The CODE 2.0 Graphical Parallel Programming Language*

Peter Newton
James C. Browne

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712

ABSTRACT

CODE 2.0 is a graphical parallel programming system that targets the three goals of ease of use, portability, and production of efficient parallel code. Ease of use is provided by an integrated graphical/textual interface, a powerful dynamic model of parallel computation, and an integrated concept of program component reuse. Portability is approached by the declarative expression of synchronization and communication operators at a high level of abstraction in a manner which cleanly separates overall computation structure from the primitive sequential computations that make up a program. Execution efficiency is approached through a systematic class hierarchy that supports hierarchical translation refinement including special case recognition. This paper reports results obtained through experimental use of a prototype implementation of the CODE 2.0 system.

CODE 2.0 represents a major conceptual advance over its predecessor systems (CODE 1.0 and CODE 1.2) in terms of the expressive power of the model of computation which is implemented and in potential for attaining efficiency across a wide spectrum of parallel architectures through the use of class hierarchies as a means of mapping from logical to executable program representations.

*This research is supported by a DARPA/NASA Research Assistantship in Parallel Processing administered by the Institute for Advanced Computer Studies, University of Maryland, by IBM Corp. through grant 61653, and by the State of Texas through TATP Project 003658-237.

1. INTRODUCTION

Common use of architectures that support macro-level coarse-grain parallel computation has been hindered by the difficulty of programming such machines. Effective parallel programming systems for these architectures must satisfy three practical goals. They must be easy to use, they must provide a portable representation basis for algorithms, and they must produce efficient executable parallel structures.

Several approaches to parallel programming are currently in use. Most, if not all, fail to satisfy one or more of these requirements to a major degree.

This paper will appear in Proc. ACM Int. Conf. on Supercomputing, July, 1992.

- Augment sequential languages with architecture-specific procedural primitives.

This approach permits the creation of efficient parallel programs, but the primitives supplied tend to be at such a low level of abstraction that they may be awkward to use for a wide variety of algorithms. Program development with them tends to be slow and error prone. In addition, parallel architectures are quite diverse, and their programming models are equally diverse. For this reason parallel programs written using architecturally specific extensions to sequential languages tend to be quite non-portable.

- Have compilers automatically detect parallelism in sequential language programs.

This approach clearly provides application portability. It is the case, however, that current parallel compilers often miss significant parallelism due to the difficulties engendered by name ambiguity in programs written in today's sequential programming languages [EIG91].

- Extend sequential languages to allow data partitionings to be specified.

One emerging trend is to include declarative partitioning of data structures in the sequential program formulation and to ask the compiler to utilize this parallel structure [HIR91]. This promising method is as yet immature.

The CODE Approach

The approach to medium and coarse grain parallel structuring in the CODE series of programming environments [BRO89,WER91], has been to have the programmer express the parallelism directly and declaratively at a high level of abstraction by composing macro-level units of computation into parallel structures in the form of generalized dependence graphs [BRO85]. These graphs are then automatically translated into architecture specific code. The syntax of the "programming language" of CODE is essentially a multigraph, with the semantics of the nodes and arcs mostly specified declaratively.

Versions of CODE prior to 2.0 demonstrated considerable success in ease of use, component reuse [LEE89] and efficiency of the resulting programs [JAI91], however they implemented only a rather static version of a unified model of parallel computation [SOB90]. There were many useful algorithms which were cumbersome to express in these earlier versions of

CODE. CODE 2.0 addresses the expressiveness limitations of the earlier versions of CODE with what appears on the basis of limited experimentation to be improved efficiency.

There is, of course, no free lunch in parallel programming. The goals of expressive power, ease of use and portability on one hand and efficiency of execution on the other are difficult to satisfy at the same time. The CODE 2.0 approach exploits several key technologies and a compromise in order to satisfy all three. The key technologies and their benefits relative to CODE 2.0's goals are shown in figure 1.1.

Technol. \ CODE 2.0 Goal	Ease of Use	Expressive	Execution Efficiency	Portabil
Integrated Graphical/ Textual Interface	○			
Declarative, Abstract, Dynamic Model	○	○		○
Program Component Reuse	○			
Hierarchical Class-based Model Basis			○	○
Lazy Creation, State Retention			○	
Integration of Function and Data Parallelism	○	○		

Figure 1.1 - CODE 2.0 Goals and Key Technologies

CODE 2.0 is easy to use because it rests upon an expressive declarative abstract model of parallel computation and because it has a graphical interface. Users create parallel programs by drawing a picture showing communication structure (a graph) and annotating it. There are no complex parallel programming primitives to learn. This approach also enhances portability since CODE's abstract program representation basis is not tied to any particular architecture.

Execution efficiency is approached by the use of a class hierarchy to represent the abstract model and by the software engineering structure of the CODE 2.0 translators. By means of class refinement, new heuristic translation optimizations may be added (in general) without rewriting old ones. The CODE 2.0 translation scheme may be thought of as a grab-bag of optimizations. This is further explained in section 6.

Also, the CODE 2.0 model is the result of a delicately balanced compromise between expressiveness and efficiency. A model that is abstract and powerful, especially in the area of runtime determined communication structure and data types, may be easy to use, but is difficult to implement efficiently. Yet, the model must be abstract and powerful enough to permit a wide class of algorithms to be conveniently expressed. Much of the CODE 2.0 design effort has gone into finding the correct level of abstraction and expressive power to achieve all of the system's goals. The attributes of the programming model at this balance point include the following.

1. User specified parallelism. The system does not have to automatically discover parallelism. It does have to automatically exploit it.
2. Declarative expression of algorithm communication and synchronization structure.
3. Runtime instantiation of a dynamic set of *instances* from a set of *types* that is fixed at runtime.
4. Lazy creation of instances. This refers to the fact that no CODE 2.0 object (node, graph, etc.) is instantiated at runtime until it is needed. A node is needed when a value is first placed onto an incoming arc. A side effect of this is that CODE 2.0 graphs may be recursive.
5. State retention. Once an instance is created, it is retained for possible future use and may be used many times. The cost of using a dynamic set of instances is essentially equal to the cost of using a static set, once the set is created.

The dynamic nature of the dependence graphs is fundamental to the CODE 2.0 model of parallel computation. Each node and each arc is a member of a type family of nodes and arcs. An instance of any type known to the system can be instantiated at any time. Instances are distinguished by integer valued indices.

A brief discussion of the interaction of dynamic parallel structuring and efficient execution may help to illustrate these trade-offs. Very often, parallel algorithms are most readily expressed as graphs whose structure is input-driven. Hence, the complete structure is not known until runtime. The structural parameters which may need to be resolved at runtime may be the number of occurrences of some unit of computation at a level of the dependence graph, a choice of what unit of computation to instantiate, or the pattern of connectivity among a family of object instances. CODE 1.0 and CODE 1.2 could express only the first type of dynamic structure. One simple illustration of these circumstances is a block-oriented parallel algorithm for the solution of a triangular set of linear equations. The desirable runtime parameters include the total size of the matrix and the size of the blocks into which it is to be decomposed. These two parameters together determine the number of logical units of computation and the connectivity among the units of computation. CODE 1.2 cannot readily express the general case of this algorithm. A single CODE 2.0 program easily captures the full dynamic structure of the algorithm for all cases. The result, however, is that the number of processes to be created is not known until runtime. A CODE program that solves this problem is presented in section 7.2.

Processes could be created at the beginning of the computation by the commonly used early binding strategy, but this would result in a tremendous waste of space and time. At the same time, if objects which are likely to be reused, as in loops in a dependence graph are not saved from iteration to iteration, the cost of creation becomes very high. The CODE 2.0 strategies of lazy creation and state retention minimize these overheads. At the same time the declarative graph structure of the program sometimes allows the runtime system to determine when a point of the graph will definitely not be revisited so that unneeded node instances can be discarded.

2. CODE 2.0 PROJECT STATUS

The model of parallel computation which is to be implemented by CODE 2.0 has been defined. A prototype implementation which utilizes all of the key technologies of the CODE 2.0 model has been developed and used to test the effectiveness of the model of computation on several modest scale programming projects. It translates graphs into parallel Ada, which is executed on a Sequent Symmetry. The major omission from the model of parallel computation in the prototype is a mechanism for sharing names and bound values among a set of computational elements. The results are serving as a guide for design and implementation of the next version of CODE 2.0 implementing the full model of parallel computation. This version will be available for public distribution and will be used in classes at The University of Texas at Austin. The releasable version of CODE 2.0 will be C (rather than Ada) oriented and will run under X-Windows on Sun and IBM RS6000 workstations. It will eventually produce parallel programs for several environments including both partitioned and shared memory machines.

3. CODE 2.0 USER INTERFACE

The CODE 2.0 user interface will not be discussed in detail in this paper, although much information about it can be inferred from the examples in the next section. The interface is a what you see is what you get graph editor in which nodes and arcs can be drawn and then annotated by means of filling out forms.

4. THE CODE 2.0 MODEL

In this section, we present an overview of CODE 2.0's basic model of computation, mostly by means of a simple example program. However, some preliminaries are needed. CODE 2.0 programs consist of a set of graph instances that interact by means of Call nodes. Graph instances in CODE 2.0 play the role of subroutines in conventional programming languages. The number and type of the instances is determined at runtime, but each graph is instantiated from one of a fixed set of graph templates. When the user draws a graph in the programming model, he is creating a template, not an instance. Instances are created at runtime when they are referenced from a Call node. This concept of dynamic instantiation from fixed templates is ubiquitous in CODE 2.0. Almost all model objects work in this way. It is helpful to view the objects the user creates with the graphical interface as *templates* for instances rather than single instances.

Graphs consist of nodes and arcs. Arcs represent channels for the flow of data from one node to another. They serve as unbounded (subject to system memory constraints) FIFO buffers. There are several types of nodes. They are shown in figure 4.1. As alluded to before, Call nodes specify arc connections to other graph instances. Interface nodes specify points at which a called graph can connect to another graph. Unit of computation nodes (UCs) represent basic sequential computations. Hence, they are the fundamental elements from which parallel programs are assembled. They consume data from incoming arcs, perform a computation, and place data onto outgoing arcs for other nodes to consume.

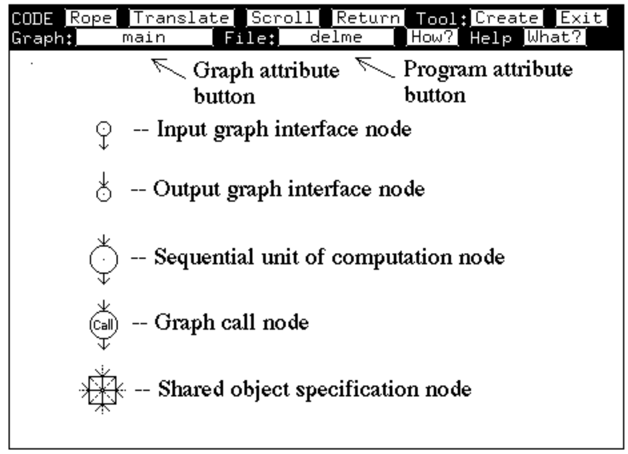
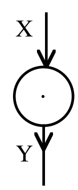


Figure 4.1 - CODE 2.0 Node Types

UC nodes have a large number of attributes. For example, the node's computation is expressed as a call to a subroutine in a conventional sequential language, and input rules specify conditions under which this computation is allowed to execute. UC nodes also have input and output ports for the communication of data to and from the node. Arcs are bound to specific ports. Hence, multiple arcs may enter or leave a node without ambiguity. Figure 4.2 shows an example.




```
input_rules {
  avail(X) => v = val(X)}
comp {
  Invert(v);}
output_rules {
  TRUE => Y = v;}
```

Figure 4.2 - Example UC Node

This example node has input port X and output port Y. The node may execute when there is a value available on the incoming arc bound to X. The value is placed into variable v and inverted by the node's computation, which is a call to a user-supplied sequential routine. Finally, the value is placed onto the outgoing arc bound to Y. UC nodes are further explained in section 4.3.

Multiple arcs may connect two nodes. Hence, CODE graphs are actually multigraphs. An extended example follows.

4.1 AN EXAMPLE PROGRAM: SGRID

Due to space considerations, the complete CODE 2.0 model of computation will not be presented [NEW91]. Instead, a simple example program will be used to demonstrate the most important aspects of the model. These include

1. Graph-based declarative model structure.
2. Hierarchical structuring via distinct graph instances interacting through Call nodes.
3. User-specified execution conditions for UC nodes.
4. Methods for specifying runtime determined communication patterns.

The example program is based loosely on a program in chapter two of [LUS87]. It computes values on a grid (represented as a 2-dimensional array) that satisfy Laplace's equation given fixed values on the boundaries of the grid. The algorithm is shown in figure 4.3.

```
Repeat Goal times:
   For each interior cell, set
      cell value to the mean of
      neighboring cell values.
```

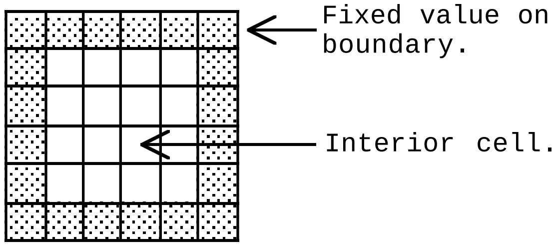


Figure 4.3 - Example Algorithm

This computation is readily parallelized by partitioning the rows to be updated during each iteration among the processors available.

4.2 HIERARCHICAL STRUCTURING

The CODE 2.0 program consists of two graphs, **main** and **Laplace** (see figure 4.4). Graph **main** reads input values and calls **Laplace** via Call node **CallLaplace** to compute the grid, which it then prints. Node **ReadInputs** is the first to execute. It reads from the user's terminal the size of the grid, the number of iterations, and the number of processors and places this information onto arcs **S**, **Goal**, and **NProcs**. These arcs are actual input parameters to graph **Laplace**. The arc leaving **CallLaplace** (arcs need not have names) is an actual output parameter.

A separate graph instance is associated with each Call node. Hence, there can be many instances of a graph in a program. For example, there could be two Call nodes in **main**, both calling **Laplace**. Then, there would be two instances of graph **Laplace** created. In fact, the number of graph instances need not be determined until runtime. This is because Call nodes may be replicated at runtime and because graphs may be recursive.

Actual parameters from **main** are bound to formal parameters in **Laplace**. There are two varieties of formal parameters in the CODE 2.0 model, creation parameters which are not displayed graphically and interface nodes which are represented by small circles. Interface nodes simply represent the binding of arcs. For example, actual parameter arc **Goal** in **main** is bound to interface node **Goal** in **Laplace**. This means that the goal number of iterations is passed from **ReadInputs** in **main** to **InitMat** in **Laplace**.

Graph **Laplace** (figure 4.5) has two creation parameters, **S** (the size of the grid) and **NProcs** (the number of processors to use for the main part of the computation). Creation parameters are given values exactly once when the graph instance that contains them is created at runtime. Hence, a graph cannot be created until values are placed on all actual parameter arcs that are bound to its creation parameters. When

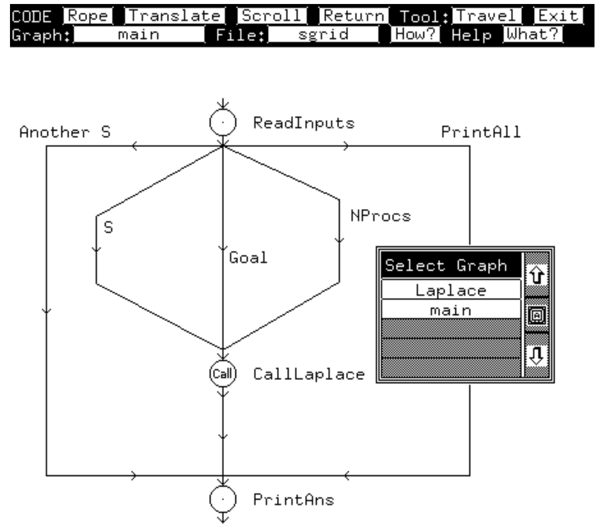


Figure 4.4 - Graph **main**

a graph is created, its creation parameters become constants in the scope of all nodes in the graph. This provides a convenient mechanism to broadcast values to all of the nodes in a graph upon their creation. Creation parameters are often used to create a graph instance at runtime to solve a problem of runtime determined size.

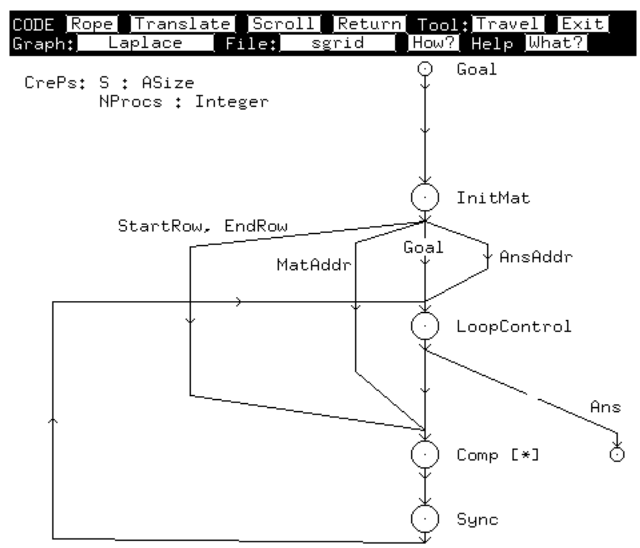


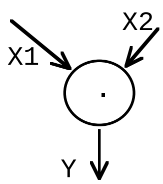
Figure 4.5 - Graph **Laplace**

Computation nodes in CODE 2.0 have user-supplied input firing conditions that determine the states in which node execution is permitted. Node **InitMat's** rule permits it to fire whenever a value is present on its incoming arc. When it fires, it allocates storage for the grid and partitions it by rows into **NProcs** pieces. Node **LoopControl** has input and output rules that cause it to act as a "for" node. It causes node family **Comp** to execute **NProcs** times, after which it passes the grid out of the graph. In the production version of CODE 2.0, the programming model will provide loop control nodes as primitive model elements.

There are **NProcs** copies of node **Comp** instantiated at runtime. The "[*]" by node **Comp** is a comment indicating that the node is instantiated more than once. These nodes perform the main computation, each acting on a piece of the grid extending from **StartRow** to **EndRow**. Of course, separate values of **StartRow** and **EndRow** are sent to each instance of **Comp**.

4.3 COMPUTATION NODES AND FIRING RULES

Let us now concentrate on the definition of a unit of computation node and the rules that define its execution conditions. Although node execution is to be considered as an atomic transition of the node and its incident arcs into a new state, node definitions consist of three principal parts, a set of input rules, a computation, and a set of output rules. A node's firing conditions are determined by its input and output rules, both of which consist of sets of Guard => Binding pairs. Figure 4.6 shows an example computation node with two input ports **X1** and **X2** and one output port **Y**. Variable **v** is local to the node.




```
input_rules {
   avail(X1) => v = v
   avail(X2) => v = v
comp {
    AddOne(v); }
output_rules {
    v > 0 => Y = v; }
```

Figure 4.6 - Computation Node

A computation node becomes eligible to fire when one or more of its input rule guards become TRUE and the corresponding rule(s) become enabled. At that point one of the enabled rules is selected nondeterministically, and its binding section executes. Function **avail** returns TRUE if and only if the its argument port contains a value. Function **val** returns that value and removes it from the arc as a side effect.

Rule guards are not limited to single terms, but their syntax differs from general Boolean expressions. The property that is required is that a guard must remain TRUE once it becomes TRUE, at least until its node fires. We call this property "monotonicity." Hence a guard like "the incoming arc is empty" cannot be allowed. Our experience shows that extremely complex guards are almost never required in practice so guard syntax in the CODE 2.0 prototype is further constrained. The rule syntax is

$$E_1, E_2, ..., E_n$$

where "," should be read as AND. The E's are either general Boolean expressions in the node's local variables, or in the case of input guards they may be calls to the **avail** function. In addition, there is a mechanism for specifying indexed replications of rules and rule terms. An example of this is given in section 4.4.

After input binding is complete, the node's computation begins. Computations are expressed as subroutines in conventional languages such as C. When the computation returns, the output rule guards are evaluated. Binding sections for *all* enabled rules are executed. The notation "Y = v" means place the value bound to v onto the arc denoted by Y.

4.4 RUNTIME DETERMINED STRUCTURES

Very often, parallel algorithms are most readily expressed as graphs whose structure is input-driven. Hence, the structure is not known until runtime. Graph **Laplace** provides a simple example and will be discussed below. The block triangular solver program of section 7.2 provides a somewhat more complex example.

Graph **Laplace's** communication structure is not known until runtime because the number of **Comp** nodes is determined by an input parameter (**NProcs**) to the graph. The structure created at runtime is shown in figure 4.7. For simplicity, we will ignore all incoming arcs to **Comp** except that which carries the current iteration count. Call this arc **GoalIn**, according to the name of the port in **Comp** to which it is connected.

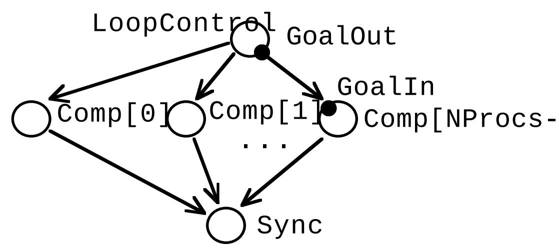


Figure 4.7 - Structure of **Comp** Family in **Laplace**

Comp is a family of nodes, and the "arcs" incident upon it represent a family of arcs. Such dynamic structures are specified in CODE 2.0 by an interaction of input rules, output rules, and *arc topology specifications* associated with arcs.

First, consider the **LoopControl** output rule that passes the current iteration count to the **Comp** nodes.

Goal > 0 => {GoalOut[i] = Goal : (i NProcs)}

GoalOut is **LoopControl**'s name for the port bound to the arc going to **Comp**, and **Goal** is the iteration count. The expression "(i NProcs)" is a replicator that causes the binding to be done for values of "i" from 0 to **NProcs**-1.

Arc topology specifications determine where data should be sent when placed onto an arc, given where it originates. The arc between **LoopControl** and **Comp** has the following specification. (Actually the node name is supplied by graphical context.)

LoopControl.GoalOut[i] => Comp[i].GoalIn

Hence, if a value is placed onto the port with name **LoopControl.GoalOut[7]**, then that value will be passed to port **GoalIn** of node **Comp[7]**. This causes node **Comp[7]** to be created if it is not already instantiated. In this manner, arbitrary communication topologies can be built at runtime.

5. PROGRAM COMPONENT REUSE

CODE graphs form a natural unit for program fragment reuse because they have clean and complete interface specifications. This has been explored with the ROPE system

[LEE89], which is coupled to earlier versions of CODE. We plan to incorporate an updated ROPE system into CODE 2.0 that will take advantage of its ability to package types with graphs.

6. FRAMEWORK FOR EFFECTIVE TRANSLATION

Abstract models of parallel computation are not useful unless they can be translated into efficient executable structures, preferably on multiple target architectures. CODE 2.0's implementation addresses this issue by raising the level of abstraction at which translators are defined. In particular, the CODE 2.0 model of computation is defined and implemented as a *class hierarchy*. Translators are defined as methods bound to the various classes in the hierarchy. The key aspect of this is that the translators associated with the various classes in the hierarchy are relatively decoupled. Thus, it is possible to alter a translator without drastic modifications to other translators. The classes are used more for their information hiding properties than for inheritance. Figure 6.1 shows the top few layers in the class hierarchy.

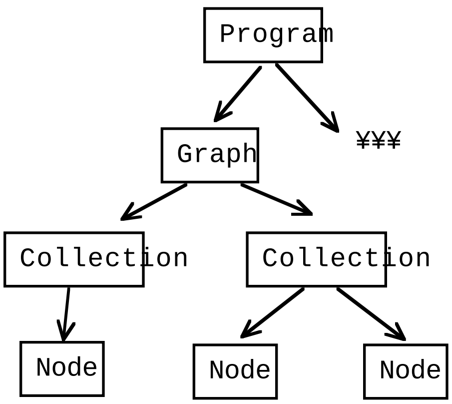


Figure 6.1 - CODE 2.0 Class Hierarchy

A program consists of a set of graphs, a graph is a set of collections, and so on.

The translation process uses the class hierarchy in a manner that is roughly analogous to code improvers for traditional languages like Fortran. The objective is to recognize and produce special optimized code for special case programming constructs. CODE 2.0's implementation permits classes to be defined for each special case. Optimized translation methods are then bound to the new classes. The process of adding classes to the hierarchy and defining new translation methods for them is called *translation refinement*. It involves preparing a new class and a new translation method and recompiling the CODE 2.0 system.

Of course, one needs to determine which special case constructs should be given optimized translation methods. Our experience with previous versions of CODE suggests that CODE programming is quite idiomatic, at least within the scientific problem domains we have studied chiefly. Hence, interesting special constructs are easy to find. Many of the most important are subgraphs (node collection classes). Figure 6.2 shows two examples.

Sequences of nodes for which pipeline parallelism is impossible or undesirable should be translated into a sequence

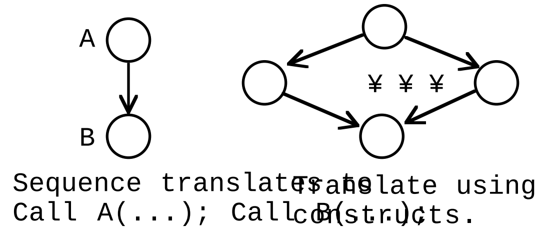


Figure 6.2 - Special Node Collections

of calls to their sequential computations. No intertask communication should be done. The common pattern of splitting data, computing, and joining can often be implemented with tailor-made barrier primitives.

Other special cases are more local in nature since they involve only a single node or arc. Some examples are listed below.

- Nodes with special firing rules-- especially pure dataflow rules in which values on all input arcs are needed for firing.
- Situations such as program SGRID in which values need not be copied on shared memory machines when passed from node to node.
- Situations in which no more than one value may appear on an arc so that buffering is not needed.

The buffering optimization is partially implemented in the prototype. We can examine its effect on the simple graph fragment shown in figure 6.3. Actual Ada code is given in Appendix A. A single arc is bound to port X of a single node. The node is enabled when there is a value on the arc.

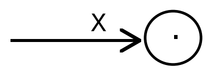


Figure 6.3 - A Simple Graph Fragment

If the buffering optimization is not applied, the CODE 2.0 prototype produces the Ada code shown by the pseudocode in figure 6.4. There is an Ada task associated with the node and the incoming arc. This task must do a rendezvous to accept an incoming value on

```
Node Task:
  select
    accept X do
      enqueue value;
    end;
    check input rules;
  or when rule is true =>
    accept StartComp;
  or accept EndComp;
          *
          *
  end;

Compute Task:
  StartComp;
  Call SeqComp
  EndComp;
```

Figure 6.4 - Unoptimized Ada Pseudocode for Fragment

the arc. It must then locally enqueue the value, and accept two rendezvous from a second task that performs the sequential computation. In total, the following resources are required.

- Two tasks.
- Node internal queueing.
- Three rendezvous.

However, if the system can assume that no more than one value will reside on the arc, much better code can be generated. This case is quite common because it is caused by loop recurrences. Figure 6.5 shows the Ada code that is produced.

```
Node Task:
     select
       accept X
       Call SeqComp;
           *
           *
     end;
```

Figure 6.5 - Optimized Ada Pseudocode for Fragment

Now, we use one task instead of two, have no internal queuing, and do one rendezvous instead of three. Since we have found that the rendezvous count is the most significant overhead on our system, this simple optimization has cut the node's overhead by a factor of three. Measurements showing the effectiveness of this are presented in the next section.

In general, experiment and measurement as well as our intuition will be used to identify candidates for optimization. CODE 2.0 translators will be able to insert instrumentation into the code they produce in order to measure its performance and find bottlenecks. It is expected that this instrumentation applied to a benchmark suite of programs will supply valuable insights.

7. EXAMPLES AND PERFORMANCE RESULTS

The process of finding optimizations begins with measuring and modeling (or in some way understanding the performance) of benchmark programs and evaluating the performance of the executable structures that CODE 2.0 generates. In order to do this, we develop three versions of each algorithm under study.

1. A sequential program (using a straightforward but high quality algorithm).
2. A hand-coded parallel program tailored to the target architecture.
3. A CODE 2.0 program.

Using these three programs, we can compute speedups relative to the sequential program and compare the CODE 2.0 program with a hand-coded parallel program. This latter comparison yields the overhead of the CODE 2.0 abstractions. We have performed such experiments on several small programs using the CODE 2.0 prototype. The results are distinctly encouraging, although it must be admitted that the benchmark problems are simple and that Ada is a reasonably easy target due to the high execution time cost of the rendezvous primitive. One noteworthy result of the prototype is that it produces code that is at least as efficient as that produced by earlier versions of CODE despite a much more dynamic model. The production version of CODE 2.0 will be C oriented, will utilize many more optimizations, and will be used for larger experiments.

7.1 SGRID (LAPLACE PROGRAM)

Performance results have been obtained for the Laplace grid example described in section 4.1. Since the CODE 2.0 prototype produces Ada, both the sequential grid program and the hand-coded parallel version were also written in Ada. All programs were run on a Sequent Symmetry shared memory MIMD machine and timed with the UNIX "time" facility. The grid size is 100 x 100. Initial results for NProcs = 7 showed that the hand-coded parallel version has a speed up of 5.2 as compared with 4.5 for CODE 2.0, a 16% difference. Measurement indicated that rendezvous on the Symmetry are quite expensive and that performance could be dramatically improved by reducing the rendezvous count. Applying the buffering optimization accomplishes this, so translation methods for it were implemented. The optimized CODE 2.0 program ran with a speed up of 5.1, a 2% percent difference. Speed ups for the three programs are shown in figure 7.1.

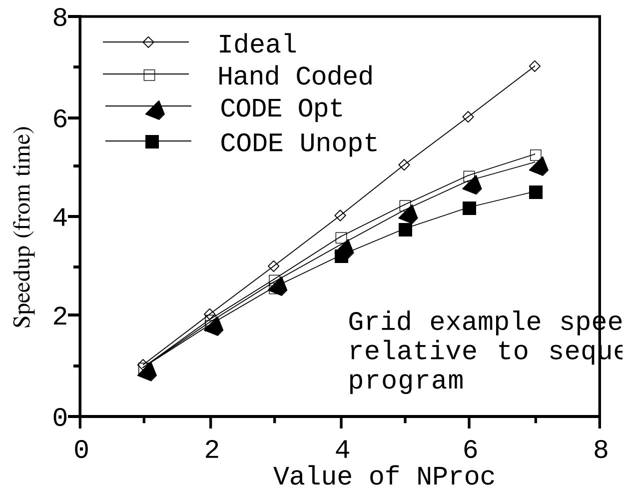


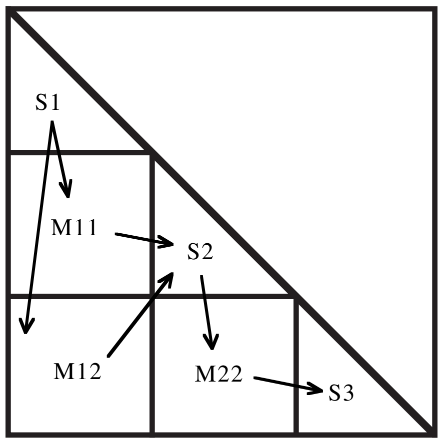
Figure 7.1 - Grid Program Performance

Remaining performance bottlenecks for both the hand-coded and CODE versions include the high cost of the rendezvous operation and some I/O done by our particular Ada implementation at program termination time. If this latter factor is discounted, 7 processor speedups are

Hand-Coded: 5.8
CODE 2.0 Optimized: 5.60
CODE 2.0 Unoptimized: 4.9

7.2 BLOCK TRIANGULAR SOLVER

This section contains an example program that solves the Ax = b linear algebra problem for a known lower triangular matrix A and vector b. The parallel algorithm is quite simple and is due to Jack Dongarra and Danny Sorenson. It involves dividing the matrix into blocks as shown in figure 7.2. The inputs to the algorithm are A, b, the size of the system, and the size of the block system used to partition it. In figure 7.2, the S's represent lower triangular submatrices that are solved sequentially, and the M's represent submatrices that must be multiplied by a vector. The arcs represent the dependences between these operations. Let the block system be of size sb x sb and note that the vector multiplications for all M's within a column may be done in parallel. This parallelism yields an ideal speedup of sb/3, for large values of sb.



- Lower triangular matrix partitioned into a 3 x 3 block system.
- S's are small lower triangular systems that are solved sequentially.
- M's are blocks that must be multiplied by a vector.
- Arrows show flow of the CODE 2.0 program. Note that M11 and M22 multiplications may be performed in parallel.

Figure 7.2 - Parallel Block Triangular Solver

The CODE 2.0 program for this example consists of three graphs, the main graph, a graph that defines A and b, and a graph that solves the system. We will concentrate solely on the latter. A, b, and x are passed to the graph as creation parameters. (It would be better to use a name sharing relation, at least for A, but they are not implemented in the CODE 2.0 prototype.) Figure 7.3 shows the graph.

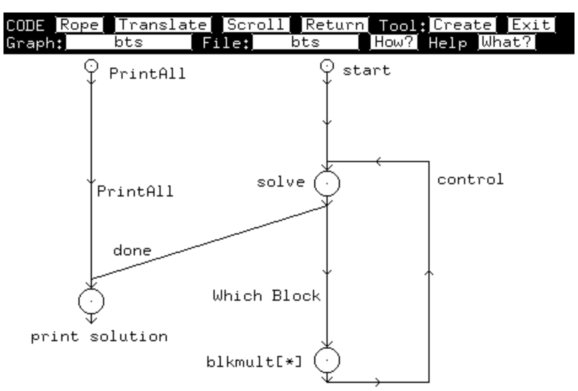


Figure 7.3 - Block Triangular Solver Graph

In this graph, a single instance of node **solve** performs all of the S operations, one after another, and sb - 1 instances of node **blkmult** perform the M's. Hence, the arc **control** implies an iteration. First S_1 is done and then $M_{1,j}$ for j = 1..sb-1. Then S_2 is done followed by $M_{1,j}$ for j = 2..sb-1, and so on. Since the block system size is an input to the program, the number of **blkmult** nodes to create is not determined until runtime. In addition, sb determines the number of times each node fires so this is also not known until runtime.

Figure 7.4 shows the ideal speedup and the speedups of the CODE 2.0 and hand-coded parallel program for a matrix of size 420 x 420.

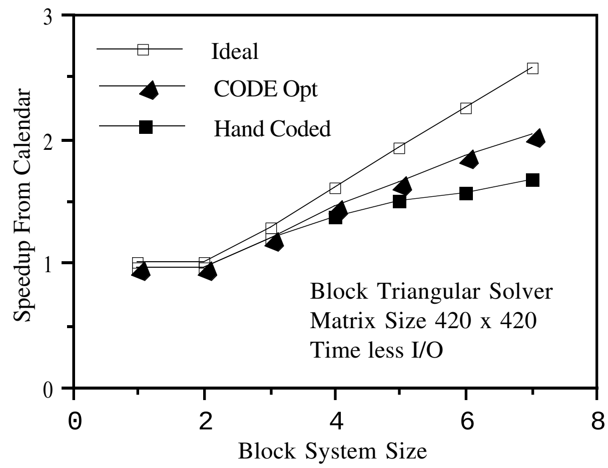


Figure 7.4 - Block Solver Speedups

The CODE 2.0 program is actually faster than the hand-coded program which was done by a graduate student (unrelated to the CODE 2.0 project) at the University of Texas at Austin. Timings used to compute these speedups include only the solving of the system, not I/O. (Due to the slow speed of naive I/O in Ada on our system, this problem is I/O bound.) The programs were run on a Sequent Symmetry.

The relatively slow speed of the hand parallelized program may be surprising. Of course, it is the case that this program is not optimally written, at least not as far as performance is concerned. It is a subjective point, but the hand written program is elegant and stylistically natural given the Ada model. Perhaps abstract parallel programming can sometimes produce faster code than low-level parallel programming just as sequential programs written in high-level languages sometimes run faster than assembly language programs-- because they are better structured at a high level.

8. CODE PROJECT HISTORY

The CODE project has been underway at the University of Texas at Austin for several years, and CODE 2.0 is based on lessons learned from much previous work-- in particular the CODE 1.2 system [WER90]. The fundamental intellectual basis of CODE 2.0, the use of a declarative graph-based abstract model of computation, is directly inherited from previous CODE systems as is the concept of graphical component reuse [LEE89].

However, CODE 2.0 expands on previous CODE systems in many ways. It's model has been completely rethought to place greater emphasis on runtime determined structures, program modularity, and a wide set of data types. This significantly expands the class of algorithms that can be effectively and efficiently expressed under the CODE model. In addition, CODE 2.0 has been implemented from scratch using more modern tools such as the GUIDE user interface toolkit. Also, the CODE 2.0 project places greater emphasis on execution efficiency. Key features of the new CODE 2.0 model include the following.

- Runtime determination of structure via template instantiation.
- User supplied Input and Output rules.
- Hierarchical structuring via Call nodes and runtime instantiated graphs.
- Clean name scoping and improved type system.

9. RELATED PROJECTS

There are several other graphical parallel programming systems in various stages of development. Each of these systems tends to have its own particular strengths and goals. We mention a few that are relatively closely related to CODE 2.0. CODE 2.0's particular strengths include a powerful node firing condition syntax, support for graphs with runtime determined structure, and a framework for the creation of optimized parallel code generators.

One of the more interesting and current systems is HeNCE [BEG91a]. This system permits programs represented as graphs to be run intelligently on a heterogeneous network of computers using the PVM system [SUN90]. HeNCE has an integrated tool for tracing and analyzing a program's execution. However, CODE 2.0's node firing conditions are more flexible and it is able to realize more complex graphs with runtime determined structure-- arbitrary cyclic and recursive graphs with fixed node types. In addition, CODE 2.0 supports hierarchical structuring by means of the graph Call node.

Phred [BEG91b] is related to HeNCE but stresses determinacy and makes use of separate control and dataflow graphs to define a computation. The Paralex system [BAB92] is less expressive than CODE 2.0 or HeNCE but stresses fault tolerance and dynamic load balancing. The PPSE system [LEW90] places emphasis on providing full lifecycle support for parallel program development. Its definition includes tools for parallelizing sequential Fortran programs, graph editing, program scheduling, code generation, performance analysis, and debugging. The GILT system by Roberts and Samwell of City University, London, U.K. is oriented towards Occam programming, and GRAPE [LAU90] has a digital signal processing orientation (while still useful for general purpose programming) and is the target of a debugger effort. The ParaGraph system [BAI91] uses graph grammars to specify graphs that have runtime determined structure. The emphasis is on massively parallel systems.

In addition, W. Mayne [MAY92] is exploring a system with a more expressive mechanism for describing node firing conditions than that used by CODE 2.0. This system permits guard priorities, for example. The system is also designed to facilitate proofs of determinacy. There is no implementation at this time.

10. CONCLUSIONS AND FUTURE WORK

The CODE 2.0 parallel programming system has three goals: ease of use, portability, and production of efficient executable structures. A number of technologies are exploited in order to achieve these goals. A graphical/textual user interface with hypertext online help and integrated support for program component reuse promote ease of use. Portability is approached by means of an abstract model of parallel computation that cleanly isolates program communication and synchronization structure from primitive sequential computations. In addition, the high level of abstraction is appropriate to the programming process and, hence, supports ease of use.

Execution efficiency is approached through a hierarchical object-based model representation basis that supports the concept of translation refinement. Ubiquitous special case programming structures with significant performance implications may be singled out and given custom translation methods. An experimental approach is used to identify such special cases. The use of object semantics and translation method inheritance assists in the software engineering of modules that translate instances in the abstract model into executable structures.

A Prototype of CODE 2.0 is running and has produced encouraging results for a relatively easy (from a performance point of view) target architecture, Ada on a shared memory MIMD machine. Future plans include the creation of a production version of CODE 2.0 that will support multiple MIMD shared and partitioned memory target architectures. In addition the ROPE graph reusability system will be coupled with CODE 2.0. It is expected that the flexibility of the CODE 2.0 model will promote graph reuse.

APPENDIX A - ADA CODE FOR EXAMPLE

--- Ada Code without the buffer optimzation

```
task body MAIN2 is    -- UC
      *
      *
      *
   procedure CHECKFIRERULES is
     YYEMPTY : BOOLEAN;
   begin
     RULETRUE := FALSE;
     YYEMPTY := TRUE;
     if not (X_ILIST.SAT(
       NM(0, 0, 0, 0, 0, 0, 0, 0))) then
      goto BEGRULE9;
     end if;
     YYEMPTY := FALSE;
     if YYEMPTY then
      goto BEGRULE9;
     end if;
     RULETRUE := TRUE;
     VARSLOCKED := TRUE;
     V := X_ILIST.VAL(
            NM(0, 0, 0, 0, 0, 0, 0, 0));
     return;

   <<BEGRULE9>>
     null;
     return;
   end CHECKFIRERULES;

   -- NOTE: This is a Task
   task body COMPUTE is
      *
      *
      *
   begin
     accept INIT;          -- Sync non-locals
     loop
```

```
      MYADDR.STARTFIRE(KILLED); -- Extra Rend.
      exit when KILLED;
      null;
      MAININST.KILL;
      MYADDR.ENDFIRE;  -- Extra Rend.
    end loop;
  end COMPUTE;

begin
  accept INIT(THEINDS     : in SNAME;
              THEADDR     : in MAIN2_PTR;
              THESERVADDR : in GENERIC_PTR) do
    NODE := THEINDS;
    MYADDR := THEADDR;
    MYADDRSERVER := THESERVADDR;
  end INIT;
  loop
    if not VARSLOCKED then
     CHECKFIRERULES;
    end if;
    select
     terminate;
    or
     accept KILL;
     KILLED := TRUE;    -- tell it to compute
    or
     when RULETRUE or KILLED =>
       accept STARTFIRE(KVAL : out BOOLEAN)
       do
         KVAL := KILLED;
       end STARTFIRE;
    or
     accept ENDFIRE;
     VARSLOCKED := FALSE;
    or
     accept CREPARAMS do
       null;
     end CREPARAMS;
     null;
     COMPUTE.INIT;
     VARSLOCKED := FALSE;
    or
     accept X(VAL : in INTEGER;
              THENAME : in SNAME) do
       X_ILIST.INSERT(THENAME, VAL);
     end X;
    end select;
  end loop;
end MAIN2;

-- Ada Code with the buffer optimization

task body MAIN2 is    -- UC
     *
     *
     *
  procedure CHECKFIRERULES is
    YYEMPTY : BOOLEAN;
  begin
    RULETRUE := FALSE;
    YYEMPTY := TRUE;
    if not (X_ILIST.SAT(
        NM(0, 0, 0, 0, 0, 0, 0, 0))) then
     goto BEGRULE0;
    end if;
    YYEMPTY := FALSE;
    if YYEMPTY then
     goto BEGRULE0;
    end if;
    RULETRUE := TRUE;
    VARSLOCKED := TRUE;
    V := X_ILIST.VAL(
              NM(0, 0, 0, 0, 0, 0, 0, 0));
    return;

  <<BEGRULE0>>
    null;
    return;
  end CHECKFIRERULES;

 -- NOTE: This is not a task.
 procedure COMPUTE is
     *
     *
     *
  begin
    null;
    MAININST.KILL;
    return;
  end COMPUTE;

begin
  accept INIT(THEINDS     : in SNAME;
              THEADDR     : in MAIN2_PTR;
              THESERVADDR : in GENERIC_PTR) do
    NODE := THEINDS;
    MYADDR := THEADDR;
    MYADDRSERVER := THESERVADDR;
  end INIT;
  loop
    if not VARSLOCKED then
     CHECKFIRERULES;
    end if;
    if VARSLOCKED and RULETRUE then
     COMPUTE;
     VARSLOCKED := FALSE;
    end if;
    select
     terminate;
    or
     accept KILL;
     KILLED := TRUE;    -- tell it to compute
    or
     accept CREPARAMS do
       null;
     end CREPARAMS;
     null;
     VARSLOCKED := FALSE;
    or
     accept X(VAL : in INTEGER;
              THENAME : in SNAME) do
       X_ILIST.INSERT(THENAME, VAL);
     end X;
    end select;
  end loop;
end MAIN2;
```

REFERENCES

[BAB92] Ö. Babaoglu, *Paralex: An Environment for Parallel Programming in Distributed Systems*, to appear in Proc. ACM Int. Conf. on Supercomputing, July, 1992.

[BAI91] D.A. Bailey, et al., *ParaGraph: Graph Editor Support for Parallel Programming Environments,* International Journal of Parallel Programming, Apr., 1991.

[BEG91a] A. Beguelin, et al., *Graphical Development Tools for Network-Based Concurremt Supercomputing*, Proc. Supercomputing '91, Albuquerque, NM, pp. 435-444, 1991.

[BEG91b] A. Beguelin and G. Nutt, *Collected Papers on Phred,* Dept. of Computer Science, Univ. of Colorado, CU-CS-511-91, Jan., 1991.

[BRO85] J.C. Browne, *Formulation and Programming of Parallel Computers: a Unified Approach,* Proc. Intl. Conf. Par. Proc., 1985, pp. 624-631.

[BRO89] J.C. Browne, M. Azam, and S. Sobek, *CODE: A Unified Approach to Parallel Programming,* IEEE Software, July, 1989, p. 11.

[BRO90] J.C. Browne, J. Werth, and T.J. Lee, *Experimental Evaluation of a Reusability Oriented Parallel Programming Environment,* IEEE Trans. Soft. Engin., Vol. 16, No. 2, 1990.

[EIG91] R. Eigenmann, and W. Blume, *An Effectiveness Study of Parallelizing Compiler Techniques*, Proc. Intl. Conf. Par. Proc., 1991, pp. II 17-25.

[HIR91] S. Hiranandani, K. Kennedy, and C.-W. Tseng, *Compiler Support for Machine-Independent Parallel Programming in Fortran D*, Rice University, CRPC-TR91132, 1991.

[JAI91] R. Jain, J. Werth, and J.C. Browne, *An Experimental Study of the Effectiveness of High Level Parallel Programming*, Proc. 5th SIAM Conf. Par. Processing, 1991.

[LAU90] R. Lauwereins, et al., *GRAPE: A CASE Tool for Digitial Signal Parallel Processing,* IEEE ASSP Magazine, Apr. 1990.

[LEE89] T.J. Lee, *Software Reuse in Parallel Programming Environments*, PhD thesis, University of Texas at Austin, Dept. of Comp. Sci., 1989.

[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.

[LUS87] E. Lusk, R. Overbeek, et al., Portable Programs for Parallel Processors, Holt, Rinehart, and Winston, New York, 1987.

[MAY92] W. Mayne, Florida State University, personal communication, Apr. 1992.

[NEW91] P. Newton, *CODE 2.0 Prototype*, unpublished internal documentation, University of Texas at Austin, July 16, 1991.

[SOB90] S. Sobek, *A Constructive Unified Model of Parallel Computation*, PhD thesis, University of Texas at Austin, Dept. of Comp. Sci., 1990.

[SUN91] V.S. Sunderam, *PVM: A Framework for Parallel Distributed Computing*, Concurrency: Practice and Experience, 2(4):315-339, Dec., 1990.

[WER90] J. Werth, et al., *CODE 1.2 User Manual and Tutorials,* Tech. Report TR-90-35, Univ. of Texas at Austin, Dept. of Comp. Sci., Nov., 1990.

[WER91] J. Werth, et al., *The Interaction of the Formal and Practical in Parallel Programming Environment Development: CODE,* Tech. Report TR-91-09, Univ. of Texas at Austin, Dept. of Comp. Sci., 1991.