Experiences with CODE and HeNCE in Visual Programming for Parallel Computing

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

Abstract

Visual programming has particular appeal for explicit parallel programming, particularly coarse grain MIMD 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. 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.

Two visually-oriented parallel programming systems, CODE 2.0 and HeNCE, will be used to illustrate these concepts. The benefits of visually-oriented realizations of these models for program structure capture, performance analysis and debugging will be explored. 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

Parallel computer architectures have clearly emerged as the future environments for high performance computation for most applications. There is convergence on the MIMD model of parallel computation, at least in the current generation of parallel architectures. However, many different mechanisms are employed for communication and synchronization. This is of critical importance since synchronization and communication mechanisms are the means by which parallel structure is specified

The barrier to widespread use of parallel architectures is that writing parallel programs which are both efficient and portable is quite difficult. Parallel programming is more difficult than sequential programming because parallel programs must express both the sequential computations and the interactions among the sequential computations which define the parallelism. In order to achieve good performance, programmers must understand the large-scale structure of their programs: what computational elements are present and when and how they communicate.

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

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 as directed graphs.

1. Graphs are a more natural representation for parallel programs than linear text because parallel program behavior is inherently multi-dimensional due to multiple threads of control.
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.

Most current text based parallel programming languages either implicitly define parallel structure and hence require advanced compilation techniques or they embed communication and synchronization with sequential computation, thus making program structure difficult to understand. Furthermore, direct use of vendor-supplied procedural primitives can preclude portability. A programming process which separates specification of sequential computations from specification of synchronization and communication and expresses synchronization and communication directly but abstractly would alleviate these difficulties.

We will show that directed graph representations of parallel programs cleanly separate specification of sequential computations from synchronization and communication and express synchronization and communication independently of choice of procedural mechanism. We will then demonstrate that a compositional model of programming based on these concepts eases programming, provides for direct display of performance controlling parallel structure and allows execution behavior to be modeled directly in terms of program structure. These conclusions are actually not major leaps of concept. Recall that optimizing compilers generate directed graph representations of parallel programs and that performance monitoring systems use graphical displays. There is conventional wisdom that graphical displays of parallel structure are beneficial. The conclusions reached herein are natural extensions of this conventional wisdom.

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. However there has been much progress since then. See [New93] for a survey. Visual parallel programming 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. These advantages will be elaborated in the sections that follow.

2.0 Graph Representations of Parallelism

MIMD Parallel programs are inherently non-linear, regardless of how they are expressed, because parallel programs consist of multiple interacting threads of control. Directed graphs provide a simple and natural mechanism for representing these multiple threads and helping users to understand them.

Consider the sequence of assignment statements shown in Figure 1. A parallelizing compiler could analyze the statements, determine that the procedures called cause no dependences and execute some of the statements in parallel. The textual representation does not make it clear at a glance what the resulting parallel structure will be, but the equivalent dataflow graph does. Each circle in the dataflow diagram represents one of the statements. A statement can execute when it has received an input on each arc.

```
1: x = proc1();
2: y = proc2();
3: z = proc3(x);
4: w = proc4(x,y,z);
```

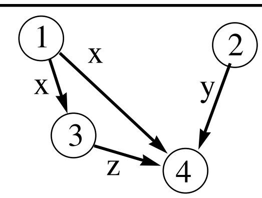

Figure 1. Dataflow Example.

Parallel programs expressed in other ways are also easier to understand when viewed as a graph. One common means of writing parallel programs is to insert calls to message-passing library routines into otherwise conventional sequential code. Several processes are run in parallel, and they interact by means of these messages.

The parallel structure of such programs can be quite difficult to understand since it is determined by message-passing calls that can be deeply embedded inside nested control flow constructs such as If and While statements. It can be hard to know precisely under which conditions a given message-passing call will be executed. This problem is commonly addressed by packages which animate the parallel program's execution, often as a space-time diagram in which there is a horizontal bar for each process and arcs are drawn from bar to bar to represent message sends. This diagram can be converted into a directed graph form simply by converting each bar into a sequence of nodes, dividing at points at which messages are sent or received.

Such animations of parallel program executions and useful and popular, but they suffer from the fact that the programmer must relate the space-time diagram back to the original textual representation of the program. Bridging this gap between program representation and behavior can be non-trivial.

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 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. For example, if the execution time of 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.

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.

4.0 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 discuss the advantages and disadvantages of 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 the computations will act upon.

4.1 An Introductory Example: CODE

Figure 2 shows an extremely simple CODE [New93] 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.

So, to create this parallel program, the programmer draws with the mouse a graph just as shown in Figure 2 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. A single dynamically replicated node could be used instead.

```
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; }
```

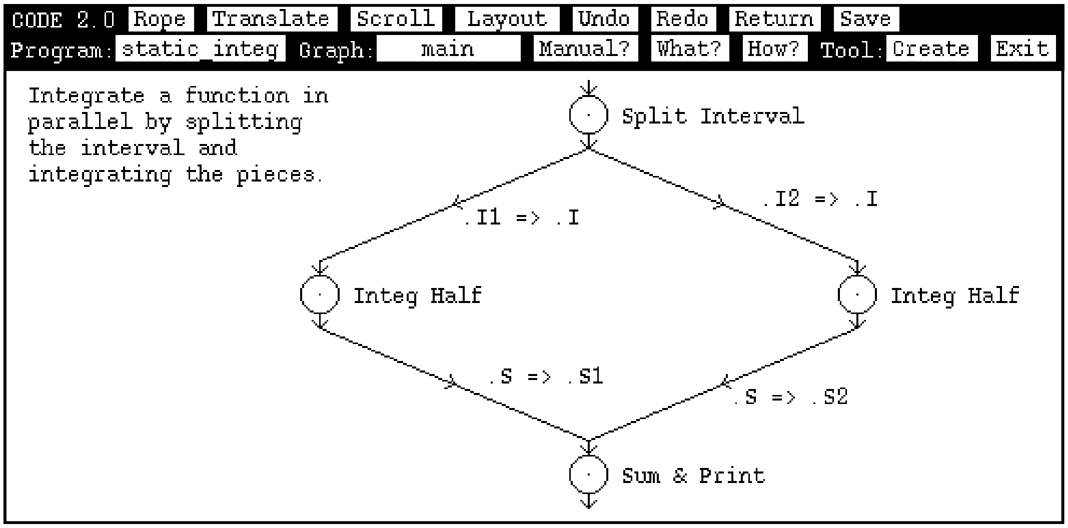

Figure 2. CODE Integration Program

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 that defines the work to be done) 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 2. 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**.

4.2 The CODE Language

The icons that may appear in CODE graphs are shown in Figure 3. 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.	Incoming Parameter.
Shared Variable Declaration.	Outgoing Parameter.
Call From One Graph to Another.	Creation (Read-Only) Broadcast Parameter.

Figure 3. Node Icons 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.

4.3 An Introductory Example: HeNCE

The equivalent HeNCE program looks exactly like the CODE graph in Figure 2, except for three points.

1. HeNCE graphs are read from bottom to top.
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 4. All node annotations are shown. In the actual HeNCE system, the annotations are in pop-up windows. An older version of HeNCE is described in [Beg91].

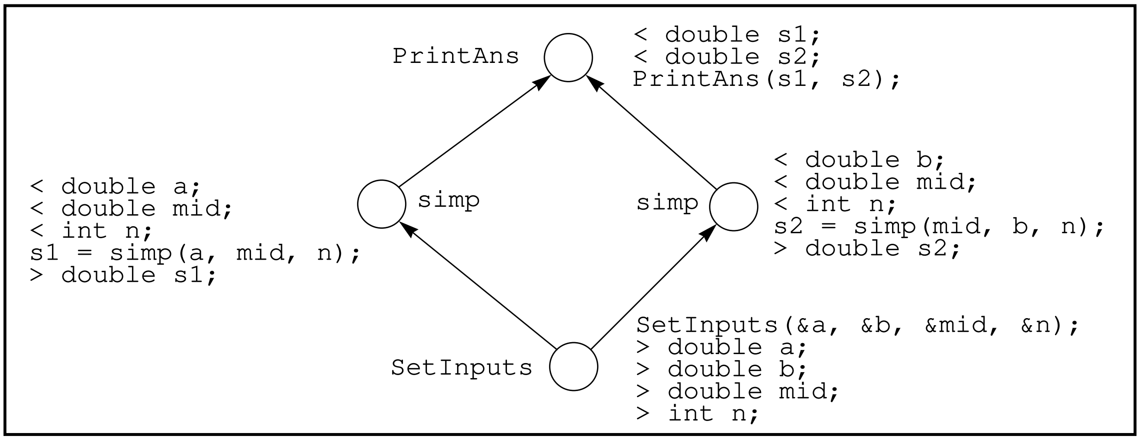

Figure 4. HeNCE Integration Program.

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 4. 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.

4.4 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 its model could support it. For now, HeNCE graphs cannot call other graphs so there is no need for interface nodes. Figure 5 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. The loop-end node (and all other end nodes) requires no annotation.

	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.

Figure 5. Node Icons in HenCE

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.

Fan node pairs create parallel structures. They dynamically replicate the subgraph between them and evaluate the replications in parallel. 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.

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.

4.5 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.

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. Thus, HeNCE firing conditions are somewhat less explicit.

CODE’s firing rules are explicit and general, but can be 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. 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 are not a natural unit of reuse 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."

5.0 Debugging in the Visual Environment

One of the advantages of visual parallel programming languages is that various programming tasks can be carried out using a common representation. Both logical and performance debugging are important aspects of the parallel programming process. Performance debugging involves 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 in a visual environment.

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. 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].

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.

The debugging process consists of a sequence of steps as outlined below.

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 parallel 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.

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.

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.

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.

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.

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 which 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.

6.0 Obtaining HeNCE and CODE

HeNCE and the PVM package it targets are available in source and binary form via xnetlib or by anonymous ftp to ftp.netlib.org. 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 runs only on Sun 4 workstations. For information, contact browne@cs.utexas.edu or newton@cs.utk.edu

7.0 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. 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.

8.0 References

[Beg91] 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.

[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.

[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.

[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.

[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.