does not take into account the amount of data transfer between nodes. An extension to this thesis would be to study placement of units of computation(UC's) such that UC's that communicate with the maximum amount of data are placed on the same PVM task. This would significantly reduce communication overheads for the PVM implementation of CODE programs.

An interesting extension would be to explore the possibility of collapsing portions of the graph into efficient serial code (for mapping to other architectures).

Also, it would be interesting to determine if the cost of abstraction is dependent on the nature of the target architecture.

Appendix A.   Relevant PVM Primitives

A brief explanation of the PVM primitives used in the implementation of this thesis is given below.

- pvm_bufinfo () - returns information about the requested message buffer
- pvm_catchout () - catch output from child tasks
- pvm_config) - returns information about the virtual machine configuration
- pvm_exit () - tells the local pvmd that this process is leaving PVM
- pvm_initsend () - clear default buffer and specify message encoding
- pvm_mcast () - multicast the data in the active message buffer to a set of tasks
- pvm_mytid () - returns the tid of the process
- pvm_nrecv () - non-blocking receive
- pvm_parent () - returns the tid of the process that spawned the calling process
- pvm_pk* () - pack the active message buffer with arrays of prescribed data type
- pvm_recv ()- blocking receive
- pvm_send () - sends the data in the active message buffer
- pvm_spawn () - starts new PVM processes
- pvm_upk* () - unpack the active message buffer with arrays of prescribed data type

More detailed descriptions of these primitives may be found in the PVM3 Users Manual[Gei94].

Bibliography

[Bab92] Ö. Babaoglu, “Paralex: An Environment for Parallel Programming in Distributed Systems,” Proc. ACM Int. Conf. on Supercomputing, July, 1992.

[Bar86] J.Barnes and P. Hut, “A Hierarchical O(N log N) Force-Calculation Algorithm”, Nature, vol. 324, p. 446, 1986.

[Beg91] A. Beguelin, et al., “Graphical Development Tools for Network Based Concurrent Supercomputing”, Proc. Supercomuting ‘91, Albuquerque, NM, pp. 435-444, 1991.

[Bro94] James C. Browne, Jack Dongarra, Syed I. Hyder, Keith Moore, and Peter Newton, “Visual Programming and Parallel Computing”, University of Tennessee Tech Report CS-94-229, April 1994.

[But92] R. Butler and E. Lusk, User’s Guide to the P4 Programming System”, Tech. Report ANL-92/17, Argonne National Laboratory, 1992.

[Don86] J.J. Dongarra and D.C. Sorenson, “ SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs”, Argonne National Laboratory MCSD Technical Memorandum No. 86, Nov. 1986.

[Cha92] K.M. Chandy and S. Taylor, “An Introduction to Parallel Programming”, pp. 145-157, Jones and Bartlett, Boston, 1992.

[Gar70] M. Gardner, “Mathematical Recreations”, Scientific American, vol. 223, no. 4, pp. 120-123, 1970.

[Gei94] A. Geist, et al., “PVM 3 User’s Guide an d Reference Manual”, Oak Ridge National Laboratory, Tennessee, 1994.

[Lew90] T.G. Lewis and W. Rudd, "Architecture of the Parallel Programming Support Environment," Proc. CompCon'90, San Francisco, CA, Feb. 26 - Mar 2., 1990.

[New92] P. Newton and J.C. Browne, "The CODE 2.0 Graphical Parallel Programming Language", Proc. ACM Int. Conf. on Supercomputing, July, 1992.

[New93a] P. Newton, "A Graphical Retargetable Parallel Programming Environment and its Efficient Implementation", Ph.D. thesis, University of Texas at Austin, Dept. of Comp. Sci., 1993.

[New93b] P. Newton and S. Khedekar, *CODE 2.0 User Manual,* 1993.

[New93c] P. Newton, *CODE 2.0 Language Refererence,* 1993.

[Sci92] Scientific Computing Associates, Inc., C-Linda Reference Manual, New Haven, CT, 1992.

[Str91] B. Stroustrup, "The C++ Programming Language", Reading, Addison Wesley, 1991.

[Sun91] V.S. Sunderam, "PVM: A Framework for Parallel Distributed Computing," Concurrency: Practice and Experience, 2(4):315-339, Dec., 1990.

Vita

Rajeev Mandayam Vokkarne was born in Bangalore, India on May 10, 1970, the son of Govindarajan and Rajalakshmi Vokkarne . After graduating from Saradavilas College in Mysore, India in 1987, he enrolled at the University of Mysore where he received the degree of Bachelor of Engineering in 1991. His major areas of study were computer science and engineering. He entered the Graduate School of the University of Texas at Austin in January 1993.

Permanent address: 216 12th Main, Saraswathipuram, Mysore India 570009.

This dissertation was typed by the author.