Visual Programming and Parallel Computing

James C. Browne [†]
Jack Dongarra [††]
Syed I. Hyder [†]
Keith Moore [††]
Peter Newton [††]

Abstract

Visual programming arguably provides greater benefit in explicit parallel programming, particularly coarse grain MIMD programming, than in sequential programming. Explicitly parallel programs are multi-dimensional objects; the natural representations of a parallel program are annotated directed graphs: data flow graphs, control flow graphs, etc. where the nodes of the graphs are sequential computations. The execution of parallel programs is a directed graph of instances of sequential computations. A visually based (directed graph) representation of parallel programs is thus more natural than a pure text string language where multi-dimensional structures must be implicitly defined. The naturalness of the annotated directed graph representation of parallel programs enables methods for programming and debugging which are qualitatively different and arguably superior to the conventional practice based on pure text string languages. Annotation of the graphs is a critical element of a practical visual programming system; text is still the best way to represent many aspects of programs.

This paper presents a model of parallel programming and a model of execution for parallel programs which are the conceptual framework for a complete visual programming environment including capture of parallel structure, compilation and behavior analysis (performance and debugging). Two visually-oriented parallel programming systems, CODE 2.0 and HeNCE, each based on a variant of the model of programming, will be used to illustrate the concepts. The benefits of visually-oriented realizations of these models for program structure capture, software component reuse, performance analysis and debugging will be explored and hopefully demonstrated by examples in these representations. It is only by actually implementing and using visual parallel programming languages that we have been able to fully evaluate their merits.

1.0 Introduction

During the past 15 years microprocessor performance has improved dramatically in comparison to the performance of larger systems [Pat90]. From a hardware point of view, this trend has made parallel computers increasingly attractive since high-performance machines can be built by combining large numbers of microprocessors that have been bought at commodity prices. The design details vary greatly from one machine to another, but most recent machines adopt the MIMD (multiple instruction streams - multiple data streams) model in which each processor can perform different computations on different data. Some machines use a shared address space for memory; others require that processors communicate via explicit message sending. It is even possible, since they are often

† University of Texas at Austin.
†† University of Tennessee at Knoxville.

available, to use a network of workstations as a parallel computer. All of these designs are intended for coarse-grain computations in which processors execute a substantial number of instructions between communications or other interactions with other processors. If the computation grain becomes too small, performance suffers. This paper will focus exclusively on visual programming methods for coarse-grain MIMD parallel architectures.

The primary reason that parallel computing is not more common than it is today is that, while the machines are fairly easy to build, it is quite difficult to write programs which are both efficient and portable across machines since the design details of parallel machines impact both the programming model and execution performance far more significantly than do the details of the designs of sequential machines. The difficulty of programming parallel machines is the major bottleneck preventing their wider acceptance.

It is easy to see that parallel programming is more difficult than sequential programming since sequential programs are simply a degenerate case of parallel programs. Coarse-grained MIMD parallel programs consist of interacting sequential elements. The programmer must specify both the sequential elements and their interactions.

A model of programming in which parallel programs are created by first defining a set of sequential units of computation and then composing them into a parallel program addresses this complexity issue by a divide and conquer (or separation of concerns) approach since the two steps are done separately. Directed graphs are a very natural mechanism for the composition step. Nodes represent atomic sequential computations, and arcs represent dependencies between them. The nature of the dependencies can vary from model to model as we shall see.

Parallel programs written in the directed graph model are also intrinsically more portable across architectures since the interactions among the sequential units of computation are expressed in the structure of the graph independently of the mechanisms in which they will ultimately be realized. The separation of concerns which assists in reduction of complexity of programming also results in reduction of the complexity of compilation of these abstract specifications for interactions into efficient executable forms. As we shall see later, the two systems used as examples in this paper, HeNCE [Beg91a] and CODE 2.0 [New93, New92], demonstrate that in at least some circumstances, competitively efficient code can be generated from the abstract specifications of interactions.

This separation of concerns also leads naturally to the reuse of components since the sequential computations from which the parallel computations are composed are defined in a precisely specified data and control context and must have clean and precise interfaces and well-understood semantics.

Parallel programming also differs from sequential programming in that programmers must understand the large-scale structure of their programs in order to understand their execution performance. This is a vital issue since performance is the major reason for the existence of parallel computing. Programmers must know what elements of their parallel program are scheduled for execution and which communicate with which, and they must have a grasp of the granularity (or size) of the computation that takes place within a sequential element between communications. Furthermore, programmers often must understand how their computations are mapped onto the processors of a parallel machine (which can also be represented as a graph).

Graphical tools are widely used to display information about execution behavior, but directed graph based visual parallel programming languages have a special advantage. The execution data can be directly related to the user's original program since they share a common graphical format. This integrates the steps of program creation and debugging, both for performance and correctness.

1.1 Conventional Approaches in Parallel Programming Languages

Many programming language and compiler approaches have been proposed to simplify programming parallel machines, but none have been completely successful. It is useful to review them before moving on to the virtues of visual parallel programming.

- 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, although there has been progress in defining standard libraries for some important and broad classes of machines.

- Have compilers automatically detect parallelism in sequential language programs.

The parallelism in a program is implicit and must be discovered and exploited by the compiler. 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]. This approach also suffers from the fact that, in practice, programmers must be aware of the parallel structures the compiler will produce from given source text since they must program idiomatically so that the compiler will be able to produce efficient code. In this sense, the parallelism is not implicit at all. It is merely expressed indirectly.

- Extend sequential languages to allow data partitions 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. It is unclear how effectively complex data structures such as unbalanced trees can be partitioned, either at compile time or at runtime.

1.2 Visual Parallel Programming Languages

Graphical displays are useful and common aspects of parallel programming environments, but they tend to be limited to displaying the performance, behavior, or structure of parallel programs that are expressed conventionally, as text. This paper argues that significant benefits can be obtained by going a step further and directly expressing parallel programs visually. The concept of visual directed graph programming systems is not new. The first significant system was probably that of Keller and Yen [Kel81] in 1981. It is, however, only in recent years that a significant impact from visual directed graph parallel programming languages has been obtained. The advantages of this approach will be discussed both in the abstract and specifically in terms of two implemented visual parallel programming languages, HeNCE 2.0 and CODE 2.0.

These two languages differ in many ways but both rest upon the notion that parallel programs can usefully be represented as directed graphs in which nodes with certain icons represent sequential computations and the graph as a whole represents the parallel structure of the program. Each graph shows, in some fashion, what sequential computations in the parallel program can be run concurrently with what other sequential computations. There are many advantages to this view.

1. Graphs are a more natural representation for parallel programs than linear text because parallel program behavior is inherently multi-dimensional.
2. A graph-based visual parallel programming language can separate the programming process into two distinct concerns, creating sequential program elements and composing them into a complete parallel program thus facilitating a divide and conquer approach to design.
3. Graphs directly display and expose large scale program structure that programmers must understand in order to achieve good performance.
4. Visual representation promotes the exploitation of data locality, another key to parallel program performance.
5. A graph model can permit logical and performance debugging to be carried out in the same framework as programming. Tools to support these tasks integrate neatly into a single visual framework.

These advantages will be elaborated in the sections that follow.

2.0 Parallel Programs Are Graphs

Representations of parallel programs and parallel program behaviors are naturally multi-dimensional. This structure, for both the program and its executions, is effectively captured by directed graphs. This suggests directed graphs as a means of representing parallel programs since they will better permit programmers to relate programs to their behavior.

The source of this non-linearity is that MIMD Parallel programs, regardless of how they are expressed, consist of multiple interacting threads of control. Two examples will demonstrate.

2.1 Direct Representation of Implicit Parallelism

Consider the sequence of assignment statements shown in the program in Figure 1. They have an obvious interpretation as a sequential program and imply the execution sequence: 1-2-3-4. This is clearly a linear representation and remains so even in the presence of multiple control flow paths since only one is taken.

```
/* step 1 */    x = 5;
/* step 2 */    y = 3;
/* step 3 */    z = x + 2;
/* step 4 */    w = x + y + z;
```

Figure 1. Example Program.

This program can also be viewed as a parallel program since some of the steps are independent since they access no common variables. For example steps 1 and 2 can be executed in parallel or in either order. Hence, the program's execution is now longer a simple sequence. Computations such as the following can all be valid interpretations of the parallel program, although not all exploit maximal parallelism. The notation (1,2) means that steps 1 and 2 are performed in parallel.

```
1-2-3-4                     2-1-3-4
1-3-2-4                     (1,2)-3-4
```

Listing all of the possible computations is a cumbersome way of understanding this program. For example, step 2 can also run in parallel with step 3 as long as the latter is done after step 1.

However, notice that a computation graph such as that shown in Figure 2 neatly summarizes the program's behavior. The nodes represent steps. The arcs in this diagram show data flow. Two nodes may be run in parallel if there is no path from either to the other.

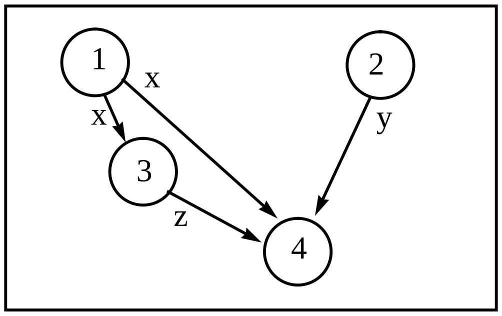
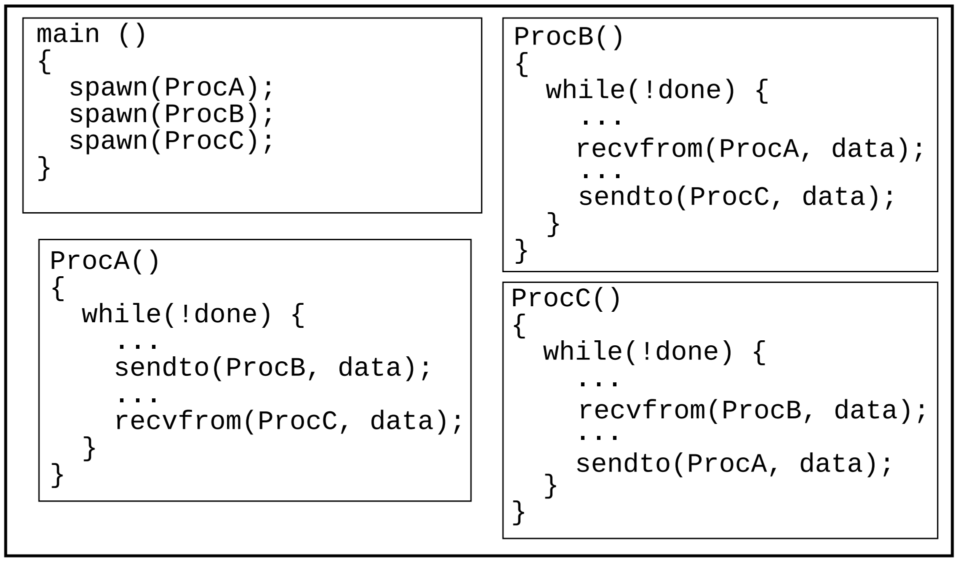


Figure 2. Computation Graph of Example Program.

2.2 Message Passing Example

Of course, parallelism at the statement level is inappropriate for machines that support only coarse-grain computation. For them, nodes must represent larger computations.

The above example suggests that parallelism, implicit in conventional sequential program representations, has a natural representation as a directed graph. This is true also of representations that show parallelism directly. Consider programs expressed in “C” with calls to explicit message passing libraries in the general style of the PVM system [Gei93]. An example is shown in Figure 3.



```
main ()
{
  spawn(ProcA);
  spawn(ProcB);
  spawn(ProcC);
}
```

```
ProcA()
{
  while(!done) {
    ...
    sendto(ProcB, data);
    ...
    recvfrom(ProcC, data);
  }
}
```

```
ProcB()
{
  while(!done) {
    ...
    recvfrom(ProcA, data);
    ...
    sendto(ProcC, data);
  }
}
```

```
ProcC()
{
  while(!done) {
    ...
    recvfrom(ProcB, data);
    ...
    sendto(ProcA, data);
  }
}
```

Figure 3. Example Message-Passing Program.

Graphical display tools often represent the behavior of such programs by means of a diagram that shows messages being sent from one process to another. In other words, every interaction between processes is shown by an arc. Figure 4 shows such as diagram and how it can be interpreted as a computation graph by identifying each segment of sequential processing between communications as a node.

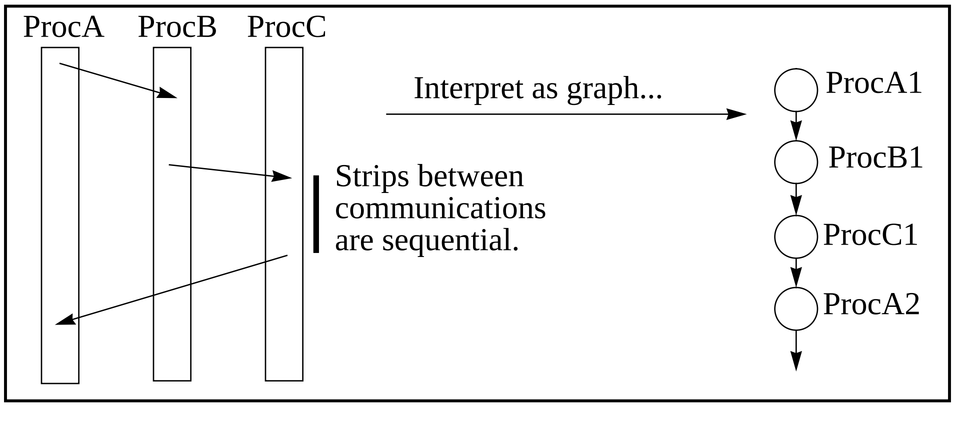


Figure 4. Message-Passing Program and It’s Computation Graph.

3.0 Visual Parallel Programming

If directed graphs are a natural mechanism for displaying the behavior of parallel programs, then why not use them as a basis for a programming language in order to reduce the distance between representation and behavior? There are many ways to go about this, but we will assume that programs are represented by directed graphs in which nodes with specific icons represent sequential computations (other icons may represent other constructs) and the graph in some fashion represents the overall parallel structure.

3.1 Two Steps in Programming

One immediate advantage of this view is that the process of creating a parallel program can be divided into two distinct steps: creation of components and the composition of these components into a graph. The primitive components can be sequential computations but other cases are allowed. For example, a component could be a call to another graph that specifies a parallel sub-computation. In any case, components can either be created from scratch for a particular program or they can be obtained from libraries. The key is that each component simply maps some inputs to some outputs with a clean and clearly defined interface. These components can then be composed into a graph which shows which components can run in parallel with which other components.

Component creation and component composition are distinct operations. Programmers need not think about the details of one while performing the other (except to ensure that the sequential routines are, in fact, defined with clean interfaces and well-specified input/output semantics). In particular the specification of parallel structure is done without concern about the inner workings of the components involved. Furthermore, the best tools available can be used for the different tasks.

3.2 Sequential Components

Both HeNCE and CODE emphasize the use of sequential subroutines expressed in C or Fortran for use as primitive components– in fact HeNCE requires it. There are several benefits from this decision.

1. Implementation is facilitated since we build on the existing tool base of tested and accepted sequential languages and compilers.
2. This approach permits subroutines from existing sequential programs to be incorporated into new parallel programs. Leveraging existing code is often vital to the acceptance of new tools.
3. The learning curve for users is less steep since they are not asked to relearn sequential programming when adopting a parallel programming environment.

3.3 Parallel Composition into Directed Graphs

It is common for programmers to draw informal diagrams that show large scale parallel structure when designing parallel programs. The purpose of these diagrams is to abstract away the details of the components of the system being designed and concentrate on their interactions. A graph-based visual parallel programming language can help to formalize this process.

Understanding the large scale structure of parallel programs tends to be of greater importance than it is in the sequential case due to the fact that large scale structure can have a dramatic impact on the execution performance of parallel programs. In order for programmers to achieve and understand program performance, they must understand the structure of the computation graph of their program– regardless of how their program is represented. Consider the computation of Section 2.2. If the execution time of the sequential segments between communications is too short, performance will suffer since it will be dominated by the overhead of message passing.

A direct graphical representation of parallel programs renders such concerns explicit. The programmer knows exactly what the sequential components are precisely because they are separate components. Especially if they are subprograms that perform some cleanly defined function, the programmer will also have a good feel for their execution time. Hence, he or she will be aware of the computation's granularity.

The graph can also directly display other information that is vital to understanding the performance of any parallel program. Issues such as poor load balance or inadequate degrees of parallelism are apparent from the shape of the graph and the execution times of the nodes, interpreted relative to communication overheads. Figure 5 shows two examples.

A graphical representation is also useful because it can promote locality in designs to the extent to which components are in different name spaces in the language. In CODE, state is retained from one execution of a node to another, and communications must be explicitly defined as part of the interface to a sequential computation node. This encourages programmers to try to package a node's data with the node. Locality is easy to express, but remote access requires more effort. Thus, beginning parallel programmers are guided towards designs that exploit good data locality.

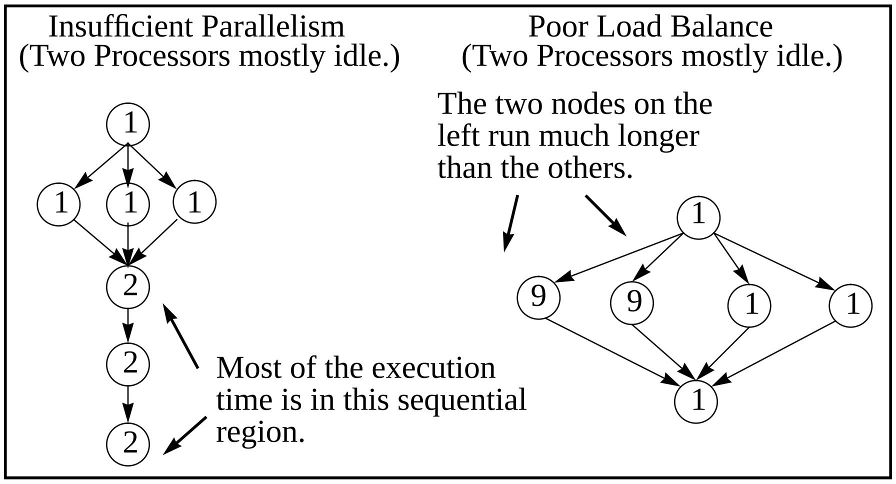

Insufficient Parallelism
(Two Processors mostly idle.)
Poor Load Balance
(Two Processors mostly idle.)
The two nodes on the left run much longer than the others.
Most of the execution time is in this sequential region.
1
1
1
1
2
2
2
1
9
9
1
1
1

Figure 5. Graphs Showing Poor Performance. (Runtimes shown in nodes.)

4.0 Compilers and Atomic Component Graph Models

Graph based models that are based on the composition of atomic components have advantages for compilation as well as for programmers. Directed graph representations abstractly express parallel structure and so are not tied to a single machine type. Portability is enhanced. Nodes are atomic mappings from inputs to outputs and can run on any type of machine. In fact, there is no reason to assume that all nodes must execute on the same type of processor. For example, HeNCE programs run on a potentially heterogeneous collection of UNIX workstations.

Compiling

Since the parallelism in the graph model is explicit, a compiler does not have to discover it; it must only exploit it. Furthermore, in CODE and HeNCE the granularity of components will likely be fairly high since they are based on calls to sequential subprograms. This reduces the difficulty of assigning tasks of appropriate granularity to processors.

The fact that components receive input, run to completion, and then send outputs also helps to control granularity and promotes language implementations that batch messages that are to be sent to the same destination. For an example, consider the following code fragment.

```
sendto(ProcA, data1);
some_short_computation();
sendto(ProcA, data2);
```

It is often better to combine the two sends into one. This is also true when sending to two different processes that have been assigned to the same physical processor.

Scheduling

The simpler incarnations of such graph models also lend themselves to the use of advanced scheduling techniques [Yan91] since the components are often arranged into directed acyclic graphs (or directed acyclic subgraphs can be found) and the execution times of components is often fixed from invocation to invocation. Furthermore execution characteristic of sequential elements are easier to define and measure since they are encapsulated. This encapsulation can also simplify dynamic (runtime) scheduling for load balancing since the state of sequential elements is fixed between executions.

The graph model also lends itself to implementation in heterogeneous parallel environments in which processing elements have varying speeds and capabilities. This is a more complex case of the scheduling problem just mentioned since characteristics of processors vary as well as characteristics of nodes. The HeNCE system is targeted towards heterogeneous environments.

Fault-tolerance tends to be simpler to implement in models in which components do not retain state from execution to execution. This factor will be most important when using a large network of independent workstations as the parallel machine to perform large computations.

5.0 CODE and HeNCE

CODE and HeNCE are implemented visual parallel programming languages that rest upon the ideas described above. They are very similar in purpose and general philosophy but are significantly different in detail. This section will summarize the languages and then provide an example of a program expressed in each.

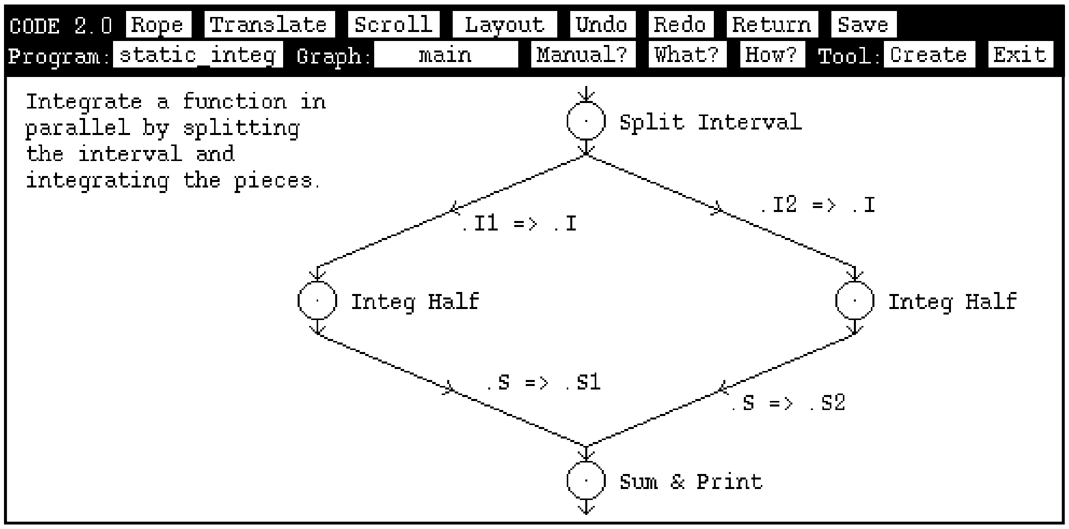
Both languages are alike in that users create a parallel program by drawing and then annotating a directed graph that shows the structure of the parallel program. Both languages offer several different node types, each with its own icon and purpose. In both cases, the fundamental node type is the sequential computation node which is represented by a circle icon. The graph annotations include sequential subroutines that define the computation that computation nodes will perform as well as specification of what data computations will act upon.

5.1 An Introductory Example: CODE

Figure 6 shows an extremely simple CODE program that will serve as an introductory example. It numerically integrates a function in parallel over a definite interval [**a**, **b**] by computing the midpoint **m** between **a** and **b** and then having one sequential computation node integrate the interval [**a**, **m**] while the other does [**m**, **b**] at the same time. The results are summed to form the final result.

The nodes in the graph that do the integration are both named **Integ Half** and a glance shows that they can run in parallel since there is no path from one to the other. The arcs in this CODE graph represent dataflow from one node to another on FIFO queues. The graph is read from top to bottom, following the arrows on arcs. Thus, the graph shows that node **Split Interval** is creating some data that are passed to the two **Integ Half** nodes. This data consists of a structure defining the integration the receiving node is to perform.

```
type IntegInfo is struct {
    real a;    // Start of interval.
    real b;    // End of interval.
    int n;     // Number of points to evaluate
};
```



```
Integrate a function in
parallel by splitting
the interval and
integrating the pieces.
```

Figure 6. CODE Integration Program

So, to create this parallel program, the programmer draws with the mouse a graph just as shown in Figure 6 and then enters textual annotations into different pop-up windows associated with various objects such as nodes, arcs, etc. This information includes such familiar items as type definitions and sequential function prototypes (for type checking calls). We will ignore these and focus on the annotations of computation nodes. When annotation is complete, the user picks “translate” from a menu, and a parallel program is created, complete with a Makefile, ready to be built and run on the selected parallel machine.

The annotation for a computation node consists mostly of a sequence of stanzas, some of which are optional. The annotation for the **Integ Half** nodes follows. Both nodes are identical. We will see later how a single replicated node could have been used in place of the two identical nodes.

```
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); }
routing_rules {
    TRUE => S <- val; }
```

The first two stanzas provide names for “ports” which are queues of data that enter and leave the node. Each node uses its own local names for these ports so that nodes can be reused in new contexts. This node will read data of type **IntegInfo** (the structure defined above) from a port called **I** and write real data onto a port called **S**.

Now briefly consider the annotation of arcs. All arc annotations are shown on Figure 6. Their purpose is to bind an output port name to an input port name. It is apparent from the graph that node **Split Interval** places data onto output ports **I1** and **I2**. Port **I1** is bound to input port **I** of the left **Integ Half** node. Thus data **Split Interval** places onto **I1** is sent to the left **Integ Half** node and data placed into **I2** is sent to the other.

Returning to the computation node annotation, the “vars” stanza defines variables that are local to the node and that its computation can read and modify.

The “firing_rules” stanza is very important. It serves two purposes. First, it defines conditions under which the node is permitted to execute. Second, it describes which local variables will have data placed in them that have been removed from designated ports. CODE's notation for firing rules is quite flexible and also sometimes complicated relative to other features of the language. The rule “I -> **i** =>” is simplest case. It signifies that

1. The node can fire when there is data waiting on port **I**.
2. When the node fires, one (structure in this case) is removed from **I** and placed in local variable **i**.

Thus, the **Integ Half** nodes simply wait for an incoming value. When one appears, they fire and produce an output.

The “comp” stanza defines what sequential computation will be performed when the node fires. The text is expressed in a language that is a subset of “C” that includes calls to externally defined sequential functions and procedures (such as **simp** which does the integration in this example). It is expected, but not required, that all significant sequential computations will be encapsulated in such external functions.

Finally, the “routing_rules” stanza determines what values will be placed onto output ports. As with firing rules, the notation is flexible and potentially complex, but this example is simple. The value of real variable **val** is placed onto queue **S**.

5.2 An Introductory Example: HeNCE

The equivalent HeNCE program looks exactly like the CODE graph in Figure 6, except for three points.

1. HeNCE graphs are read from bottom to top (this will be changed in a future release).
2. HeNCE computation nodes are always named by the (exactly) one sequential procedure they are required to call.
3. HeNCE arcs take no annotation.

The HeNCE graph is shown in Figure 7. All node annotations are shown. In the actual HeNCE system, the annotations are in pop-up windows.

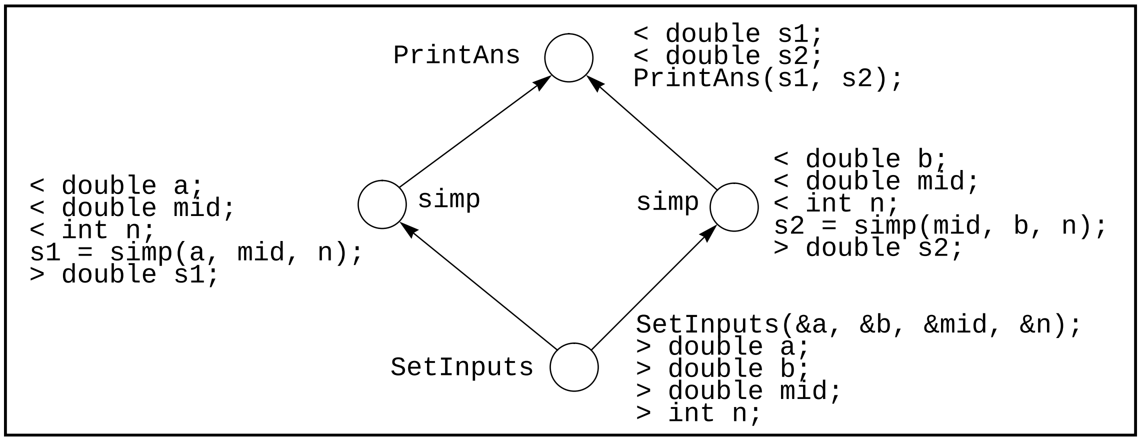


Figure 7. HeNCE Integration Program.

Although the HeNCE graph looks like the CODE graph, the meaning of HeNCE graph is very different. Except for some features that have not been discussed, arcs in CODE represent dataflow. Arcs in HeNCE represent two different concepts at the same time: control flow and variable name scope.

A HeNCE node is permitted to execute whenever all of its predecessors have executed. This is the only rule that defines when a node can run, and there is no implication that predecessor nodes have sent any data. There are no explicit node firing rules as in CODE. HeNCE has special control flow nodes that can alter the succession of node executions.

HeNCE node computations read and write variables. If a node reads a variable, the system defines that the value it will get is that set by the nearest predecessor in the graph that exports the variable. This will require an explanation and some background. Computation node annotations consist of three parts, two of which are optional.

1. Declaration of input and input-output variables (optional).

   The values of input and input-output values are read from the nearest predecessor node that outputs that variable. The value of the variable can be changed. New values of input-output variables can be seen by successor nodes. New values of input variables cannot. Input declarations contains an “<“and input-output declarations contain a “<>” token.

2. Call to a sequential procedure (required).

   The procedure may be written in either “C” or Fortran. The call’s actual parameters may be expressions. Variables that appear in the expressions are inputs, input-outputs, or outputs from node.

3. Declaration of output variables (optional).

   Output variables can be set by the node. Values are available to successor nodes. Output declarations contain a “>” token.

Consider the annotation of node **SetInputs** in Figure 7. It calls a “C” routine called **SetInputs** which provides values for variables **a**, **b**, **mid**, and **n**. The variables are made available to successor nodes because they appear in output declarations.

The two **simp** nodes are very similar, but one uses input declarations to read **a**, **mid**, and **n** from its nearest predecessor (**SetInputs**) and the other reads **mid**, **b**, and **n**. The left **simp** node makes **s1** available to its successors in the graph, and the write makes **s2** available. These variables hold the results of the integration. Subroutine **simp** actually performs the integration. It is a “C” procedure.

Node **PrintAns** reads **s1** and **s2**. It calls “C” procedure **PrintAns** which sums them and prints their value which appears in the HeNCE console window when the program is run.

5.3 Block Triangular Solver Example

We will use a somewhat more sophisticated example to introduce a few of the more advanced facilities of CODE and HeNCE. The problem is to solve the system **Ax** = **b** for a dense lower triangular matrix **A**. The algorithm to be used is quite simple and involves dividing the matrix and the vector into blocks as shown in Figure 8. Each “a” in the figure represents a sub-matrix of **A** and each “b” represents a sub-vector of **b**. Let the number of sub-blocks be **N**.

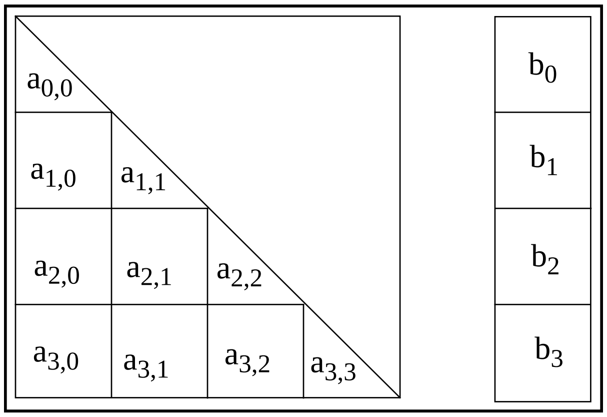


Figure 8. Blocked Matrix and Vector.

The algorithm replaces **b** with the solution vector **x**. The case for **N** = 4 is shown below.

$$b_0 = a_{0,0}^{-1} b_0$$

$$b_1 = a_{1,1}^{-1} (b_1 - a_{1,0} b_0)$$

$$b_2 = a_{2,2}^{-1} (b_2 - a_{2,0} b_0 - a_{2,1} b_1)$$

$$b_3 = a_{3,3}^{-1} (b_3 - a_{3,0} b_0 - a_{3,1} b_1 - a_{3,2} b_2)$$

Notice that once b_j has been computed, the operations $b_i = b_i - a_{i,j}b_j$ can be performed in parallel for i = j+1 to N-1. Thus, the algorithm proceeds iteratively, working on columns of the blocked system one at a time from left to right. Let **Solve** be a sequential function that solves this problem (applied to a single block).

```
To process the jth column do
  Solve($a_{j,j}$, $b_j$);
  for each i from j+1 to N-1 do
     $b_i$ = $b_i$ - $a_{i,j}$ * $b_j$;
```

Each of the iterations of the for loop can be done in parallel. Assuming the sub-blocks are of adequate size, each iteration represents a fairly coarse grain computation– a multiplication of a matrix sub-block by a vector sub-block with the resulting vector subtracted from another vector sub-block. For the remainder of this discussion assume that a procedure called **BlkMult** performs this operation.

The parallelism in this algorithm stems from the ability to perform the **BlkMult** operations “beneath” the **Solve** operation for a column in parallel. This is readily seen in a dataflow graph for the algorithm as shown in Figure 9. The “S” nodes are calls to **Solve** and the “M” nodes are calls to **BlkMult**. In the next few sections we will show how to express this algorithm in CODE and in HeNCE.

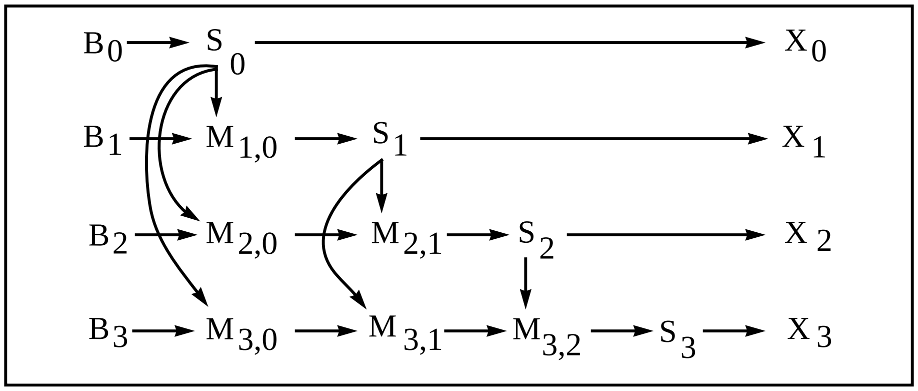


Figure 9. Dataflow for Block Triangular Solver.

5.4 The CODE Language

Before presenting the CODE block triangular solver, we introduce all of the icons that may appear in CODE graphs. They are shown in Figure 10. Many of the icons are used to define the interface to a graph. CODE graphs can call other CODE graphs by means of the Call icon shown. Arcs incident upon Call nodes are actual parameters of the call. These arcs are bound to interface nodes in the called graph via a name binding that is an attribute of the actual parameter arcs. This is similar to arcs binding port names between two nodes as seen above. Interface nodes are required to have names that are unique within the graph.

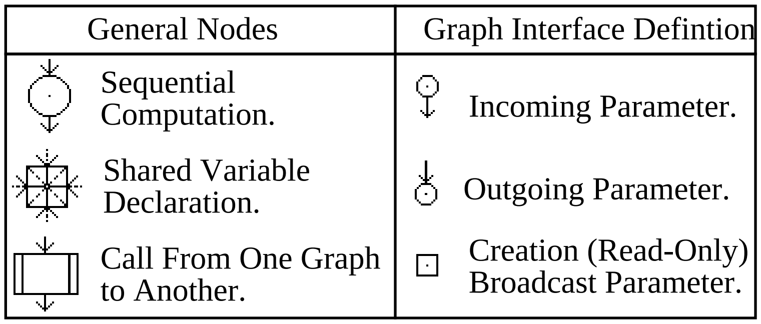

General Nodes
Graph Interface Defintion
Sequential Computation.
Shared Variable Declaration.
Call From One Graph to Another.
Incoming Parameter.
Outgoing Parameter.
Creation (Read-Only) Broadcast Parameter.

Figure 10. Node Icons in CODE.

The small circle interface nodes bind incoming and outgoing parameters in a very straight-forward way. Consider an input. As Figure 11 shows, the arc entering the Call becomes associated with the arc leaving the interface node. In effect, the two arcs become one.

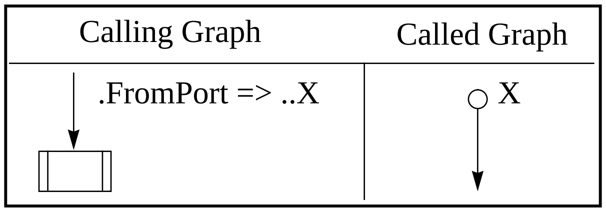


Figure 11. Formal-Actual Binding in CODE.

Creation parameters are also bound to an incoming arc. They extract exactly one value from this arc at the time the called graph is instantiated at runtime. All nodes within the called graph may use the creation parameter name as a constant. Its value comes from the arc.

The shared variable icon is used to declare variables that will be shared among a set of nodes. Each node must declare whether it requires read-only or read-write access to the variable.

CODE Block Triangular Solver

Many of these node types may be seen at work in Figure 12, a graph that implements the block triangular solver algorithm. Notice that the matrix (**a**), the size of the system (**n**), the size of the block system (**N**), and the size of a block (**blk**) are all passed to the graph as creation parameters. Their values are only read within the graph.

The known vector (**b**) is passed into the graph by means of a dataflow arc. It arrival causes node **DIST** to fire. The result vector (**x**) is passed out of the graph on a dataflow arc.

Node **DIST** sends the appropriate segments of **b** to the nodes that perform the **S** and **M** operations of Figure 9. Node **GATH** collects the segments of **x** from the **S** and **M** nodes and combines them into the single vector **x**.

In this implementation, a single instance of node **Solve** performs, one after another, all of the **S** operations shown in Figure 9, and **N** - 1 instances of node **Mult** perform the **M**'s. The mechanism by which multiple instances of a node are created is interesting and involves an interaction between the routing_rules of **Solve** and the annotation of the arc from **Solve** to **Mult**. **Solve**'s annotation is shown below. It contains a line

```
TRUE => { B_TO_M[i+WhichBlock] <- copy(b); : (i N-WhichBlock) };
```

which places a copy of the vector b onto an output port with an index. The notation

```
: (i N-WhichBlock)
```

causes index variable **i** to take on values from 0 to **`N-WhichBlock`** -1. Thus, the index of the output port takes on a range of values from **j**+1 to **N**-1, where **j** is the number of the column being processed. The annotation of the arc from **Solve** to **Mult** is

```
.B_TO_M[i] => [i].B_FROM_S
```

where **B_FROM_S** is node **Mult**'s input port. This routing rule is binding an output port name to an input port name as before, but now indices are involved as well. Suppose **i** in the arc annotation has value 7 (because the expression **i**+**WhichBlock** in the routing rule happens to be 7). Then, the arc specification binds **Solve**'s output port with index 7 to the input port **B_FROM_S** of node **Mult** with an index 7. Different instances of nodes have different indices. Thus, **Solve** sends data to the appropriate instance of **Mult** by using an index value in its routing rule. The arc annotation completes the binding.

Any number of node instances can be created in this way at runtime. The set of instances that can be created is dynamic in that it is determined by the runtime values of variables. This mechanism is quite powerful. It is possible for a node with a self loop arc to "expand" into an arbitrary graph, each node of which is a different instance.

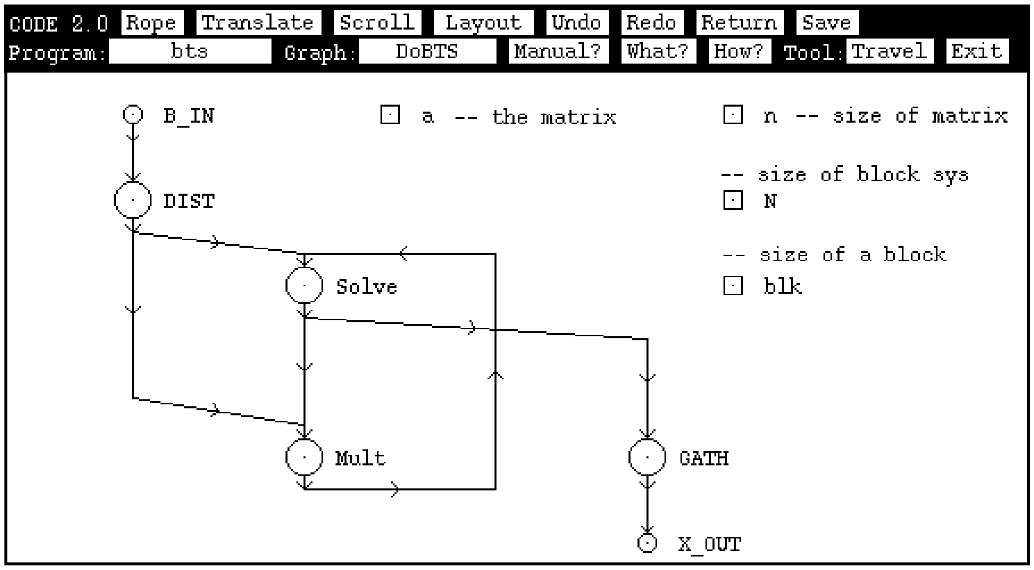


Figure 12. CODE Graph for Block Triangular Solver (DoBTS).

Nodes can have from 0 to 7 indices. The zero case is a default of sorts since no indices need exist in the program. The nodes in the integration example of Section 5.1 used no indices. That program could be improved by dynamically replicating a (now poorly named) node **Integ Half**. The resulting program would exploit N-way parallelism where N is chosen at runtime instead of a fixed two-way parallel structure.

The arc leaving **Mult** implies an iteration. First $\mathbf{S_0}$ is done and then $\mathbf{M_{i,0}}$ is performed in parallel for **i** = 1..**N**-1. Next $\mathbf{S_1}$ is done followed by $\mathbf{M_{i,1}}$ in parallel for **i** = 2..**N**-1, and so on. Since the block system size is an input to the program, the number of **Mult** nodes to create is not determined until runtime. In addition, **N** determines the number of times each node fires so this is also not known until runtime.

Some of the nodes in graph **DoBTS** have fairly sophisticated firing and routing rules. Perhaps it is useful to examine node **Solve**'s specification. Its firing rules can be understood by relating them to Figure 9. **Solve** fires the first time in order to perform computation $\mathbf{S_0}$. This computation depends on receiving a block of vector **b** from node **DIST**. Hence, **Solve** can fire when it receives a sub-vector (piece of a vector) on the arc from node **DIST** to its input port **B_FROM_DIST**. **Solve** fires next repeatedly to perform computations $\mathbf{S_1}$ to $\mathbf{S_{N-1}}$. For this it must receive a sub-vector from one of the **Mult** nodes on input port **B_FROM_M**. The node has a firing rule for each case. The node is permitted to fire when it receives data from either source.

After **Solve** fires to perform $\mathbf{S_k}$, it must send sub-vectors for $\mathbf{M_{j,k}}$ for **j** taking on values from **k**+1 to **N**-1. It's routing rules does just this. Variable **WhichBlock** is a counter that holds the value **k**+1. The routing rule also sends the portion of the solution vector **x** just computed to node **GATH**. **Solve**'s specification follows.

```
input_ports { Vector B_FROM_DIST; Vector B_FROM_M; }
output_ports { Vector B_TO_GATH; Vector B_TO_M; }
vars { Vector b; int WhichBlock; }
init_comp {
   WhichBlock = 0;
}

firing_rules {
   B_FROM_M -> b => ||
   B_FROM_DIST -> b =>
}

comp {
   solveblock(WhichBlock, a, b, blk);
   WhichBlock = WhichBlock + 1;
}

routing_rules {
   TRUE => { B_TO_M[i+WhichBlock] <- copy(b); : (i N-WhichBlock) };
            B_TO_GATH <- b;
}
```

This CODE program has been run on a 14 processor Sequent Symmetry. With a matrix of size 420 x 420 it shows a speedup measured relative to a straightforward sequential implementation of 3.5 with 14 processors. A parallel program that was hand-written using low level parallel primitives shows a speedup of 3.7, so CODE compares well with it. The theoretical maximum speedup of this algorithm is 4.9 with 14 processors.

5.5 The HeNCE Language

Like CODE, the HeNCE language also support additional icons other than the circle that represents a sequential computation. These new node types represent control structures. HeNCE has no facility for hierarchical implementation, although there are plans to add this ability to the language. For now, HeNCE graphs cannot call other graphs so there is no need for interface nodes. Figure 13 show all of HeNCE’s icons.

All of HeNCE's control flow icons work in pairs. One icon begins a construct and another ends it.The subgraph that appears between the icons is acted upon. For example, the subgraph between a loop-begin and a loop-end node is executed repeatedly, much like the body of a “C” for loop. The loop-begin is annotated with a statement to assign its index variable an initial value, a termination condition expression, and a statement to give its index variable its next value.

```
(variable = initial_value;
 termination_condition;
 variable = next_value);
```

○	Sequential Computation.
⟲ ⟳	Loop Begin and End - Enclosed subgraph is iterated over an index range such as i = 0 TO N.
¿ ?	Conditional Begin and End - Enclosed subgraph is executed only if an expression evalutes to TRUE.
△ ▽	Parallel Replication (Fan) Begin and End - Enclosed subgraph is replicated such that all copies execute in parallel. Copies are indexed as in i = 0 TO N.
⊓ ⊔	Pipeline Begin and End - Enclosed subgraph is replicated to form a pipeline with indexed stages.

Figure 13. Node Icons in HenCE.

Figure 14 shows a HeNCE loop and a static graph that its execution mimics. The loop-end node (and all other construct-ending nodes) requires no annotation. Notice that there is no explicit arc back to the start of the loop as flow chart would have. HeNCE graphs are acyclic. The subgraphs in a HeNCE control construct can contain other control constructs, but they must be properly nested.

Conditional node pairs define an “if-then” structure. The conditional-begin node annotation contains an expression. If the expression evaluates to TRUE (meaning non-zero, following the C language convention), the subgraph between the pairs is executed. Otherwise, it is not. HeNCE does not contain an “if-then-else” structure.

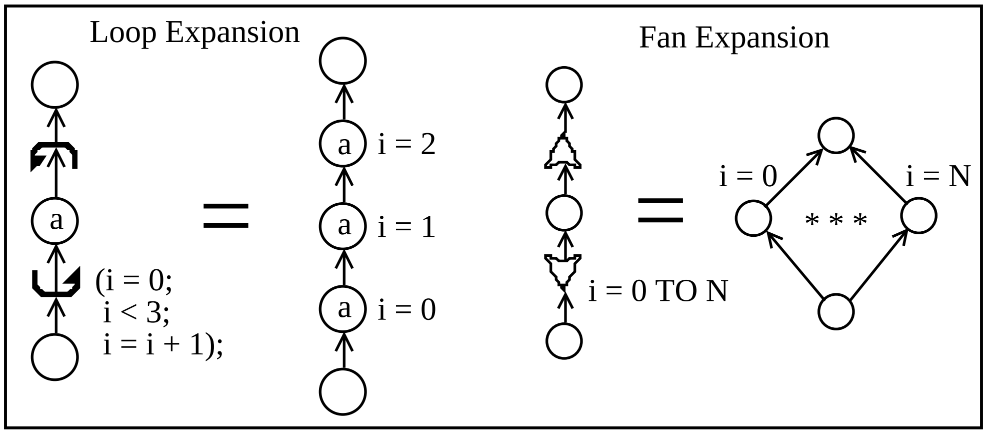


Figure 14. HeNCE Loop and Fan Constructs.

Fan node pairs create parallel structures. They replicate the subgraph between them and evaluate the replications in parallel. Figure 14 shows the effect of a fan. The fan-begin node's annotation consists of an index statement.

```
IndexVar = StartValue TO EndValue;
```

IndexVar takes on a different value in each of the replicated subgraphs. In this way, each replication has a unique index.

Pipe node pairs create a pipeline structure. The subgraph within the pipe is replicated, somewhat like a Fan node, but the dependence structure differs in a manner that is inspired by pipelines. Pipe constructs are rarely used.

HeNCE programs run on a collection of UNIX workstations on a common network. The workstations need not all be of them same type or even made by the same manufacturer. The capabilities and speeds of such a heterogeneous collection of machines can very widely. HeNCE graphs are converted into programs which run under the PVM message passing library [Gei93]. PVM is designed to be used directly by programmers as well.

The names of all of the workstations must be listed in the window segment labeled “Virtual Machine” (see Figure 16). The programmer also lists estimates of the cost of running each of the program’s sequential procedures on each of the machines. HeNCE uses this information to make intelligent choices about where to run nodes.

During execution, the utilization of the hosts in the virtual machine is displayed in the host utilization strip chart at the bottom right of the window. The horizontal axis is time, and there is a horizontal bar for each host which is divided into segments with colors that signify the state of the host. Figure 15 shows a case in which total utilization is poor since only one processor is running most of the time.

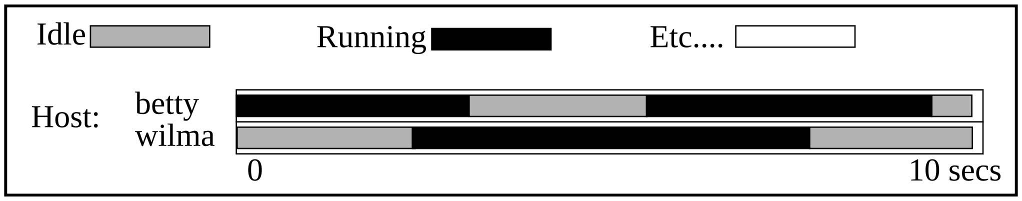
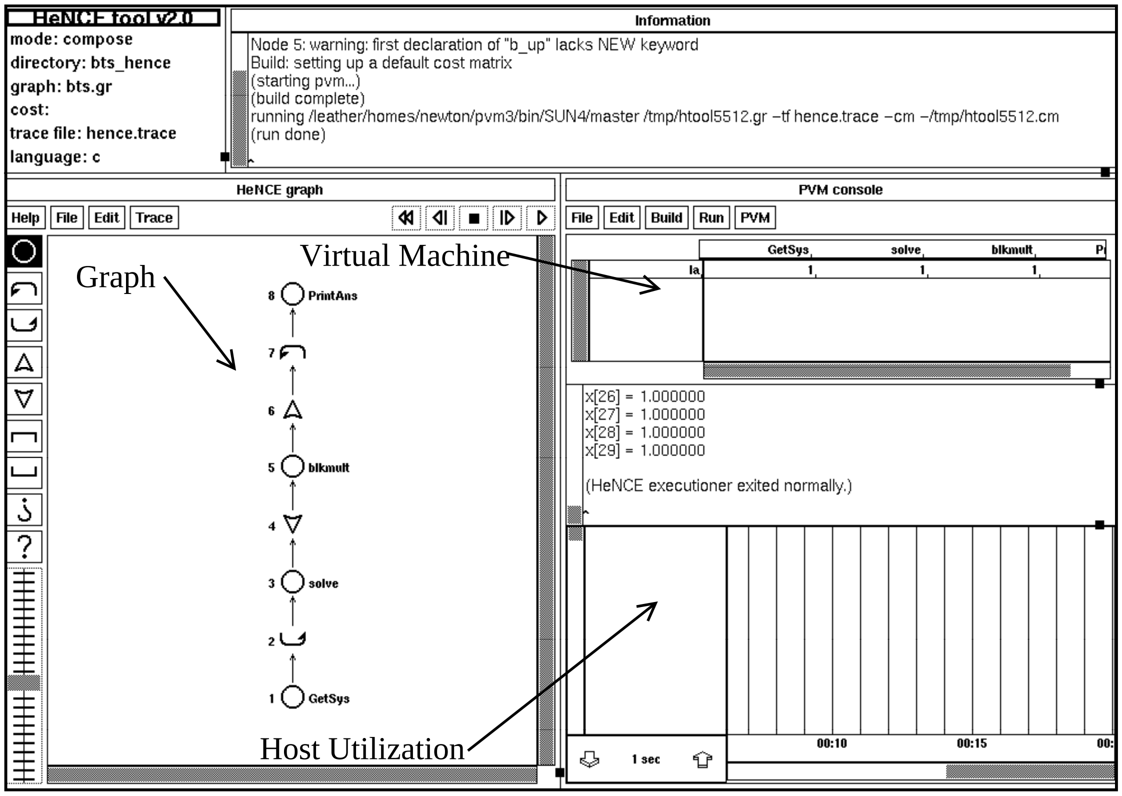


Figure 15. HeNCE Host Utilization Strip Chart.

HeNCE also changes the colors and shapes of the icons in the graph during execution to animate the state of the computation as it runs. Furthermore, this information is captured in a trace file and can be replayed after the program's execution is complete.

HeNCE Block Triangular Solver

The block triangular solver can also be expressed in HeNCE, and in some respects it is simpler in HenCE than it is in CODE. The program is shown in Figure 16. This figure is a screen dump which shows the entire HeNCE window as well as the segment in which the graph is drawn.



```
HeNCE tool v2.0
mode: compose
directory: bts_hence
graph: bts.gr
cost:
trace file: hence.trace
language: c
```

```
Node 5: warning: first declaration of "b_up" lacks NEW keyword
Build: setting up a default cost matrix
(starting pvm...)
(build complete)
running /leather/homes/newton/pvm3/bin/SUN4/master /tmp/htool5512.gr −tf hence.trace −cm −/tmp/htool5512.cm
(run done)
```

```
x[26] = 1.000000
x[27] = 1.000000
x[28] = 1.000000
x[29] = 1.000000

(HeNCE executioner exited normally.)
```

Figure 16. HeNCE Block Triangular Solver Program.

Execution begins with node **GetSys** running. This node declares the matrix **a** and vector **b** as well as other variables needed. All are made available to all successor nodes. **GetSys** gives them initial values. These variables are declared to be "NEW" input-output variables in order to have HeNCE allocate storage for them. If output-only variables were used, the sequential subroutine would have to allocate storage for them. The resulting vector **x** will be written into **b**.

```
-- annotation of node GetSys.
NEW <> int N;                -- Number of blocks in system.
NEW <> double a[500][500];   -- Matrix
NEW <> double b[500];        -- Vector
NEW <> int blk;              -- Number of elements in each block.
NEW <> int n;                -- Number of elements in matrix b.
GetSys(a, b, &n, &N, &blk);
```

The annotation of the loop node is simply "(**j**=0; **j**<**N**; **j**=**j**+1);" which iterates the subgraph between the loop-begin and loop-end node for every column of the blocked system.

The annotation of node **Solve** makes use of a HeNCE default. All variables that are used but not explicitly declared to be input, output, or input-output default to input-output status. The call to sequential procedure **solve** makes use of array index range expressions to pass just the required blocks of **a** and **b** to the routine.

```
-- annotation of node solve.
solve(a[j*blk:(j+1)*blk-1][j*blk:(j+1)*blk-1],
      b[j*blk:(j+1)*blk-1], blk);
```

The fan-begin node has a simple annotation "**i** = **j**+1 TO **N**-1;" that causes an instance of the enclosed subgraph consisting only of the node **blkmult** to be created for each value of **i** in the stated range. Within each instance, **i** takes on the appropriate value.

The instances of node **blkmult** make use of index variable **i** to select the blocks of **a** and **b** that they must process using array index range expressions as before. The portion of **b** that contains the solution of the last call to **solve** is stored in **b_up**.

```
-- annotation of node blkmult.
< double b_up[blk]=b[j*blk:(j+1)*blk-1];
blkmult(a[i*blk:(i+1)*blk-1][j*blk:(j+1)*blk-1],
        b[i*blk:(i+1)*blk-1], b_up, blk);
```

The annotation of node **PrintAns** also makes use of the default that variables that are not otherwise declared are input-output. Notice that the node automatically gathers the individual blocks of **b** that **solve** wrote into one vector of length **n**.

```
-- annotation of node PrintAns.
PrintAns(b, n);
```

5.6 CODE and HeNCE Compared

One of the challenges in research in visual programming is evaluating new ideas. We have found that it is necessary to actually implement systems and use them in programming projects and in university programming classes in order to thoroughly understand new ideas' merits and limits. Results are often subjective and context-dependent. Many ideas have both virtues and limitations. For example, CODE's firing rules are powerful but complicated. For some problems they are desirable and even necessary, but they increase the training time for new CODE users, and one of our chief goals has always been ease of use.

Implementing the various versions of HeNCE and CODE has allowed us to create ever more effective visual parallel programming environments. This section contrasts the two languages and points out circumstances in which one model may be more effective than the other. However, an overall conclusion regarding the two languages is probably not possible as each has strengths which stand out in different circumstances. This conclusion suggests that visual programming environments could benefit from supporting multiple representations, including textual ones.

1. Node firing conditions are explicit and general in CODE.

Programmers must explicitly define the exact circumstances under which a computation node is allowed to execute in CODE. The specification language is quite flexible and general, and firing conditions can depend on the internal state of the node. For example, it is easy to define a node that non-deterministically merges data from two streams in CODE. Such a computation is impossible to state in HeNCE.

It is tempting to say that HeNCE's firing rules are fixed. Nodes are permitted to fire when all predecessor nodes have fired, but this is an oversimplification. Execution of a HeNCE node is dictated also by the control flow constructs in which it is embedded just as is the case with statements in conventional languages. Thus, HeNCE firing conditions are somewhat less explicit.

CODE's firing rules are explicit and general, but can get complicated and wordy. HeNCE's firing rules are simple and concise, but are not always adequate to express algorithms. They do appear to be adequate for many interesting numerical algorithms, however.

2. CODE is capable of expressing more dynamic graph topologies.

Both due to its powerful firing rules and its method of instantiating nodes, CODE is capable of expressing communications patterns that HeNCE cannot. For example, CODE can accept an adjacency graph as input data and create a graph with the specified topology. Of course, such arbitrary expansions limit the extent to which CODE visually displays parallel structure. The static display of programs whose structure is determined at runtime is a significant research goal.

3. Explicit dataflow increases the complexity of graphs.

CODE shows all dataflow or common shared variable access via arcs. This is desirable in that it shows more completely the communication patterns of programs, but dataflow graphs are often complex, and worse still they are often unstructured. Programs with complex dataflow can become a rat's nest of arcs. This can be hard to understand and is also cumbersome since programmers must individually annotate all of the arcs.

HeNCE programs are related to flow charts of structured programs. They are concise and orderly even when graphs become large. Dataflow is implicit so less structural information is presented to the programmer, but computational elements are still clear and well encapsulated and parallel structure is displayed. On the negative side, since dataflow is implicit it is possible for programmers to make errors in which the “wrong” node is another node’s nearest predecessor for some variable.

4. HeNCE lacks hierarchy.

Since HeNCE graphs cannot call other HeNCE graphs, hierarchical implementation is not supported. Thus, the current implementation of HeNCE is ill-suited to large projects since graph sizes become excessive. This shortcoming is not a necessary aspect of HeNCE’s model. A future version of HeNCE will allow graph calls.

5. CODE’s basic unit of reuse is the sequential computation.

CODE is designed with the idea that the sequential computation node is the basic unit of component reuse. The CODE model supports libraries of computation nodes. Thus, CODE computation nodes must be completely encapsulated. They must have well defined interfaces, and they must be completely defined in isolation from other elements of a graph. This is the reason that CODE ports exist. They are a node’s formal parameters.

Nodes in HeNCE to not satisfy this property due to the manner of their use of variable names. If a programmer copies a node from one HeNCE program to another, he or she will likely have to edit the node when it is placed in its new context. For example, the new program may name some array **A** while the old program called it **B**.

CODE allows name binding on arcs, thus bridging the name spaces of any two nodes. The downside is that programmers *must* specify name bindings on every arc. The CODE computation node is somewhat analogous to a simple statement in a conventional programming language like Fortran. It is cumbersome to be required to specify name binding between “statements.”

Both CODE and HeNCE (with Call added to its model) support reuse on the graph level. This is not cumbersome since graph calls are much less common– like subprogram calls in Fortran.

6. CODE has efficient mechanisms for blocked arrays.

Both CODE and HeNCE were designed for computations involving arrays and many modern array algorithms are based on blocking as in the block triangular solver example above. It was not discussed, but CODE supports mechanisms for blocked arrays which are both efficient and reasonably simple. Programmers can use a 4 dimensional array to represent a blocked two dimensional array. The array is a two dimensional array of two dimensional arrays. HeNCE programmers can do this as well but for implementation reasons it is not as efficient.

HeNCE’s mechanism for specifying blocks by index ranges (as in the HeNCE block triangular solver) is also not efficient and arguably not simple. As the expressions used in the index ranges become complex, it becomes increasingly difficult for compilers to find a range’s “meaning” and implement a partitioning operation directly rather than by element by element copying at runtime. The compiler cannot statically analyze data access patterns. Also the expressions can become complex for programmers. Note that the HeNCE block triangular solver assumes that the degree of parallelism (size of the block array) evenly divides the size of the array. If this were not the case, the necessary expressions would be much more complex.

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

5.7 A Proposed Language

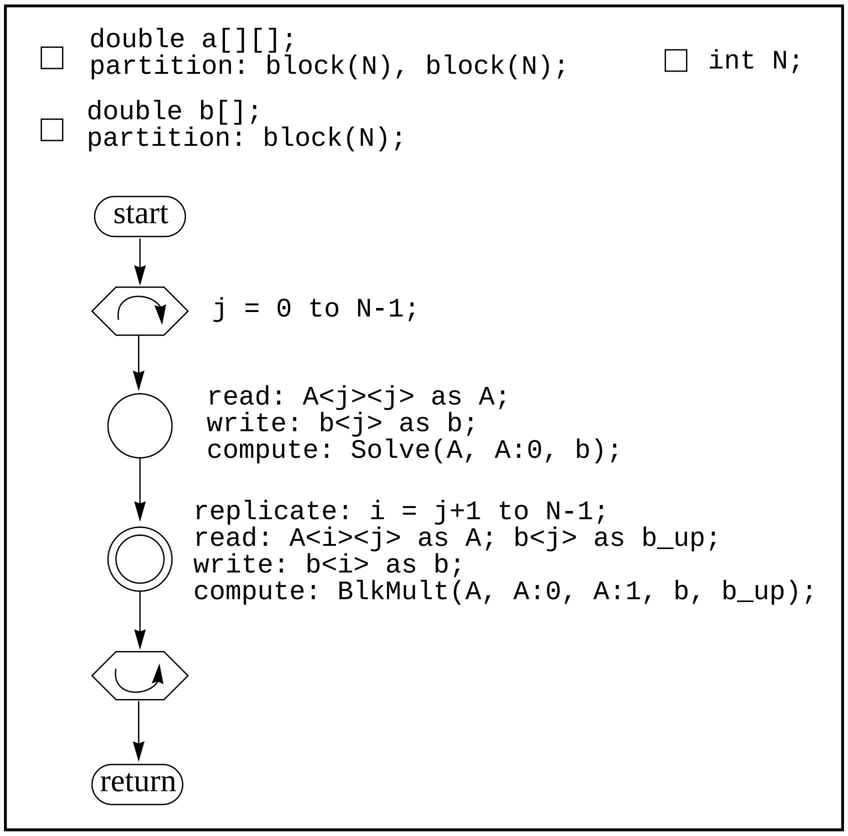
Visual programming language designers must search for a point of compromise between the extremes of ease of use and flexibility. This point is dependent on the problem domain of interest. We target computational science and hence have a bias towards numerical and matrix-oriented algorithms. In this domain, the balance point probably lies between CODE and HeNCE, especially since execution efficiency is also a major goal. Model expressiveness and effective implementation also tend to be competing goals.

Figure 17 shows the block triangular solver expressed in a language that is to be even simpler to use than HeNCE without being much less capable. The model supports graph calling and so has interface nodes as in CODE. Graphs show control flow and so are closer to the HeNCE model. Shared variables accessible within a graph are defined as separate icons as in CODE, but arcs are not required to show that a computation node access a variable. The unit of reuse is the graph since concise representations are required. As in HeNCE, arcs require no annotation.

There are two classes of computation nodes: replicated and non-replicated. Double circle icons represent replicated nodes. Both are annotated by the set of variables for which read access is needed, the set for which write access is needed, and a sequential computation.

Array variable icons allow the direct definition of partitions to support blocked algorithms. Angle brackets (as in <expression>) are used to access blocks, and square brackets (as in [expression]) access array elements. Blocks may overlap, but only one block “owns” an element. Writes to overlap regions in block that do not own them are local only.

Finally “:” is a size operator. For example, **A:i** returns the number of elements in the **ith** dimension of **A**, counting from zero.



double a[][];
partition: block(N), block(N);
int N;
double b[];
partition: block(N);
start
j = 0 to N-1;
read: A<j><j> as A;
write: b<j> as b;
compute: Solve(A, A:0, b);
replicate: i = j+1 to N-1;
read: A<i><j> as A; b<j> as b_up;
write: b<i> as b;
compute: BlkMult(A, A:0, A:1, b, b_up);
return

Figure 17. Block Triangular Solver in Proposed Language.

We believe that a simple model such as the one outlined above would be a valuable parallel programming tool. Concerns that it is not adequately flexible can be addressed by designing environments in which many different representations (both parallel and sequential) can co-exist. Beginning parallel programmers can begin with the most simple representations and learn the more complex on an as needed basis. All representations must provide a unit of reuse with a common semantics in order to interoperate. We adopt the simple semantics of an atomic computation which accepts inputs and computes outputs with no inter-unit interactions in between.

6.0 Debugging in the Visual programming Environment

Both logical and performance debugging are important aspects of the parallel programming processes. Performance debugging relates to understanding and improving the speed of a program, and logical debugging relates to it correcting errors in the logic of a program. This section focuses primarily on logical debugging.

Debugging establishes the relationship between the program (typically some small segment of a large program) and its execution. The entities involved in debugging include: (i) the program, P, (ii) a specification of a program's (or a program segment's) expected execution behavior (which we call M for model of behavior), and (iii) some representation of the program's actual execution behavior, E. Debuggers for programs written in pure text forms typically use a different representation for each of these entities, and this imposes much additional work on the programmer. This is especially true since execution environments and representations are typically very different across parallel architectures. Ideally each of the entities P, M, and E would be expressed in the same representation so that the programmer would not have to understand and manipulate several different notations. One of the major benefits of visual directed graph programming is that it supports the formulation of parallel program debuggers in which all of the entities can be expressed in a single notation. This simplifies the task of debugging not only because it allows the programmer to think in terms of the program which is what he understands but also because it admits of ready automation of the often tedious tasks of comparing expected and actual behaviors. It also facilities identification of the logic faults in the program. Efforts are ongoing to implement such a debugger in the context of the CODE system [Hyd93].

The example which follows illustrates how the visual directed graph representation of a program supports both the problem formulation and the analysis steps of parallel debugging. As before, the program is a directed graph whose nodes are sites for the execution of atomic actions.

Definition 1: An *action* is an operation for which there exists a known input/output relation for a given initial state. It is the atomic function of a node.

The execution of a program is the traversal of the graph starting with an assignment of an initial state, until the execution of a final state. Traversal of the graph causes execution of actions at the nodes and generates a partially ordered set of action executions or events.

Definition 2: An *Event* is an execution of the action at a node of the program graph.

The partially ordered set of events of an execution defines a directed acyclic graph of events that corresponds to instantiations of the actions defined at the nodes of the program graph structure.

Definition 3: *Debugging* is the process of identifying those actions of the program that are responsible for the failure of the program to meet its final state specification.

Let us start with a program, P, that has been observed to produce invalid final states for execution from one or more initial states. We use a version of the Block Triangular Solver for CODE (Figure 12) to which we have deliberately introduced a sequencing error to illustrate the various steps of the debugging process. We have incorrectly coded the array index of **`B_TO_M`** in the routing rule for **Solve** as

```
B_TO_M[i]
```

instead of

```
B_TO_M[i+WhichBlock].
```

The execution of this bugged version starts with an initial state where N=5, and terminates with a segmentation fault. We now follow the different steps of the debugging process:

1. Identify and select the portions of the program whose behavior is to be monitored.

This is a set of “suspect” nodes or subgraphs. Note that it is typically impossible to monitor the entire execution behavior of large complex programs (which are actually the ones that need debugging). The visual/graphical representation of P makes the selection of suspect portions of the program easy. In our example, we click on the **Solve** and **Mult** nodes of the graph of Figure 12 to inform the debugger that they need to be monitored. The debugger makes additional preparations to filter out event executions of other nodes like **Dist** and **Gath**. This greatly helps in later steps as much of the irrelevant information is filtered out.

2. Specify the expected execution behavior of the set of nodes that are to be monitored.

The natural mode of representation of execution behavior for graphical programs is the partially ordered set of events expected to be generated by the execution of the actions at the nodes of the suspect subgraphs. Let us call this representation, M, for model of expected execution behavior. M is given as a partially ordered set of events. We can either construct this set of events directly, or construct a graph of actions whose execution will generate the desired partially ordered sets of events. In this case, we specify M by drawing a graph that is shown in Figure 18(a). In the data-flow description of Figure 9, we expect that the ***i***-th execution of **Solve** will be followed by the executions of **Mult** instances whose node indices would range from ***j*** = ***i*** to ***N-1***.

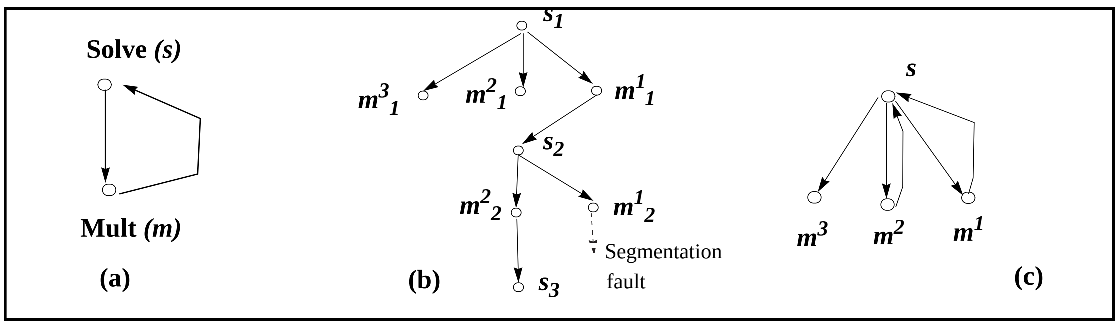


Figure 18. (a) Graph of M. (b) Partial Order graph of E. (c) Elaborated Graph of M.

3. Capture the execution behavior of the selected portions of the program.

The execution behavior is a partially ordered sequence of events that actually occurred in the execution. Let us call this partially ordered set of events E. The user obtains E by selecting a set of program nodes with the mouse and then running the program. The system then automatically records all of the necessary events and orderings. The selection of **Mult** and **Solve** nodes in step 1 produced such an annotation. As a result, the actual execution behavior observed by the debugger as shown in Figure 18(b) contains event executions of only the selected nodes. In the figure, event s_i indicates the ***i***-th execution of **Solve**, and event m^j_i indicates the ***i***-th execution of that instance of **Mult** whose node index is ***j***.

4. Map E to M to determine the locations where the actual and expected events first diverge.

The mapping of E to M can be done automatically since they are specified in the same representation. The result is identification of event sequences in E that do not correspond to the allowed set defined in M. In Figure 18(b), we note that the events $\boldsymbol{m^1_1}$, $\boldsymbol{m^2_1}$ and $\boldsymbol{m^3_1}$ follow event $\boldsymbol{s_1}$. We, however, expected that events $\boldsymbol{m^j_i}$ that follow event $\boldsymbol{s_i}$, would have node indices that range from $\boldsymbol{j} = \boldsymbol{i}$ to $\boldsymbol{N-1}$. As N = 5 and there is no event $\boldsymbol{m^4_1}$, this detects the occurrence of an unexpected behavior. Moreover, the mapping of E to M gives an elaborated graph of M as shown in Figure 18(c). This is a run-time structure that shows dynamically created instances of nodes. The elaborated graph is obtained from the partial order graph of E in Figure 18(b) by folding back the later executions of a node, to their first execution. Figure 18(c) shows three dynamically created instances of node **Mult**; $\boldsymbol{m^1}$, $\boldsymbol{m^2}$ and $\boldsymbol{m^3}$. Note that we expected four instances.

5. Map the elaborated graph of M back to P to define corrective action.

Since the elaborated graph of M contains instances of the nodes of P, the mapping is automatic, and guides us towards the offending action in P. The mapping from Figure 18(c) to Figure 12, helps in ascertaining the cause of the unexpected number of instances of Mult. As explained in Section 5.4, the creation of multiple instances of **Mult** is tied to the indices of the output port specified in the routing rule of **Solve**. Output port **`B_TO_M`** of **Solve** connects to **Mult** and the data placed on its indices is responsible for creating various instances of **Mult**. The mapping, therefore, guides us to the incorrect coding in the routing rule of **Solve**.

There are various points that should be noted in the above process. A programmer would often cycle through these steps a number of times before identifying the bug. In each cycle, the programmer will progressively come closer to the offending piece of code.

The use of actions, instead of events, in the representation of M greatly helps the debugger in filtering out of the irrelevant information. This restricts the execution history displays of E to only the events that are of interest to the user. This filtering greatly simplifies the checking of M that can either be done visually by the user with the help of the execution displays provided by the debugger, or can be done automatically by the model checking facility of the debugger.

An Animation facility provided by the debugger is simply a visualization of the mapping of E to M. The elaborated graph acts as an underlying structure for animation and greatly helps in animation.

Thus, a visual programming environment provides a consistent graphical representation for all the different entities used in the debugging process and simplifies the design of a concurrent debugger that coherently relates the various steps of the debugging process. It also provides a unified framework for supporting different concurrent debugging facilities like execution history displays, animation, and model checking facilities.

7.0 Related Work

There has been much work on visual programming languages and environments for sequential systems. Prograph [TGS92] and PICT [Gli84] are substantial examples. We will focus on visual parallel programming languages.

7.1 Older Systems and Proposals

CODE and HeNCE are certainly not the first systems to be designed for visual programming of parallel systems via graphs that show parallel structure. This sections surveys some of the earlier attempts.

Karp and Miller [Kar66] proposed a graph-based model of parallel computation that includes non-fixed firing conditions. The model also permits proof of determinacy and useful theorems on computation terminations and bounding the size of queues on arcs. The model is capable of expressing some interesting numerical algorithms, but is not flexible enough for general use.

There have been several proposals for visual dataflow oriented programming languages. Adams's model [Ada68] is an early example. Computations are deterministic. There are sophisticated techniques for mapping inputs to outputs, as firing and routing rules do in CODE. Ackerman [Ack82] provides a general discussion of early ideas in dataflow languages. Keller and Yen [Kel81] discuss directed graph programming and Davis and Keller [Dav82] present a dataflow language with special purpose nodes for non-standard firing rules and discuss graph composition.

- CODE 1.x

J.C. Browne has been investigating computation graph systems for many years. The general advantages of such systems and the outline of a model of parallel computation are presented in [Bro85], and Browne and his students also developed several earlier versions of CODE [Bro89, Jai91]. These form the intellectual basis for the current version but are much less capable. CODE 1.2 served as the basis for experiments in reuse within the context of a graph model [Lee89, Bro90].

- Schedule

Schedule [Don86] is a visual computation graph oriented system that facilitates calling separate sequential routines. Ideas from it influenced later systems including Phred, HeNCE, and some versions of CODE.

- Phred

Phred [Beg91b] is a graphical system that greatly expands on the semantics of schedule. It uses graph grammars in its language definition and special nodes for firing rules, computations, and some runtime determined computation structures. Programs consist of a combination of control flow and dataflow graphs. Phred heavily influenced HeNCE.

- Neptune

Neptune [Tra90] is a computation graph based graphical programming environment that is similar in most respects to older versions of CODE.

- Poker

Poker [Sny85] is noteworthy as an early graphical parallel programming environment. It is, however, a fairly distant relative of CODE and HeNCE. It graphical displays lattices of virtual processing elements and allows these nodes to be annotated with a sequential algorithm.

- Paragraph

Paragraph [Bai91] is a very interesting model of parallel programming in which computation graphs are expressed by productions in a graph grammar, thus allowing dynamically structured graphs.

7.2 Recent Systems

There are some more recent systems that are similar to CODE, many of which are still under active development. Some are primarily general purpose parallel programming environments. Other mostly serve as platforms for research in scheduling.

- Paralex

Paralex [Bab92] is a graphical programming environment with a model similar in expressive power to earlier versions of CODE. However, it incorporates sophisticated facilities for fault tolerance and dynamic load balancing on target architectures such as networks of workstations.

- PPSE

The Parallel Programming Support Environment (PPSE) [Lew90] is an ambitious integrated environment for the development of parallel programs. It consists of many tools beginning with a dataflow computation graph based graphical programming environment called Parallax [Lew93]. The Parallax language includes hierarchy, dataflow (with named) ports, and variable storage nodes.

At this time, firing guards are not yet implemented. This greatly reduces the expressiveness of its model of computation. For now, PPSE’s primary role appears to be as a basis for research in scheduling.

Other tools in PPSE include a graphical target machine description system, task analysis, mapping, and scheduling tools, a code generator that targets the STRAND language, a simulator, and various performance and schedule visualization tools.

- VERDI

VERDI is a visual language used to develop distributed programs in Raddle [Gra90]. Programs consist of a set of graphs. Control flow in a graph is represented by the flow of a token through it. Data may be attached to the token. Communication and synchronization are carried out by means of an N-party interaction facility in which tokens must arrive at a box (with a common label) in each of the interacting graphs. The language supports non-determinism since tokens can flow to varying sets of boxes, each of which would enable a different N-party interaction. The system non-deterministically chooses. The language supports indexed replication, somewhat like HeNCE's.

- D^2R

D^2R (Dynamic Dataflow Representation) [Ros93] is a recent model that bears much resemblance to HeNCE. However, it appears to have an orientation towards scheduling research. It has been implemented on a multi-transputer system called DAMP [Bau91].

As with older versions of HeNCE, the language has both a textual and a graphical representation. There are special constructs for loops, parallel fork-join constructs, and alternatives (n-way if). The model is dynamic since the width of a fork-join, and the loop count of a loop are runtime parameters. There is no hierarchy and no concept of general firing rule guards.

Simple dataflow nodes wait for data on all input before firing and produce data on all outputs when firing is complete. Functions that nodes run are stored in separate files.

- ALEX

ALEX [Koz90] is an interesting functional visual parallel programming language in which the focus is on drawing pictures of data, typically arrays. For example, a matrix multiply program is created by drawing two rectangles that represent input matrices and then drawing a representative row within one and a column within another. In conjunction with library routines for multiplication and addition, the diagram is then extended to show how to form the resulting matrix from the rows and columns of the input matrices.

- PFG

PFG is a graphical parallel programming language whose formal operational semantics are described by HG [Sto90]. Like HeNCE, it uses special icons to create constructs that control node execution. It has facilities for parallel branching (generalized conditional branching) and non-deterministic branching. PFG is billed as an "assembly language" for higher level visual parallel programming languages.

8.0 Obtaining HeNCE and C

HeNCE and the PVM package it targets are available in source and binary form via xnetlib or by anonymous ftp to Internet host netlib2.cs.utk.edu. Both HeNCE and PVM run under X windows on a wide variety of UNIX workstations. Questions on HeNCE may be emailed to hence@cs.utk.edu.

CODE is available by anonymous ftp to pompadour.csres.utexas.edu. Only binaries for Sun 4 workstations are available. Questions may be emailed to newton@cs.utk.edu or browne@cs.utexas.edu.

9.0 Conclusion

Difficulty of programming is a major impediment to the wider acceptance of coarse-grain MIMD parallel computer systems. Visual parallel programming languages based on directed graphs have many attractive properties that can help to lessen this problem, especially in some application domains. Directed graphs naturally capture the multi-dimensional structure of parallel program behaviors. If a semantics is adopted in which basic nodes represent atomic sequential computations, this view supports a programming methodology that emphasizes the creation of parallel programs by the composition of sequential elements. This separates the concerns of creating sequential sub-computations and creating parallel structures.

A visual programming environment based on directed graphs helps programmers to understand the large-scale structure of their programs and this is vital for performance in a way that is not the case for sequential programs. Programmers must understand the granularity of their computations as well as having a general idea of the frequency with which different segments of there program run and interact with other segments. Furthermore, programmers must understand data partitioning and placement since data locality is necessary for speed on modern parallel architectures. Debugging of parallel programs is also difficult due to the complexity of the interactions between program elements that can arise. A visual directed graph framework allows all of these program aspects to be understood in a common high-level representation that presents programmers with abstractions that are well suited to the various phases of the programming process.

CODE 2.0 and HeNCE 2.0 are implemented visual programming languages that demonstrate many of these advantages. It is only through constructing CODE and HeNCE and using them that we have been able to evaluate these benefits and to continue the evolution towards better visual parallel programming abstractions. Although based on the same idea, the two languages differ significantly in detail, and their relative strengths and weaknesses are illuminating. HeNCE’s method of defining data interrelationships among nodes and node firing conditions is simple and concise but less expressive than CODE’s. CODE on the other hand, tends to be more wordy and harder to learn. Also complete simultaneous display of all dataflow relationships tends to yield complex unstructured graphs.

We believe that an environment which allows multiple graphical and textual relationships to co-exist is most desirable. Since programmers will then be able to choose representations that are most appropriate to a given situation. Also beginning programmers will be able to use simpler tools. The key to such an environment is the selection of Call semantics that apply to all representations. The semantics of sequential subroutines serve well. Components can be viewed as atomic in that they map inputs to outputs with no inter-component interactions in between.

10.0 References

[Ack82] W. B. Ackerman, “Data Flow Languages”, *Computer*, Vol. 15. pp.15-25, Feb., 1982.

[Ada68] D. A. Adams, “A Model for Parallel Compilations”, *Parallel Processor Systems, Technologies and Applications*, pp. 311-334, Spartan/MacMillan, New York, 1968.

[Bab92] Ö. Babaoglu, “Paralex: An Environment for Parallel Programming in Distributed Systems,” 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.

[Bau91] A. Bauch, R. Braam, and E. Maehle, “DAMP: A Dynamic Reconfigurable Multiprocessor System with a Distributed Switching Network”, Distributed Memory Computing, Lecture Notes in Computer Sciences, ed. A. Bode, Vol. 487, Springer-Verlag, pp. 495-504, 1991.

[Beg91a] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam, “Graphical development tools for network-based concurrent supercomputing,” Proceedings of Supercomputing 91, pages 435--444, Albuquerque, 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.

[Dav82] A. L. Davis and R. M. Keller, “Data Flow Program Graphs”, *Computer*, Vol. 15., pp.26-41, Feb., 1982.

[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.

[Eig91] R. Eigenmann, and W. Blume, “An Effectiveness Study of Parallelizing Compiler Techniques,” Proc. Intl. Conf. Par. Proc., 1991, pp. II 17-25.

[Gei93] G. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam, “PVM 3 User’s Guide and Reference Manual,” Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, 1993.

[Gli84] E. P. Glinert and S. L. Tanimoto, "PICT: An Interactive Graphical Programming Environment," IEEE Computer, Vol. 17, No. 11, Nov., 1984.

[Gra90] M. Graf, "Building a Visual Designer's Environment", in Principles of Visual Programming Systems, S.-K. Chang, ed., Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[Hir91] S. Hiranandani, K. Kennedy, and C.-W. Tseng, "Compiler Support for Machine-Independent Parallel Programming in Fortran D," Rice University, CRPC-TR91132, 1991.

[Hyd93] S. I. Hyder, J. F. Werth, and J. C. Browne, "A Unified Model for Concurrent Debugging," Proc. International Conference on Parallel Processing, July 1993.

[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.

[Kar66] R .M. Karp and R. E. Miller, "Properties of a Model for Parallel Computations: determinacy, Termination, and Queueing", SIAM J. Appl. Math., Vol. 14, No. 6, Nov. 1966.

[Kel81] R. M. Keller, and W.-C. J. Yen, "A Graphical Approach to Software Development using Function Graphs", IEEE COMPCON 81, Feb. 23-26, San Francisco, 1981.

[Koz90] D. Kozen, et al., "ALEX– An Alexical Programming Language", in Visual Languages and Applications, T. Ichikawa, E. Jungert, and R .R. Korfhage, eds., Plenum Press, New York, 1990.

[Lau90] R. Lauwereins, et al., "GRAPE: A CASE Tool for Digital Signal Parallel Processing," IEEE ASSP Magazine, Apr. 1990.

[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.

[Lew93] T. Lewis and H. El-Rewini, "Parallax: A Tool For Parallel Program Scheduling", IEEE Parallel and Distributed Technology, pp. 62-72, May 1993.

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

[New93] P. Newton, "A Graphical Retargetable Parallel Programming Environment and Its Efficient Implementation", Technical Report TR93-28, Dept. of Computer Sciences, Univ. of Texas at Austin, 1993.

[Pat90] David A. Patterson and John L. Hennessy, Computer Architecture: a Quantitative Approach, Morgan Kaufmann, San Mateo, CA, 1990.

[Ros93] J. Rost, “D^{2}R: A Dynamic Dataflow Representation for Task Scheduling”, ACM SIGPLAN Notices, Vol. 28., No. 8, August, 1993.

[Sny85] L. Synder, “Poker 3.1: A Programmer’s Reference Guide”, Dept. of Comp. Sci. Technical Report TR-85-09-03, University of Washington, Seattle, WA.

[Sto90] P. D. Stotts, “Graphical Operational Semantics for Visual Parallel Programming”, in Visual Languages and Visual Programming, S.-K. Chang, ed., Plenum Press, New York, 1990.

[TGS92] TGS Systems Limited, Prograph Reference, Halifax, Nova Scotia, Canada, 1992.

[Tra90] B. Traversat, “NEPTUNE: The Application of Course-Grain Data Flow Methods to Scientific Parallel Programming”, Ph.D. dissertation, The Florida State University, 1990.

[Yan91] T. Yang and A. Gerasoulis, “A Fast Static Scheduling Algorithm for DAGs on an Unbounded Number of Processors”, Proceedings of Supercomputing 91, pages 633–642, Albuquerque, 1991.