Chapter 1
Visual Programming and Parallel Computing

Peter Newton*

Abstract

Visual programming languages have a number of advantages for parallel computing. They integrate well with programming environments and graphical program behavior visualization tools, and they present programmers with useful abstractions that aid them in understanding the large-scale structure of their programs. Such understanding is important for achieving good execution performance of parallel programs. Furthermore, graphical programming languages can be easier for non-specialists to use than other explicitly parallel languages since they relieve the programmer of the need to directly use low-level primitives such as message sends or locks.

This paper discusses some of these general advantages and presents simple examples in the existing visual parallel programming languages, HeNCE and CODE 2.0.

1 Introduction

It is widely believed that the use of parallel computation is limited by the difficulties associated with programming parallel machines. This fact limits non-specialists' access to parallel computing. Automatic parallelism detection in programs written in sequential (or non-explicitly parallel) languages would be a solution except that current detection technologies miss much parallelism. This paper focuses on languages in which parallel structure is explicitly expressed.

There are many well-known reasons why parallel programming is more difficult than sequential programming. The programmer must manage multiple threads of control rather than one. Primitives supplied are at a low-level and often are tied to a single vendor's machines. Perhaps even more important is that programmers need a greater understanding of large-scale structure of parallel programs than they do of sequential programs. The reason is that performance is much more greatly affected by large-scale structure in the parallel case.

In order to write fast and efficient parallel programs, programmers must fully understand issues such as what tasks make up their program, which communicate with which, the size of the communications, and the granularity of computations that occur between communications. Furthermore they must be aware of and exploit data locality. All of this is true on both shared address space and message-passing machines. These issues are grounded in the coupling between task-sized elements of programs. Hence, large-scale structure matters.

Heterogeneous environments further complicate the programming task. It is more difficult to design for performance in such systems since they are harder to model– even

*Computer Science Dept., University of Tennessee

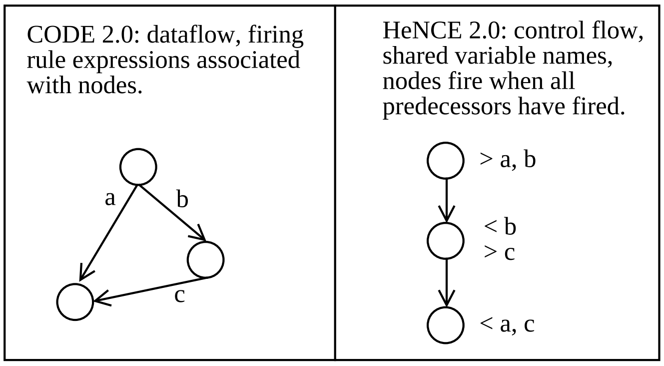


FIG. 1. *A Program in CODE and HeNCE.*

informally. Programmers are faced with multiple communications types and must use multiple tools that do not inter-operate easily. More prosaic problems also abound and are especially vexing for non-specialists. Programmers must manage builds on multiple machines and start and control tasks on them. Tools like PVM [3] solve some of the problems. Visual programming languages can solve many others.

2 Visual Parallel Programming Languages

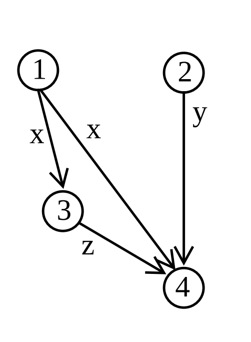
Visual programming languages (VPLs) are those which use visual or graphical elements to directly represent program structure or logic. The program is written in a language of pictures or diagrams. This stands in contrast to tools which allow users to graphically view the structure of programs written in non-visual languages and tools which are designed to assist in GUI development.

Flowcharts are an example of a sequential visual programming notation, and there are many others. There are relatively few examples of parallel visual programming languages, but many do exist. See [4] for a survey. The idea of parallel VPLs is not new. Adams [1] proposed an early dataflow language in the late 1960s. Nevertheless, parallel VPLs have not been widely adopted. This paper will discuss some of the general advantages of visual programming in parallel computing and then focus on two of the more recent and mature efforts, HeNCE [2] and CODE 2.0 [4, 5].

Parallel VPLs can take many forms in which the graphical elements have different meanings, but all are based on the idea that programs are created by drawing and then annotating a picture or diagram. The annotations are often textual. The diagram is then automatically compiled into an executable form. Figure 1 shows a simple program expressed in both CODE and HeNCE. In both programs, the circles represent sequential computations which are defined by calls to sequential subprograms in a conventional language (usually "C"). These calls are a part of the node's textual annotation in both cases.

Despite the similarities in the meaning of nodes, the graphs express the program very differently. The CODE program is a dataflow graph (although CODE also has a shared variable capability). The graph shows the creation of data by one sequential computation that is needed by another. A node may fire when its programmer-supplied firing rules are satisfied. The firing rules are part of a node's textual annotation. The simplest allow the node to fire when there are data present on all incoming arcs. The nodes in the figure have such firing rules.

The graph in the HeNCE program shows control flow and name scoping. The graph does




```
T1:                    T3:
  send(T3, x);           recv(T1, x);
  send(T4, x);               *
                             *
T2:                      send(T4, z);
  send(T4, y);
                       T4:
                         recv(T3, z);
                         recv(T1, x);
                         recv(T2, y);
```

FIG. 2. *A Dataflow Diagram.*

not show dataflow. HeNCE uses a shared name model; two or more nodes can access the same variable. Nodes may fire when all of their predecessors in the graph have fired. Access to variables is controlled by the node annotations that are shown in figure 1. The notation "> a, b" means make variables "a" and "b" available to any successor node that must access them. The notation "< b" means access variable "b" from the nearest predecessor node that makes it available by means of a ">" expression. Thus, the CODE and HeNCE graphs represent the same computation.

2.1 Advantages of Parallel VPLs

Visual programming has a number of obvious advantages for parallel programming. In order to achieve good performance, programmers must understand large-scale program structure. With visual programming languages, they directly create, view, and manipulate it.

It is easier to relate parallel program behavior to a graphical representation than it is to a textual representation because text is inherently linear and both graphical representations and parallel program behavior are inherently multi-dimensional. As a simple example, consider figure 2. The program's parallelism is obvious at a glance from the dataflow graph. The textual representation must be carefully examined to obtain the same information.

The parallel VPL permits the use of pictures and declarative annotation in place of low-level primitives such as locks and sends. It also serves as a natural basis for a programming environment in which tasks such as program editing, library use, multi-platform compiles, heterogeneous execution, and both logical and performance debugging can be performed. The programmer will be able to debug and relate runtime information (perhaps in the form of animation) directly onto the program.

2.2 Elements of a VPL Model

Visual parallel programming languages can be designed in many ways and with many goals. Some of the parameters include the meaning of all graphical symbols and their attributes and the mechanism for specifying them by annotation. One must consider issues of name scope and methods of partitioning data across parallel computational elements. The issue of supported data types is important since general pointers are not likely to be included. It is necessary to provide some alternative to pointers and heap storage such as is used in "C" to build complex custom data types. Furthermore, any reasonable programming notation should support hierarchical development. Thus, there will be issues of formal/actual parameter binding. Visual programming languages face many of the same

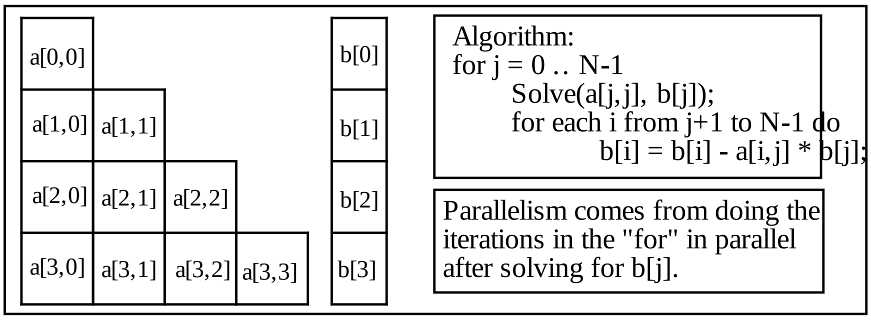


FIG. 3. *Block Triangular Solver (BTS).*

issues as textual programming languages and can benefit from lessons learned from them.

A parallel VPL will be designed with a set of goals in mind. These include the use of simple and easy to use primitives that allow concise but readable expression of algorithms. One might also want to design the language to support static analysis for issues ranging from type checking to detection of race conditions. The language must produce efficient executable structures and be implementable on a reasonable class of parallel machines. Furthermore, in order to gain acceptance, a language should inter-operate in some form with existing tools.

Clearly, many of these goals are incompatible. Language designs must compromise among competing goals. Domain-specific programming languages may allow more satisfactory decisions.

3 A Brief Introduction to CODE 2.0 and HeNCE 2.0

This section will describe and contrast a few of the choices made in the design of two implemented parallel VPLs, CODE 2.0 and HeNCE 2.0. It is not intended to be a complete introduction to either language. The purpose is to summarize some of the advantages and disadvantages involved in the two approaches.

3.1 An Example Problem

We will describe the two languages in the context of an application that is very simple but displays a parallel structure that is not completely trivial. The problem is the solution of $Ax = b$ for a dense lower triangular matrix A. As figure 3 shows, the matrix and vector must be partitioned into blocks. Thus $a[i, j]$ represents a sub-matrix. The solution overwrites vector b.

Figure 3 shows the algorithm used. It proceeds iteratively, working on one column of the block system at a time. When doing column j, it first sequentially solves for block $x[j]$ which it writes into $b[j]$. It then performs $N - j - 1$ operations in parallel. Each of these operations consists of a single iteration of the "for" loop and performs a matrix-vector multiplication and a vector-vector subtraction. Call this operation "mult."

3.2 CODE 2.0

Figure 4 shows a CODE 2.0 graph called *DoBTS* that performs this computation. Instead of attempting to fully explain how this program works, we will provide a broad overview. This graph shown is actually only a part of the whole program. Graphs in CODE are

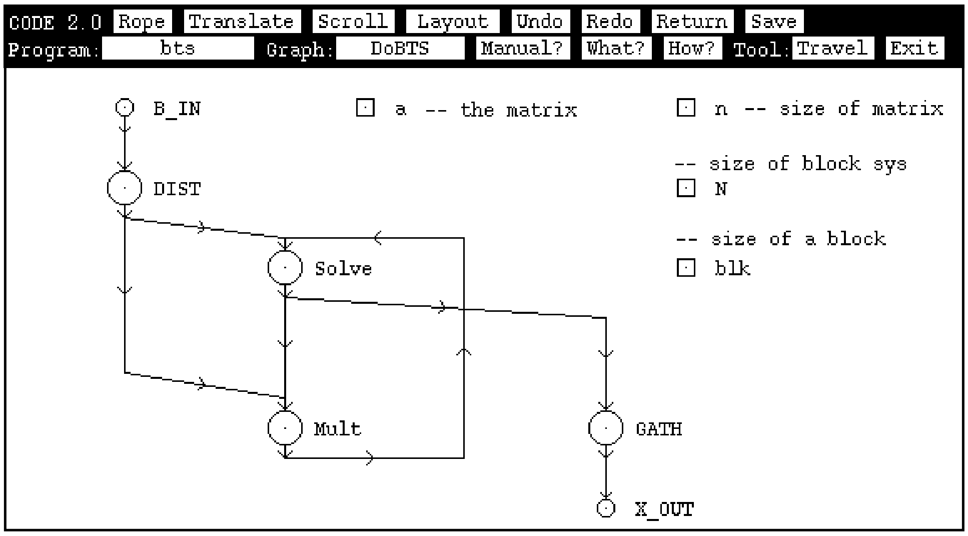


FIG. 4. *BTS in CODE.*

subprograms; they can be called from other graphs and parameters can be passed in and out. *DoBTS* is called from a "main" graph that does the program's I/O.

The small square icons in figure 4 are "creation parameters." They are given values in the calling graph and their values may be read anywhere within the graph in which they are defined. Since the matrix a is read-only, it is passed as a creation parameter. The vector b is passed in as a dataflow parameter. Its arrival enables node *DIST* to fire. This node passes appropriate blocks of b to node *Solve* and to multiple instantiations of node *Mult*. The *Solve* node then fires once for each column in the block system and computes $x[j]$ which it sends to the $N-j-1$ instances of the *Mult* nodes. Instance $j+1$ of the *Mult* nodes passes $b[j+1]$ back to *Solve* to begin the next column, and so on.

To create this graph, the programmer draws it and then annotates it. Let us examine the annotation for node *Solve*. It consists of a sequence of stanzas. All are entered by the programmer.

The first two, input_ports and output_ports, define the node's interface. Data are received into local variables and sent to other nodes on these ports. They are strongly typed; both carry data of type "Vector" in this example. Arcs interconnect ports.

```
   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;
}
```

The vars stanza is used to declare local variables that will be used by the node's computation. Values will be removed from input ports and placed into these variables.

The firing_rules stanza defines the conditions under which the node is permitted to execute. Firing rules also define which local variable will receive a value when it is removed from an input port. This example contains two firing rules. The node may execute when data arrives on either input port *B_FROM_M* or input port *B_FROM_DIST*. In both cases, the vector received is placed into local variable b. CODE's firing rule syntax is very flexible; this example is one of the simplest cases.

The comp stanza defines the sequential computation that the node will perform when it fires. *Solveblock* is a sequential subprogram written in "C."

The routing_rules stanza determines which output ports will have data placed onto them when the node's execution is complete. Indices may be used to direct data to a particular instance of the receiving node. Thus, node instances are named by indices.

3.3 HeNCE 2.0

The block triangular solver program may also be expressed in HeNCE as is shown is figure 5. This figure also shows the annotations of nodes *GetSys* and *solve*. Notice that HeNCE graphs are read from bottom to top.

The second node from the bottom begins a loop. The subgraph between this node and the loop-end node is repeated for $j = 0..N-1$. Iteration j solves for $x[j]$ and writes it into $b[j]$. The triangular nodes represent a "fan" construct. They create copies of the subgraph that they enclose. The copies are indexed from $i = j+1..N-1$ and all run in parallel.

The annotation of node *GetSys* shows variables being declared, and the node calls a "C" subprogram also called *GetSys* to assign input values to the matrix and input vector. The notation "$NEW <>$" makes these variables available to successor nodes.

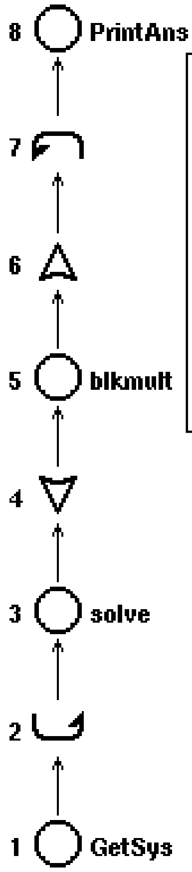
The annotation of node *solve* consists only of a call to "C" routine *solve*, but the actual parameters specify blocks of matrix a and vector b by means of index ranges.

3.4 Problems

Both the CODE and the HeNCE program have virtues, but they also have problems. CODE's notations are powerful but not concise since the user must explicitly specify firing and routing rules. Also, there are many required layers of name-binding. The programmer must bind ports between nodes and then bind ports to local variables in firing rules and often actual parameters to formal parameters in calls within the comp stanza. Also, dataflow graphs tend to be ill-structured. Structured dataflow graphs are less natural than structured control flow graphs. Finally, CODE's firing rules can become complex.

HeNCE suffers from the complexity of specifying partitions of arrays using index expressions. These are hard for programmers to write and hard for compilers to translate into efficient code since complex expressions lead to element by element copying. This problem is not fundamental. Explicit declarative array partitioning specifications could be added to HeNCE.

HeNCE also suffers from the fact that its firing rules are implicit– they are determined




```
-- Annotation of node GetSys

NEW <> int N;               -- Num blocks in sys
NEW <> double a[500][500];  -- Matrix
NEW <> double b[500];       -- Vector
NEW <> int blk;             -- Num elmts in block
NEW <> int n;               --Size of a
GetSys(a, b, &n, &N, &blk);

-- 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);
```

Fig. 5. *BTS in HeNCE.*

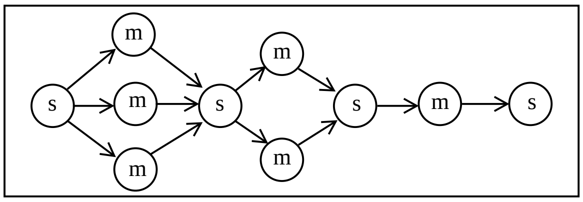


FIG. 6. *BTS Communication Pattern.*

by the control flow structures in which the node is embedded. For example, HeNCE lacks an effective mechanism to exit a loop on an error condition. Its method of specifying firing rules is simple but not powerful. Also, HeNCE completely fails to graphically display communications between nodes. The graph shows only control flow.

Both HeNCE and CODE fail to effectively display replicated structures. In both programs, the entire parallelism in the program is among multiple indexed replications of the array multiply node. Both languages display that node as a single instance. HeNCE programs, in fact, tend to be linear because runtime replication is ubiquitous. The general superiority of HeNCE's notation over (also linear) text is unclear.

Visual representation of nodes that are replicated at runtime under the control of runtime variables is a difficult problem to solve in a manner that is concise, powerful, and unambiguous. Perhaps it is more practical to address this issue by means of animation of runtime behavior. Figure 6 shows a possible display.

4 Processes as Components

Both CODE and HeNCE adopt the semantics that the execution of primitive sequential computation nodes can be viewed as being an atomic mapping of inputs to outputs. There are many advantages to this approach. It is simple and permits implementation with non-preemptive scheduling. Also state changes occur at well-defined points. This facilitates the implementation of fault tolerance, process migration, etc.

However, this model (as CODE in particular shows) can lead to complex firing conditions that are not explicitly present in traditional message-passing process models. Control flow within a process implicitly defines firing rules in a manner that distributes the conditions throughout the program at conditional branch points. The issue can be clarified by a small example of making firing conditions explicit. As the control flow becomes more complex and deeply nested, the firing rules become more complex.

```
       Implicit                                Explicit

if (A) {
   if (B) send(T1, x);           A && B --> send(T1, x);
}  else send(T1, y);             ~A     --> send(T1, y);
```

Such implicit firing rules are both a blessing since they are powerful and concise and a curse since they obscure large-scale structure. Implicit firing rules can be used in a visual parallel programming language by defining the basic graphical element to be a process as shown in figure 7. This program also implements the block triangular solver.

In this proposed language, the user creates a program by drawing a picture that shows processes. Boxes are annotated with the text of the process as well as other information such

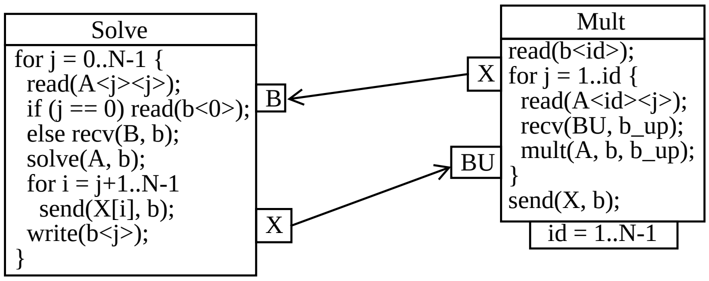


FIG. 7. *BTS as a Pair of Processes.*

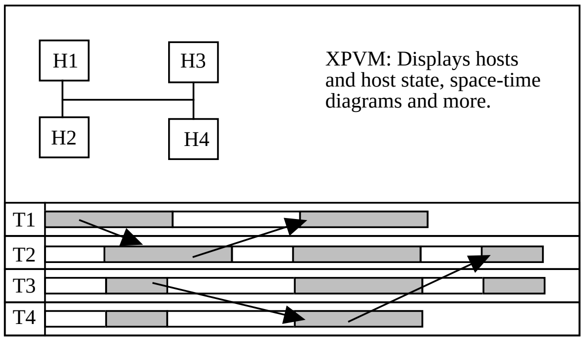


FIG. 8. *XPVM Window (Drawing).*

as replication parameters and the architecture type the process is to be executed on. Also, each process has an interface (much like CODE ports) that allows processes to be defined independently of other processes. Thus, processes can be reused as library components. Arcs interconnect these ports, as in CODE.

5 Integration with XPVM

Since we intend to implement visual parallel programming languages using the PVM system, the proposed process-oriented language has some special virtues. It is more in the spirit of PVM than CODE or HeNCE and can likely be integrated with existing PVM tools such as XPVM (figure 8).

XPVM is an X Windows based environment for controlling and visualizing PVM programs. It was developed by Jim Kohl at Oak Ridge National Laboratory. XPVM provides facilities for defining and viewing the state of the machines (usually workstations on a LAN) that make up a PVM virtual parallel machine. It permits users to run PVM parallel programs and then animates their execution by means of space-time diagrams and other displays. It also captures and can replay trace information via animation.

XPVM contains many of the elements of a visual programming environment. A process-based visual parallel programming language could be integrated with it. Also support for automatically compiling executables on all machine types could be added.

XPVM could then animate the program directly by relating trace information directly to the program as drawn by the user. Messages could be shown traveling on arcs, and nodes that are instantiated more than once could be drawn multiple times. Also, direct debugger support could also be added. Our current plans are to pursue this approach.

References

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

[2] 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.

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

[4] 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.

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