HeNCE: A Heterogeneous Network Computing Environment

Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, and Keith Moore

Computer Science Department

CS-93-205

August 1993

HeNCE: A Heterogeneous Network Computing Environment *

Adam Louis Beguelin (adamb@cs.cmu.edu)
School of Computer Science and Pittsburgh Supercomputing Center
Carnegie Mellon University
Pittsburgh, PA 15213

Jack. J. Dongarra (dongarra@cs.utk.edu)
University of Tennessee and Oak Ridge National Laboratory

George Al Geist (geist@msr.epm.ornl.gov)
Oak Ridge National Laboratory

Robert Manchek (manchek@cs.utk.edu)
Keith Moore (moore@cs.utk.edu)
University of Tennessee

August 27, 1993

Abstract

Network computing seeks to utilize the aggregate resources of many networked computers to solve a single problem. In so doing it is often possible to obtain supercomputer performance from an inexpensive local area network. The drawback is that network computing is complicated and error prone when done by hand, especially if the computers have different operating systems and data formats and are thus heterogeneous.

HeNCE (Heterogeneous Network Computing Environment) is an integrated graphical environment for creating and running parallel programs over a heterogeneous collection of computers. It is built on a lower level package called PVM. The HeNCE philosophy of parallel programming is to have the programmer graphically specify the parallelism of a computation and to automate, as much as possible, the tasks of writing, compiling, executing, debugging, and tracing the network computation. Key to HeNCE is a graphical language based on directed graphs that describe the parallelism and data dependencies of an

*This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract DE-AC05-84OR21400 and the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615.

application. Nodes in the graphs represent conventional Fortran or C subroutines and the arcs represent data and control flow.

This paper describes the the present state of HeNCE, its capabilities, limitations, and areas of future research.

Index Terms:

graph models,

heterogeneous machines,

parallel computing,

programming environments,

programming languages,

visual programming.

1 Introduction

Computer networks have become a basic part of today's computing infrastructure. These networks connect a wide variety of machines, presenting an enormous computing resource. The HeNCE project focuses on developing methods and tools which allow a programmer to tap into this resource. In this paper we describe a tool and methodology which allows a programmer to write a single program and execute it in parallel on a networked group of heterogeneous machines.

The HeNCE programmer explicitly specifies the parallelism of a computation and the environment aids the programmer in designing, compiling, executing, debugging, and analyzing the parallel computation. HeNCE provides programming support through a graphical interface which the programmer uses to perform various tasks. HeNCE supports visual representations of many of its functions.

In HeNCE the programmer specifies directed graphs where the nodes in the graphs represent either procedures or special types of control flow and the edges denote dependencies. These program graphs are input by the programmer using a the HeNCE graph editor. (There is also a textual interface allowing a programmer to create a program graph using a conventional text editor.) The procedures represented by the nodes of the graph are written in either C or Fortran. In many cases these procedures can be taken from existing code. This ability of software reuse is a great advantage of HeNCE. The environment provides facilities for editing and compiling these procedures on the various architectures of the user defined collections of computers, also called a virtual machine.

A unique capability of HeNCE is the ability for the user to specify multiple implementations of HeNCE nodes based on the target architecture. If a node in the HeNCE graph is executed on a Cray then code written explicitly for that machine can be used. However, if the node is executed on a distributed memory multicomputer then another algorithm may be used which has been tuned for that architecture. During execution of a HeNCE program, HeNCE dynamically chooses which machine executes a node. Scheduling choices are based on programmer input and the current HeNCE related load on a machine. This combination of graph specification and multiply defined node implementation is a departure from current single program multiple data (SPMD) approaches to massively parallel programming systems.

Once a HeNCE program graph has been written and compiled it can then be executed on the user's virtual machine. Using HeNCE the programmer specifies the various machines on which the program is to run. Each machine is specified by its internet name. This collection of machines can be widely varying. A target machine could be anything from a single processor workstation to a 64K processor CM-2. Because of the nature of the interconnection network, HeNCE procedures are intended to be large grain. The programmer may also specify a cost matrix showing the relative costs of running procedures on various architectures. HeNCE will automatically schedule the program procedures on particular machines based on this user defined cost matrix and program graph.

As the program is executing, HeNCE can graphically display an animated view of its program's state based on the computational graph. Various statistics recorded during program execution may also be stored and animated post mortem. Debugging support is available. Beyond the post mortem analysis, HeNCE can execute shells and other debugging tools on user specific machines.

HeNCE is implemented on top of a system called PVM (Parallel Virtual Machine) [23, 5, 4]. PVM is a software package that allows the utilization of a heterogeneous network of parallel and serial computers as a single computational resource. PVM provides facilities for spawning, communication, and synchronization of processes over a network of heterogeneous machines. PVM differs from HeNCE in that it provides the low level tools for implementing parallel programs, while HeNCE provides the programmer with a higher level abstraction for specifying parallelism and a comprehensive environment for developing, running, and tuning programs.

In this paper we will expand upon the points introduced here. Section 2 presents an in depth explanation of the HeNCE paradigm and how procedures and graphs are interfaced. In section 3 the environment's major features are presented along with screen dumps showing HeNCE in action. We compare HeNCE to related work in Section 4. In Sections 5 and 6 we discuss some future directions for this work and provide pointers for retrieving the software. Finally in Section 7 a summary is given of the project's current status and the directions of our continuing work.

2 The HeNCE Paradigm

In developing software, the initial definitions and specifications are often performed graphically; flow charts or dependency graphs are well known examples. The overall structure of the problem can be visualized far more easily from these graphical representations than from textual specifications; from the development point of view, such a representation enhances the quality of the resulting software. However, in order to be executed, such a description must be converted to program form, typically manifested as source code. These graphical representations therefore, must eventually be translated to operational programs — the graphical depiction of a concurrent application, and strategies for its successful execution on a heterogeneous network are the two fundamental inputs to the HeNCE environment.

With the use of a graphics interface, implemented on a workstation for example, a user can develop the parallel program as a computational graph, where the nodes in the graph represent the computation to be performed and the arcs represent the dependencies between the computations. From this graphical representation, a lower-level portable program can be generated, which when executed will perform the computations specified by the graph in an order consistent with the dependencies specified. This programming environment allows for a high-level description of the parallel algorithm and, when the high-level description is translated into a common programming language, permits portable program execution. This environment presents the algorithm developer with an abstract model of computation that can bind effectively to a wide variety of existing parallel processors. Specific machine intrinsics may be confined to the internal workings of such a tool in order to provide a common user interface to these parallel processors.

Another problem facing the developers of algorithms and software for parallel computers is the analysis of the performance of the resulting programs. Often performance bugs are far more difficult to detect and overcome than the synchronization and data dependency bugs normally associated with parallel programs. We have developed a fairly sophisticated postprocessing performance analysis tool associated with the graphics programming interface just described. (See section 3.4.) This tool is quite useful in understanding the flow of execution and processor utilization within a parallel program.

In HeNCE, the programmer explicitly specifies a computation's parallelism by drawing a graph which expresses that parallelism. HeNCE graphs provide a usable yet flexible way for the programmer to specify parallelism. The user directly inputs the graph using the HeNCE compose tool. HeNCE graphs consist of *subroutine* nodes which represent subroutines (written in Fortran or C) and *special* nodes which denote four different types control flow: conditionals, loops, fans, and pipes. Arcs between subroutine nodes represent dependencies. It is the user's responsibility to specify these dependencies. There exist tools such as Parascope [1] and Forge90 [19] which will take can help the programmer discover parallelism in a sequential code. HeNCE helps the user describe and run a parallel program but it does not find parallelism for the programmer. For instance if there is an arc from node a to node b then node a must complete execution before node b may begin execution. During the execution of a HeNCE graph, procedures are automatically executed when their predecessors, as defined by the graph, have completed. Functions are mapped to machines based on a user defined cost matrix. The costs are also used as an indication of machine load. The sum of the costs of the functions currently mapped to a machine constitute a machine's HeNCE originated load. This load is considered when mapping a new function to a machine.

Figure 1 shows a simple HeNCE graph containing only dependency arcs. This graph represents a fractal computation. The *start* node reads input parameters denoting which part of the complex plane the computation should cover. After the *start* node completes, the dependency arcs from it to the *tile* nodes are satisfied and they may begin execution, computing the pixel tile assigned to them via the parameter list.

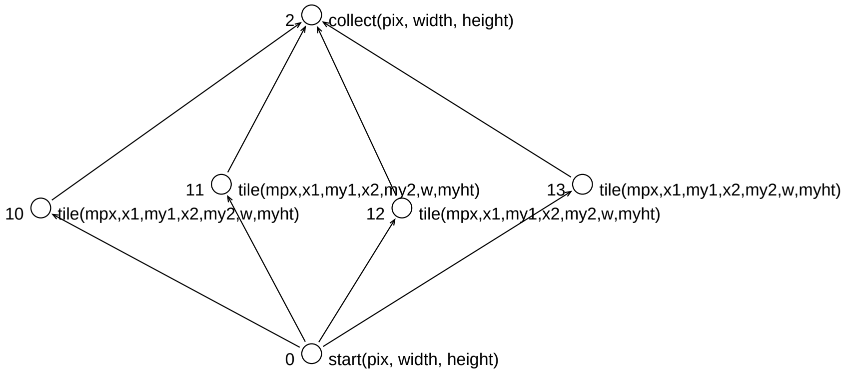

Figure 1: Simple HeNCE graph with dependency arcs.

In this case they are invoked on different parts of the complex plane. Once all of the compute nodes are finished the *collect* node can execute and display the resulting fractal.

When a subroutine node executes, its parameters are taken from its immediate predecessors. Parameter matching is done on the named variables in the parameter list attached to a subroutine node. If two or more predecessors have a node's parameter then one of the parameters is chosen randomly. If a parameter does not exist among the predecessors to a node, then the program will deadlock. When these conditions can be checked a priori they will produce warnings.

Next, we use an integrate program to demonstrate the dynamic behavior of HeNCE graphs. Figure 2 shows the graph for computing the integral of a function. This program uses the rectangular rule for computing the integral. Attached to node 1 is the *init* function. *init* will initialize the array x with the values at which $f(x)$ will be evaluated for integration. The parameters max_parts and num_procs indicate the number of maximum number of x values and the number of processes to be used in the parallel computation.

The triangular nodes 5 and 6 are special nodes which specify that node 4 will be replicated dynamically at runtime. The statement attached to node 5 specifies that node 4 will be replicated $num\_procs + 1$ times. Each copy of node 4 will compute a partial integral of the function f on the interval from $x1$ to $x2$. In this example only node 4 is replicated. It is possible that an entire subgraph be replicated at runtime.

Besides strict dependencies and dynamic constructs, HeNCE also supports pipelining, looping, and conditionals. Each of these constructs are used in a manner similar to those of the fan construct. Begin and end nodes mark the subgraph which is affected.

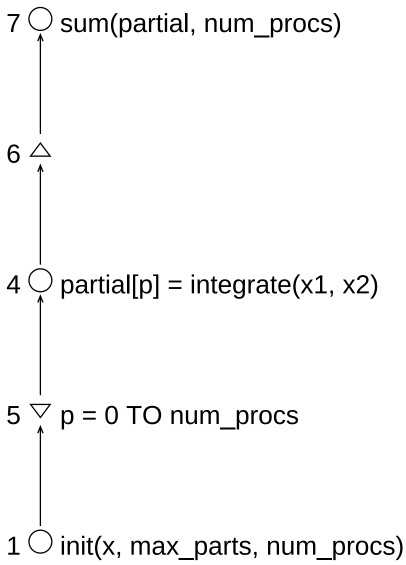

Figure 2: HeNCE program for numerical integration.

HeNCE nodes are stateless. Input and output from these nodes occurs only via the parameter list. In this way HeNCE is very similar to programming in a sequential language with no global variables. HeNCE does not prohibit the node from creating external *state*, for instance in the form of files. However, internal state is not preserved beyond the node's parameter list. For example, if a node dynamically allocates storage, this storage will not be accessible to other nodes in the system nor will it be accessible to future invocations of the same node. It is in this sense that HeNCE nodes are stateless.

HeNCE nodes may access files. However, HeNCE makes no special provisions for file access. Since nodes are mapped to machines at runtime, any files would need to be accessible from the target machines. If all the machines are using the same file system via NFS or AFS then files will be globally accessible and thus not a problem. If a global file system is not available the programmer can control the mapping of nodes to machines via the HeNCE cost matrix described in section 3.2.

In general, any combination of constructs may be properly nested. When expressions are evaluated or subroutines called, their parameters always comes from their predecessors in the graph.

2.1 Parameter Passing in HeNCE

The subroutine are attached to nodes in a HeNCE graph simply by specifying the subroutine call as in conventional Fortran or C. This approach to automatically passing parameters makes the HeNCE programs easy to build from pieces of code that have already been written. Re-usability is enhanced.

When a procedure is attached to a particular node, the parameters to be used when the node is invoked are also specified. Parameters can be input, output, or input/output. Input parameters take their values from ancestors in the graph. Output parameters pass their values on to descendents in the graph. Scalar, vector, and two dimensional arrays of all base types are supported. For non-scalar parameters, the row and column sizes must be input. If only a certain subarray is needed, only the upper and lower bounds for the subarray need to be specified. For instance,

```
NEW <> float local_x[a] = x[0:a-1];
```

specifies that a new local variable called *local_x* will contain the values of the variable x in the index range 0 to $a - 1$.

2.2 Heterogeneous Function Implementation

Another strength of HeNCE is its ability to tie together different architectures into a virtual parallel supercomputer. In fact, the machines that make up this virtual parallel supercomputer may themselves be parallel supercomputers. We have shown that HeNCE graphs describe the parallelism of a set of procedures. Each procedure in a HeNCE graph may be implemented by several algorithms. For instance, computing the LU decomposition of a matrix on an Intel iPSC/860 is much different from computing the same procedure on a Cray C90. HeNCE supports the specification of different procedure implementations and invokes the appropriate implementation when a procedure is executed on a particular machine. Thus a complete HeNCE program consists of a set of procedures represented by nodes in the program graph and, for each procedure, a set of implementations of that procedure for various architectures. HeNCE provides mechanisms for specifying which source files implement which procedures for which architectures. Facilities for compiling source files on various architectures are also supported. In the future we hope to add facilities which visually repre-

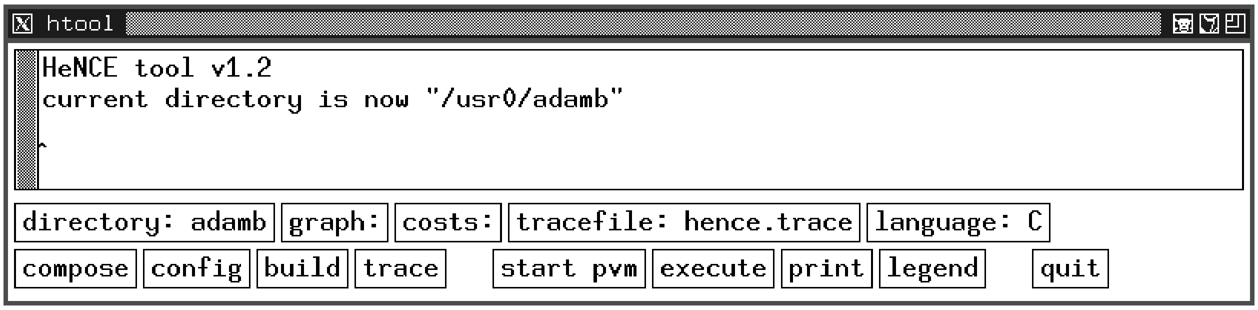

Figure 3: HeNCE main menu.

sent the status of a user's virtual machine with respect to which machines are able to execute which HeNCE nodes.

3 The HeNCE Tool

HeNCE provides an X window based user interface for using HeNCE facilities. This interface aids the user in creating HeNCE graphs, configuring virtual machines, visualizing trace information, compiling, executing, and debugging HeNCE programs. Although the graphical front end makes the use of these facilities easier, for portability reasons it is still possible to utilize HeNCE without it's graphical interface. All essential HeNCE facilities have textual counterparts.

Figure 3 shows HeNCE's main menu. The upper part of the window is a scrollable window that displays various HeNCE related information. The directory button allows the user to change directories. The graph:, costs:, and tracefile: buttons allow the user to load the associated files. The language: button specifies whether the node subroutines will be written in C or Fortran. The compose, config, build, and trace buttons are for entering those related modes of HeNCE operation. In compose mode the user draws a program graph. Config mode allows the user to configure a virtual machine, specify the hosts involved in executing the distributed program. Build mode will compile node programs for execution on the virtual machine. Trace allows the programmer to animate aspects of program execution. The start pvm button will start executing PVM on the user specified virtual machine. Execute runs the current HeNCE program on the virtual machine. Finally, the print prints a copy of the HeNCE graph and legend displays a useful legend of HeNCE symbols.

3.1 Compose

The compose mode of HeNCE allows the user to interactively create HeNCE graphs. Figure 4 shows HeNCE in compose mode. Various nodes may be drawn and connected together with arcs. For each procedure node in the graph the user must specify a procedure which will be called when that node's dependencies have been satisfied. For control flow constructs, loops and so on, the required expressions and variables must be specified. When a procedure is attached to a node the actual parameters are specified. From compose mode the user can also open an edit window for a particular node's subroutine source code.

3.2 Configure a Virtual Machine

In order to compile and execute a HeNCE program, a set of machines must be specified. In HeNCE this is called configuring a virtual machine. Any machine where the user has a login and accessibility via a network can be used. HeNCE makes use of PVM for most functions. If the PVM daemon is not running on a machine, then HeNCE can start it. The user may add hosts to the virtual machine simply by supplying an internet name or address. HeNCE can be customized via X resources to handle different login names on different machines. (X resources are variables supported by the X windowing environment.)

In configure mode the user may also specify costs for running nodes on different architectures. For a particular node, costs are ordered pairs of machines and integers. The higher the integer, the more expensive it is to run that node on that host. HeNCE will use these costs when mapping a procedure to a machine at run time. Figure 5 shows HeNCE in configure mode where the cost and configure information is input by the user.

The costs are used at runtime when mapping nodes to machines. These costs reflect the relative runtimes for HeNCE nodes executing on various machines. The user should estimate these costs based on what he or she knows of the machines ability to execute a HeNCE node. This should reflect not only the type of machine but it's expected load. For instance, an unloaded DEC Alpha workstation may be faster than a heavily loaded Cray C90. In this case it may make sense for a node to have a lower cost on the Alpha than the Cray. HeNCE keeps track of where nodes are executing. Using the cost matrix, HeNCE can decide on the least costly process placement. This placement only takes into account HeNCE processes, ignoring other

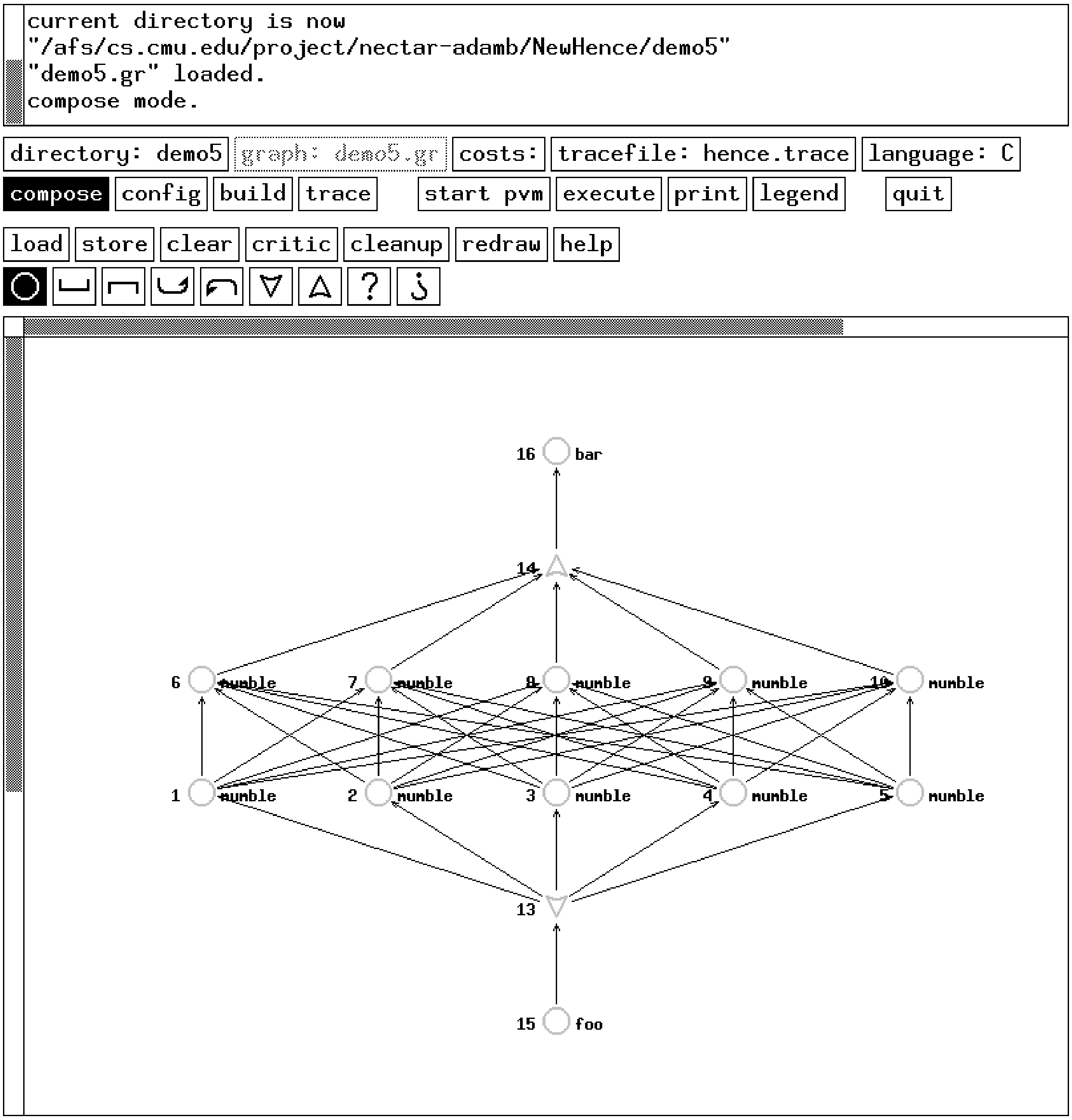

Figure 4: HeNCE in compose mode.

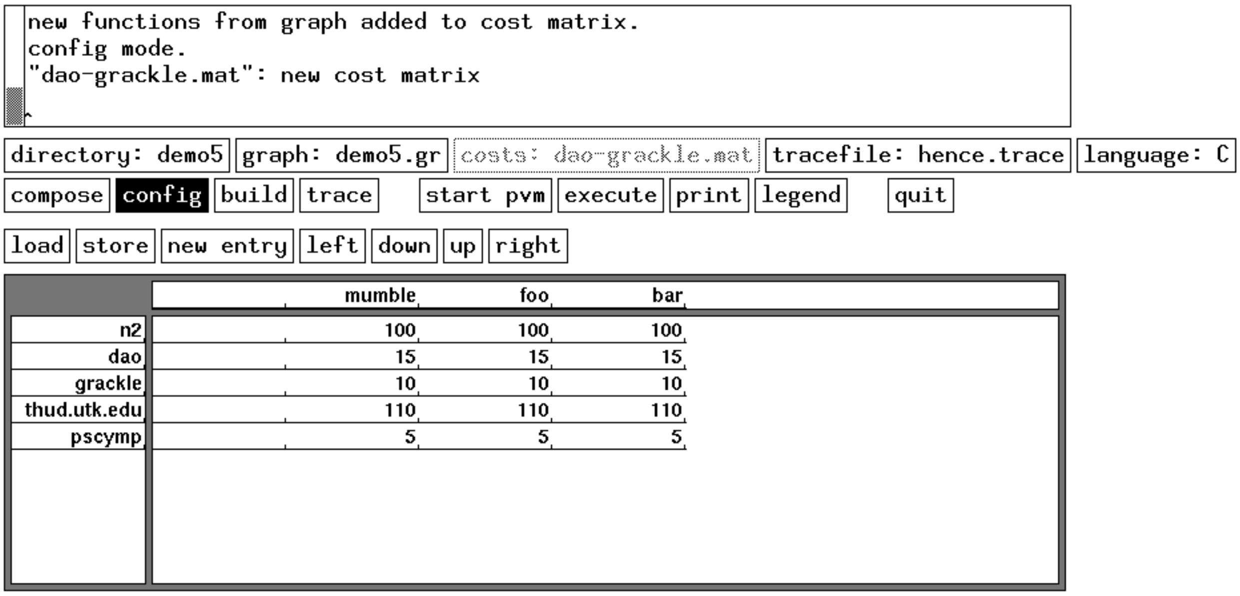

Figure 5: HeNCE in configure mode.

external load factors. We are exploring more comprehensive methods of load balancing. The advantage of this method is that it is simple and the user can easily tune a set of costs for a particular combination of application and virtual machine.

While a HeNCE program is executing, the machine configuration may not be changed by the user. However, between program executions machines may be added and removed at will. Machine configurations can be stored in files for easy retrieval. For instance, a different cost matrix may be used late in the evening rather than during the daytime when certain machines are more heavily loaded.

3.3 Compiling for a Virtual Machine

Each HeNCE node must be ultimately implemented by a binary executable file. The build mode of HeNCE supports compiling HeNCE nodes for different architectures in the the user's virtual machine. (See Figure 6.)

For each subroutine in the HeNCE graph a wrapper is automatically generated when the $\boxed{\text{write wrappers}}$ button is pressed. This wrapper code is linked into user subroutines. The wrappers handle the spawning, synchronization and communication necessary to execute a HeNCE program. Pressing the $\boxed{\text{write makefile}}$ button causes a custom makefile to be generated for compiling the program's source code. Specialized

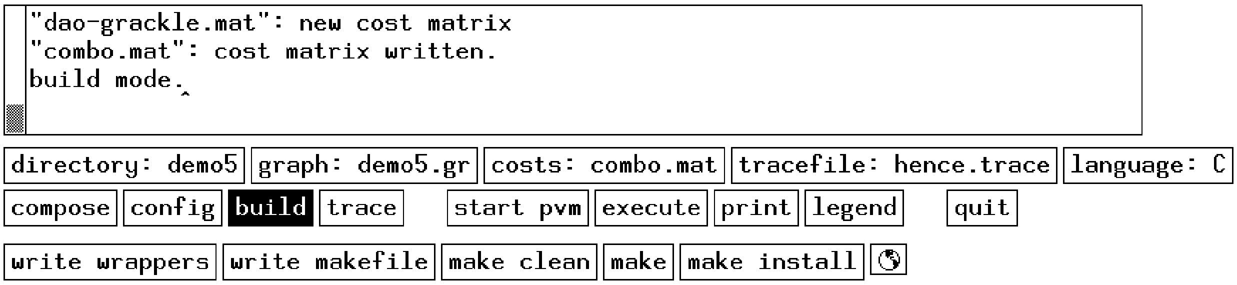

Figure 6: HeNCE build mode.

libraries (i.e. Xlib) can also be included in the makefile by setting the correct X resource.

Make can be invoked by clicking $\boxed{\text{make clean}}$, $\boxed{\text{make}}$, or $\boxed{\text{make install}}$. HeNCE will carry out the make operation locally or on all machines in the current cost matrix depending on the local/world toggle. In Figure 6 this toggle, just to the right of the $\boxed{\text{make install}}$ button, is set to world. When compiling on multiple machines, HeNCE uses a program called pvmrsh for carrying out the remote makes. This global make handles cases when the user's home directory is and is not cross mounted.

3.4 Visual Execution Analysis Tools

HeNCE provides tools for visualizing the performance of HeNCE programs. HeNCE can emit trace information during a program's execution. This trace information may be either displayed as the program is running or stored to be displayed later. Figure 7 shows trace mode in action. Trace information is displayed as an animation sequence. Replay of this sequence can be controlled by the user; rewind, stop, single step, and continuous play modes are supported.

The main window shows the graph, drawn using different shapes and colors to indicate the graph's state of execution. These shapes and colors are explained in the legend window. In Figure 7 the `dprod` node is actually inside a fan construct. This means there may be several copies of this node executing. The `2r0i` annotation signifies there are two copies of this node running and zero copies idle. The Host Map window displays icons for each machine in the virtual machine. These icons change color indicating that host is either performing housekeeping duties or executing a user's subroutines. The icons for burgundy and concord are also annotated with `3/0` and `3/1` indicating instances 0 and 1 of node 3 are running on these machines.

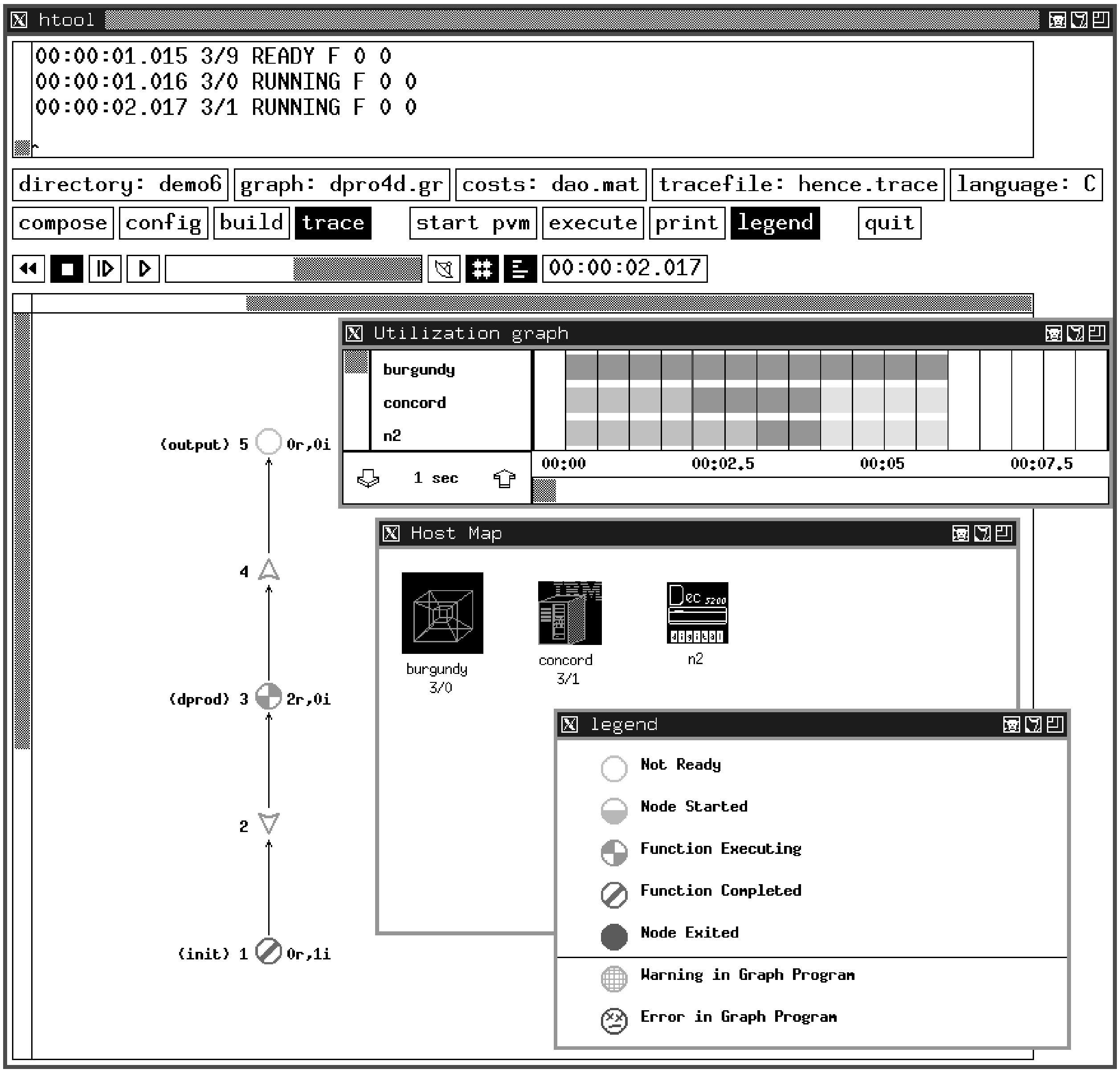

Figure 7: Tracing a program in HeNCE.

Finally, the utilization graph shows the state of the machines over time. The red line indicates the location of the current time in the animation sequence. The time scale of the utilization graph can be changed by clicking on the arrows.

Tracing is optional and may be turned off for efficiency during production runs of a program. However, if the granularity of the computation is coarse enough then the trace events will not add significant overhead.

3.5 Running a HeNCE Program

Once a virtual machine has been configured and the subroutine nodes have been compiled, HeNCE can execute the HeNCE graph. First PVM must be started on the configured virtual machine. This can be done by clicking the $\boxed{\text{start pvm}}$ button. Clicking this button starts PVM and opens a console window for interactively controlling the virtual machine. PVM console commands such as `reset` and `ps` can be issued in this window. The $\boxed{\text{execute}}$ button starts the HeNCE program on the currently configured virtual machine. During execution standard output and error are directed to the PVM console window for the duration of the program's execution. During execution the programs progress can be monitored in trace mode by clicking on the monitor button (which looks like a satellite dish) and then the $\boxed{\text{play}}$ button.

4 Related Work

Several research projects have goals which overlap with those of HeNCE. Paralex, Schedule, Phred, Code, the Cosmic environment, Network Linda, Isis, and Express are a few examples [3, 15, 7, 12, 22, 13, 10, 16].

Paralex [2, 3] is probably the most closely related project to HeNCE. In Paralex, the programmer also explicitly specifies dependency graphs where the nodes of the graph are subroutines. Paralex programs also execute on a heterogeneous distributed network of computers. There are, however, several major differences between Paralex and HeNCE. HeNCE requires less system specific code than does Paralex to interface subroutines to the graph. The HeNCE programmer must only specify the types and sizes of the elements in a procedure's parameter list. HeNCE automatically ensures parameters are available for a procedure when it is called. In Paralex, a procedure is only allowed to output one value. Although this value is not limited to scalars it is the programmer's responsibility to code "filter" nodes which partition procedure output for subsequent nodes in the Paralex graph. The requirement of user-defined graph-specific code impairs the portability and reuse of Paralex programs. The Paralex filter nodes appear to need specific knowledge of the graph coded into them at a low level. HeNCE's traditional parameter list approach allows for greater portability and reuse of code. HeNCE graphs are richer than those of Paralex. HeNCE provides conditional, loop, fan in/out, and pipeline constructs within its graphs. Paralex graphs do not provide these constructs. Pipelining is provided in Paralex but at the level of the entire graph. An advantage of Paralex over HeNCE is it's support of fault tolerance. Fault tolerance in Paralex is provided by the user specifying the number of times a node may fail. The current version of HeNCE does not support fault tolerance. The description of future work in Section 5 discusses fault tolerance for HeNCE.

Paralex is built on Isis [10]. Isis is a parallel programming toolkit for fault tolerant parallel computing over a network of heterogeneous machines. Compared to Isis, HeNCE is a higher level programming tool. HeNCE could be built on Isis rather than PVM. PVM was chosen for several reasons. Isis is a much larger system. A goal of HeNCE is to allow the use machines where one simply has a login. Isis alone requires on the order of tens of megabytes of disk storage and a system administrator to install. Comparatively, PVM is a much smaller system, requiring the order of a megabyte of disk storage and PVM can be easily installed by a user. The main difference between PVM and Isis is that Isis provides fault tolerance and more complicated process communication and control. Research into adding fault tolerance to PVM is currently underway [18].

Code [12, 21] is also a graph based parallel programming system. It allows the users to specify a parallel computation using Unified Computation Graphs [11]. Code has more advanced rules for node execution. Code firing rules are akin to guards used in logic programming.

Express [16] supports parallel programming approximately at the same level as PVM. The programmer writes explicit parallel code which makes calls to the Express libraries for process control and message passing.

The Cosmic environment [22] is a publicly available programming environment targeted toward tightly coupled homogeneous groups of local memory MIMD machines or multicomputers. The Cosmic environment is a lower level parallel programming tool than HeNCE. If HeNCE were targeted for a multicomputer environment then it would be possible to implement HeNCE on top of the Cosmic environment. Similar to Express and PVM the Cosmic environment provides a low level infrastructure for process control and communication. Unlike PVM the cosmic environment is not targeted toward heterogeneous networks of machines.

Network Linda is a commercial implementation of the Linda primitives [13] which runs over a network of processors. Network Linda does not support the heterogeneous data formats automatically; it will, however, support a Linda tuple space over a network of machines which conform to the same data formats. To effectively use Network Linda the programmer must explicitly write programs which use Linda primitives. This is similar to writing a program at the PVM level where process initialization and communication is explicit. This contrasts to HeNCE where the programmer specifies the high level synchronization and standard parameter list procedure invocation is handled automatically.

Piranha [17] is a system which is built on top of network Linda. Programming in Piranha is similar to Linda programming except the Piranha tasks migrate around the network. A major goal of Piranha is to consume unused compute cycles without disturbing machines which are in use. Piranha monitors machine utilization and will migrate tasks off of machines which are being used.

Condor [20] is similar to Piranha in it's goals. A major difference between Condor and Piranha is that Condor runs single threaded applications and Piranha applications are typically multithreaded. An advantage of Condor is that programs can utilize the system without changing any source. Piranha programs need to be explicitly written to be used with the system and a retreat function, to be called when a task is to migrate, must be provided by the programmer. We are currently exploring the use of Condor with PVM programs. The main challenge here is to provide efficient checkpointing for PVM programs.

Schedule [14, 15, 9] is similar to HeNCE. Although HeNCE graphs are more complex than those of Schedule, the basic HeNCE dependency graphs are equivalent. Schedule runs on a shared memory multiprocessor, not a heterogeneous network of distributed memory machines. Schedule programs also rely on shared memory semantics which are unavailable in HeNCE, since it is intended for distributed memory architectures. However, a HeNCE node that executes on a shared memory machine may take advantage of the available shared memory. If fact, a HeNCE node executing on a shared memory machine could actually utilize the Schedule primitives.

Phred [6, 7, 8] is also similar to HeNCE. Phred graphs are more complicated than those of HeNCE; they contain separate data and control flow graph constructs. The pipe and fan constructs of HeNCE are based on similar constructs from Phred. However, the emphasis of Phred is not heterogeneity but determinacy.

5 Future Work

While HeNCE supports many facets of the programming task, it is still deficient in several areas, mainly debugging, fault tolerance and efficiency.

The HeNCE trace mode does a good job with respect to tuning a program but more debugging support is needed. Support for source level debugging in HeNCE would be helpful. Issues with respect to parallelism and debugging need to be addressed. It is not sufficient to simply open a window for each node in the graph running dbx. This would become unwieldy all too soon. HeNCE could support the examination and alteration of node subroutine parameter lists during runtime. These parameter lists may be examined either before or after a subroutine executes.

Currently fault tolerance is not supported. As the number of computers involved in a computation grows, so does the chance that any one machine may fail. The addition of checkpointing to HeNCE is very attractive way of dealing with failures. The state of a HeNCE node can be described by the parameter list it was sent. If a node fails and this parameter list is stored, then a new version of the node can be started. Version 3 of PVM supports dynamic configurations of computers in the virtual machine. HeNCE can take advantage of this feature by adding new machines when one fails.

Once the fault tolerance support is added, it would be interesting to extend the system to migrate tasks off of machines under certain circumstances. If a workstation owner arrives and starts using his system, HeNCE could migrate that task by killing it and restarting it on another machine. HeNCE could also place or move processes based on other load factors. The cost matrix would need to be modified so the user could supply information about how to treat specific machines.

The execution of HeNCE programs is carried out by a centralized controller. This leads to efficiency problems, mainly due to bottlenecks. It would be beneficial to explore a more efficient execution scheme for HeNCE programs which eliminated much of this support on a central locus of control.

6 Availability

An underlying philosophy of our work has been to emphasize practicality; therefore, our work is oriented towards software systems that are realizable and immediately applicable to real-life problems. The software produced is freely distributed to researchers and educators, allowing them to harness their distributed compute power into comprehensive virtual machines.

At least one Unix workstation with X-Windows is required to use HeNCE. HeNCE assumes PVM is installed. HeNCE will work with either version 2 or 3 of PVM.

PVM and HeNCE are available by sending electronic mail to netlib@ornl.gov containing the line "send index from pvm" or "send index from hence". For version 3 of PVM use "send index from pvm3". Instructions on how to receive the various parts of the PVM and HeNCE systems will be sent by return mail.

7 Summary

HeNCE is an integrated graphical environment designed to simplify the task of writing, compiling, running, debugging, and analyzing programs on a heterogeneous network. In HeNCE the user explicitly specifies parallelism by drawing a graph of the parallel application and indicating data dependencies. HeNCE then writes the necessary message passing code (using PVM) and compiles this code on the machines the user has requested in the HeNCE configuration. When HeNCE executes an application graph HeNCE dynamically load balances the parallel tasks taking into account the heterogeneity in the algorithm and the machine performance. During execution HeNCE collects trace information which can be displayed graphically in real time or saved for replay. The drawing, writing, compiling, executing, and tracing steps are all integrated into a single X-Window environment.

Future research in HeNCE will focus mainly on debugging, fault tolerance, and efficiency. Source level debugging is a useful feature not presently integrated into HeNCE. As the number of computers added to a configuration grows, so does the need for more robust fault tolerance. The nature of HeNCE allows for the straightforward incorporation of fault tolerance at the application level. The efficiency of program execution under HeNCE can be improved and future work will look at better algorithms and protocols to facilitate this.

8 Acknowledgements

We would like to acknowledge the efforts of Vaidy Sunderam and James Plank. Many intense discussions with both of these scientists helped shape this research. Vaidy can also be credited with coming up with the acronym HeNCE. James Plank developed the initial version of the execution system for dynamic HeNCE graphs.

References

[1] J. R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execition. In *Conf. Rec. 14th ACM Sym. Principles of Programming Langues (POPL)*, volume 14, pages 63–76, 1987.

[2] Ozalp Babaoglu, Lorenzo Alvisi, Alessandro Amoroso, and Renzo Davoli. Paralex: An environment for parallel programming in distributed systems. Technical Report UB-LCS-91-01, University of Bologna, Department of Mathematics, Piazza Porta S. Donato, 5, 40127 Bologna, Italy, Febuary 1991.

[3] Ozlap Babaoglu, Lorenzo Alvisi, Alessandro Amoroso, Renzo Davoli, and Luigi Alberto Giachini. Paralex: An environment for parallel programming in distributed systems. In *1992 International Conference on Supercomputing*, pages 178–187. ACM, ACM Press, July 1992.

[4] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam. Solving computational grand challenges using a network of heterogeneous supercomputers. In Jack Dongarra, Ken Kennedy,

Paul Messina, Danny C. Sorensen, and Robert G. Voigt, editors, *Proceedings of Fifth SIAM Conference on Parallel Processing for Scientific Computing*, pages 596–601, Philadelphia, 1991. SIAM.

[5] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam. A users' guide to PVM parallel virtual machine. Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, July 1991.

[6] Adam Beguelin and Gary Nutt. Collected papers on Phred. Technical Report CU-CS-511-91, University of Colroado, Department of Computer Science, Boulder, CO 80309-0430, January 1991.

[7] Adam Beguelin and Gary Nutt. Examples in Phred. In Jack Dongarra, Ken Kennedy, Paul Messina, Danny C. Sorensen, and Robert G. Voigt, editors, *Proceedings of Fifth SIAM Conference on Parallel Processing for Scientific Computing*, pages 602–608, Philadelphia, 1991. SIAM.

[8] Adam Beguelin and Gary Nutt. Visual parallel programming and determinacy: A language specification, an analysis technique, and a programming tool. Technical Report CMU-CS-93-166, Carnegie Mellon University, June 1993. To appear in JPDC in 1994.

[9] Adam L. Beguelin. SCHEDULE: A hypercube implementation. In *Proceedings of The Third Conference on Hypercube Concurrent Computers and Applications*, volume 1, pages 468–471, January 1988.

[10] Kenneth Birnam and Keith Marzullo. Isis and the META project. *Sun Technology*, pages 90–104, Summer 1989.

[11] James C. Brown. Formulation and programming of parallel computers: a unified approach. In *Proc. Intl. Conf. Par. Proc.*, pages 624–631, 1985.

[12] Jim Browne, Muhammad Azam, and Stephen Sobek. CODE: A unified approach to parallel programming. *IEEE Software*, 6(4):10–18, July 1989.

[13] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[14] J. J. Dongarra and D. C. Sorensen. A portable environment for developing parallel FORTRAN programs. In *Proceedings of the International Conference on Vector and Parallel Computing – Issues in Applied Research and Development*, pages 175–186, July 1987. Published in Parallel Computing, Volume 5, Numbers 1 & 2.

[15] J. J. Dongarra and D. C. Sorensen. SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs. In D. B. Gannon L. H. Jamieson and R. J. Douglass, editors, *The Characteristics of Parallel Algorithms*, pages 363–394. The MIT Press, Cambridge, Massachusetts, 1987.

[16] J. Flower, A. Kolawa, and S. Bharadwaj. The express way to distributed processing. *Supercomputing Review*, pages 54–55, May 1991.

[17] David Gelernter and David Kaminsky. Supercomputing out of recycled garbage: Preliminary experience with Piranha. In *1992 International Conference on Supercomputing*, pages 417–427. ACM, ACM Press, July 1992.

[18] Juan Leon, Allan L. Fisher, and Peter Steenkiste. Fail-safe pvm: A portable package for distributed programming with transparent recovery. Technical Report CMU-CS-93-124, Carnegie Mellon University, February 1993.

[19] J. M. Levesque. FORGE 90 and High Performance Fortran (HPF). In J. S. Kowalik and L. Grandinetti, editors, *Software for Parallel Computation*, pages 111–119. Springer-Verlag, Berlin Germany, 1993.

[20] M. Litzkow, M. Livny, and M. Mutka. Condor — A hunder of idle workstations. In *Proceedings of the Eighth Conference on Distributed Computing Sys tems*, San Jose, California, June 1988.

[21] Peter Newton and James C. Browne. The code 2.0 graphical parallel programming language. In *1992 International Conference on Supercomputing*, pages 167–177. ACM, ACM Press, July 1992.

[22] Charles L. Seitz, Jakov Seizovic, and Wen-King Su. The C programmer's abbreviated guide to multicomputer programming. Technical Report Caltech-CS-TR-88-1, California Institute of Technology, Department of Computer Science, Pasadena, California 91125, 1988.

[23] V. S. Sunderam. PVM : A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.