editor, *Proceedings of Fifth SIAM Conference on Parallel Processing*, Philadelphia, 1991. SIAM.

[3] 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.

[4] A. Beguelin and G. Nutt. Collected papers on Phred. Technical Report CU-CS-511-91, University of Colorado, Department of Computer Science, Boulder, CO 80309-0430, January 1991.

[5] A. Beguelin and G. Nutt. Examples in Phred. In D. Sorensen, editor, *Proceedings of Fifth SIAM Conference on Parallel Processing*, Philadelphia, 1991. SIAM.

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

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

[8] 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.

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

[10] G. A. Geist and V. S. Sunderam. Experiences with network based concurrent computing on the pvm system. Technical Report ORNL/TM-11760, Oak Ridge National Laboratory, January 1991.

[11] Charles L. Seitz. Multicomputers: Message-Passing Concurrent Computers. *Computer*, pages 9–24, August 1988.

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

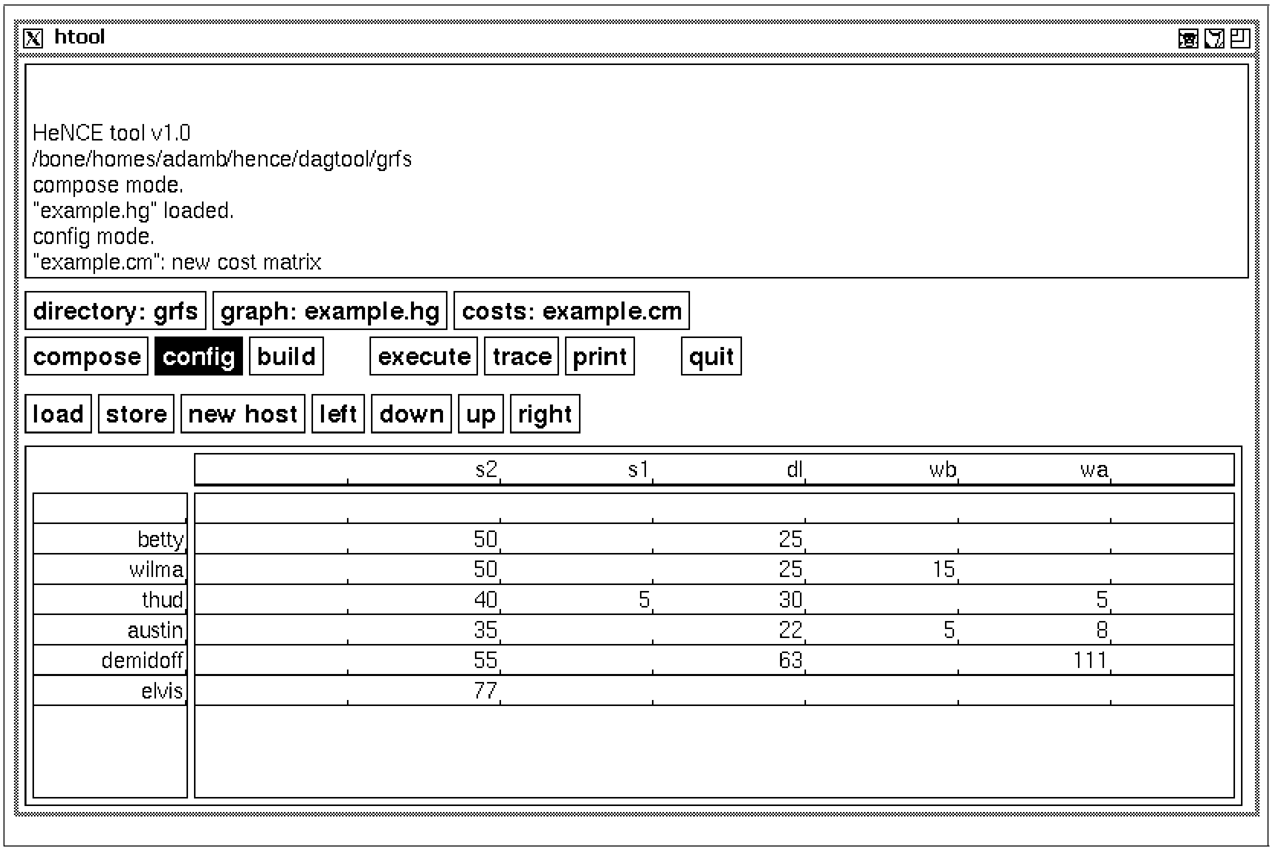

Figure 5: HeNCE tool being used to configure a cost matrix.

could be extended to support hierarchy in the graphs. This would allow the programmer to create larger programs. The trace animation tool could also use these techniques when animating a program run. More debugging and profiling tools need to be added. Allowing breakpoints to be placed on the graph and parameter contents examined or altered at runtime would be useful. Multiple trace files could be displayed in a comparative manner, showing the relative times for executing a program on different virtual machines. It would also be useful to have the HeNCE tool coordinate the execution of source level debuggers over the configured machines. HeNCE could be extended so that during program execution, it takes into account the load and speed of the machines and network when mapping subroutines to machines. This information could be experimentally determined by the HeNCE tool.

6 Availability

PVM is available by sending electronic mail to netlib@ornl.gov containing the line "send index from pvm". Instructions on how to receive the various parts of the PVM system will be sent by return mail. As the HeNCE system becomes available it too will be available from netlib.

References

[1] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam. Heterogeneous network supercomputing. *Supercomputing Review*, August 1991.

[2] 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 D. Sorensen,

tempt to validate them. The costs are used by HeNCE when mapping subroutines to machines. The scheduling algorithm is simple. HeNCE keeps track of the load it has mapped to each machine and chooses the machine $m \ni \min_m(\mathrm{load}_m + \mathrm{cost}_{m\times s})$. This strategy only takes into account HeNCE originated load.

Executing a HeNCE program Once a graph has been specified, executables generated, and a cost matrix defined, then HeNCE can execute the computation. At this point HeNCE configures a virtual machine based on the machines named in the cost matrix. The virtual machine is configured using PVM. After the virtual machine is configured, HeNCE starts executing the program graph. This execution is orchestrated by a central master process. The master process spawns subroutines on the virtual machine. These processes execute, communicating with their ancestors to obtain the necessary parameters. When they finish execution, they check in with the master and send off any outgoing parameters before exiting. Nodes generate trace events during different phases of their execution.

HeNCE achieves fault tolerance through checkpointing. When a node is spawned, its input parameters can be saved to disk. If the computation halts because of an external failure, such as a power outage, it can be restarted from the most recently saved state.

Tracing and Analyzing with HeNCE HeNCE provides a trace mode for visualizing the performance of HeNCE programs. Trace information emitted during execution may be either displayed as the program is running or stored to be displayed later. (See Figure 1 for an example of the trace mode in action.) Trace information is displayed as an animation sequence. One window displays icons for each machine in the virtual machine. These host-icons change color indicating the execution status of the node. The host-icons are also annotated with the node numbers to show how the subroutine nodes were mapped. The lower window shows the program graph. The nodes of the program graph also change colors indicating various trace events. A bar graph can be displayed showing processor activity over time. User defined trace events can also be displayed but only as textual strings. The HeNCE tool trace mode can be very useful for analyzing the performance of a distributed program. During animation, bottlenecks and load imbalances become obvious. The HeNCE programmer can fine tune a program by adjusting the cost matrix or restructuring the parallelism.

HeNCE also performs static analysis of graphs. This analysis is carried out by a built-in critic function. Syntactic information, such as cycles, are detected and the involved nodes highlighted. The critic also checks the graph against the cost matrix. If it is impossible to execute the graph to completion with the current cost matrix (i.e. an incomplete set of costs), the user is notified. Common mistakes, such as missing or mismatched parameters, are also detected.

All of the tools mentioned here have been integrated into a single programming environment. All of the facilities are accessible through a single graphical interface built under X-windows.

4 Summary

The focus of this work is to provide a paradigm and graphical support tools for programming a heterogeneous network of computers as a single resource. HeNCE is the graphical based parallel programming paradigm. In HeNCE the programmer explicitly specifies parallelism of a computation by drawing graphs. The nodes in a graph represent user defined subroutines and the edges indicate parallelism and control flow. The HeNCE programming environment consists of a set of graphical tools which aid in the creation, compilation, execution, and analysis of HeNCE programs. The main components consist of a graph editor for writing HeNCE programs, a build tool for creating executables, a configure tool for specifying which machines to use, an executioner for invoking executables, and a trace tool for analyzing and debugging a program run. These tools are integrated into a window based programming environment.

HeNCE is an active research project. A prototype of the HeNCE environment has been built and is being used.

5 Future Work

Both the paradigm and the tools are being addressed in the ongoing work on HeNCE. The HeNCE graphs are restrictive. It may be possible to develop less restrictive graphs. The current graph constructs need to be evaluated as to their usefulness. It may be that some constructs are not needed and that new ones need to be developed. This can be addressed through implementing examples in the HeNCE paradigm. There are also interesting areas to explore with respect to the HeNCE tools. The editor

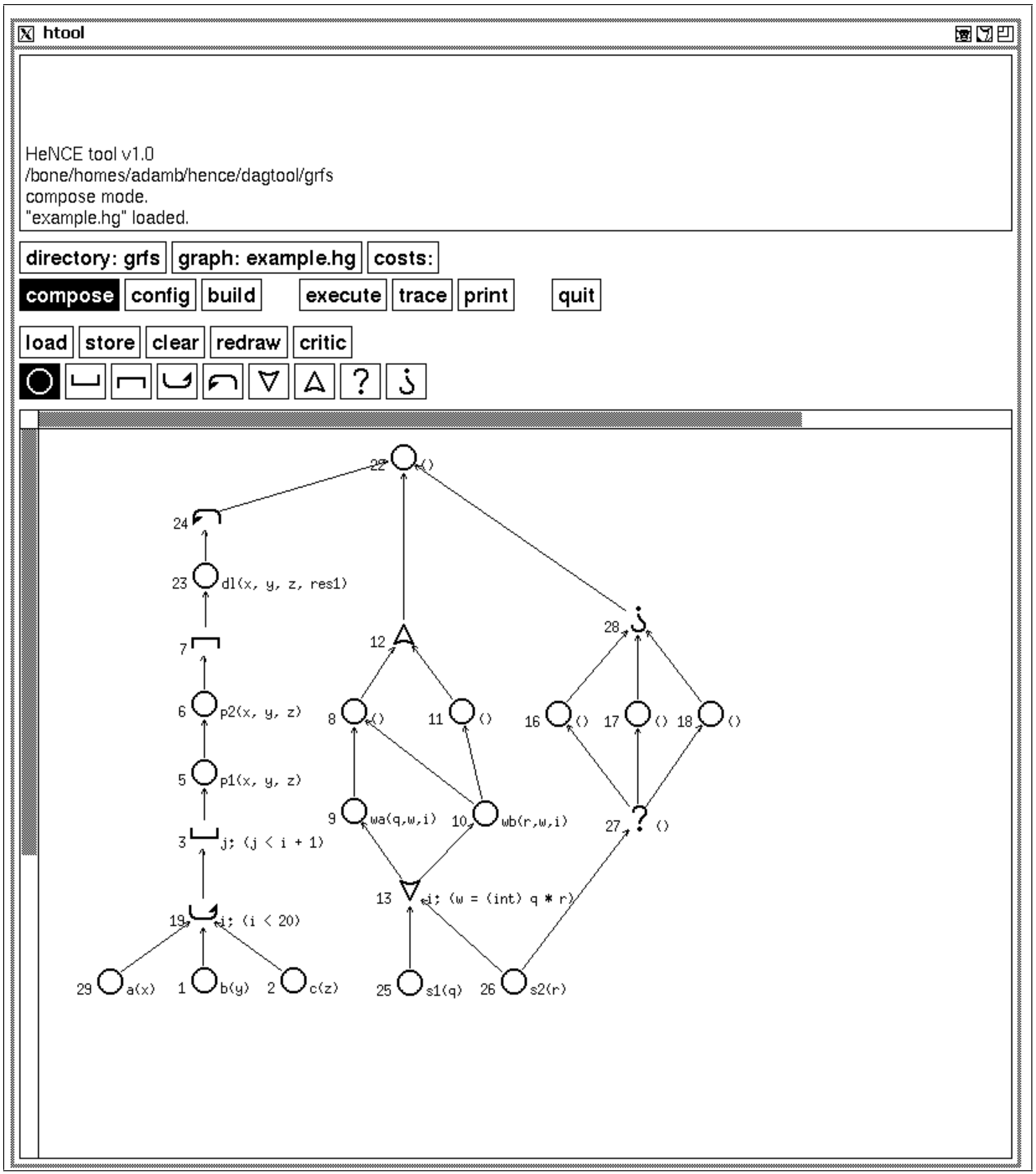

Figure 4: HeNCE tool compose mode.

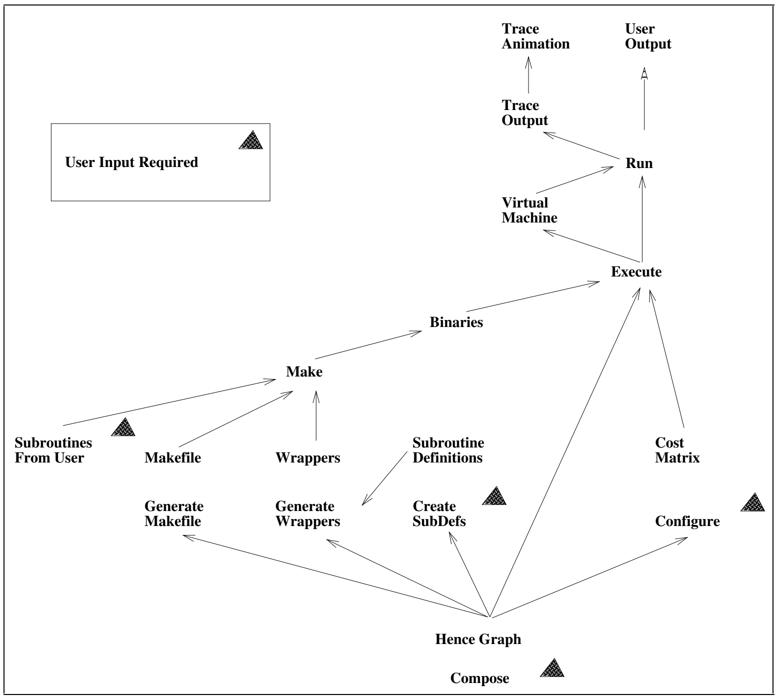

Figure 3: Functional components of HeNCE.

bugging. Wrappers are compiled along with the user supplied subroutines and linked into the HeNCE library. One executable per subroutine per architecture is produced. A strength of HeNCE is its ability to tie together drastically different architectures into a virtual parallel supercomputer. In fact, the machines which make up this virtual parallel supercomputer may themselves be parallel supercomputers. A subroutine in a HeNCE graph may be implemented by several different algorithms, depending on a subroutine's target architecture. For instance, computing the LU decomposition of a matrix on an Intel iPSC/860 is much different from computing the same procedure a Cray YMP. HeNCE supports the specification of different subroutine implementations. HeNCE invokes the appropriate implementation when a subroutine is executed on a particular machine. The wrapper code is architecture independent. It is the user's code for the subroutine that differs. The wrapper code (and thus the executable) is also independent of the graph. In other words, although a subroutine may occur at several places in the graph, the wrapper code is generic enough to be invoked as any node in the graph. This greatly simplifies the generation of executable code for a graph since fewer binary files need to be manipulated. HeNCE will also generate a generic makefile that can be used to make the necessary executables for the various host machines. This makefile may have to be customized if the program uses libraries or source code which is not directly related to HeNCE. (For instance if one of the subroutine nodes in the HeNCE graph opens a window to display some results, then it may need to be linked into a windowing library.)

Configuring a PVM So far we have discussed how the program is specified in HeNCE. The HeNCE user must also indicate which machines will be used to execute the distributed program. HeNCE addresses this problem by requiring that the user specifies a cost matrix. The cost matrix is a $|machines| \times |subroutines|$ matrix where entry (m, s) is an integer that specifies an estimated cost for running subroutine s on machine m. The HeNCE tool provides a spreadsheet like interface for inputing these values. (Figure 5 shows the HeNCE tool in configure mode.) Subroutine names are automatically taken from the graph. The user inputs the machine names (a string) and the costs (a positive integer). The higher the integer (m, s), the higher the cost of running subroutine s on machine m. If a cost does not exist for an entry (m, s) then subroutine s will not be executed on machine m. These costs are entirely user defined and HeNCE makes no at-

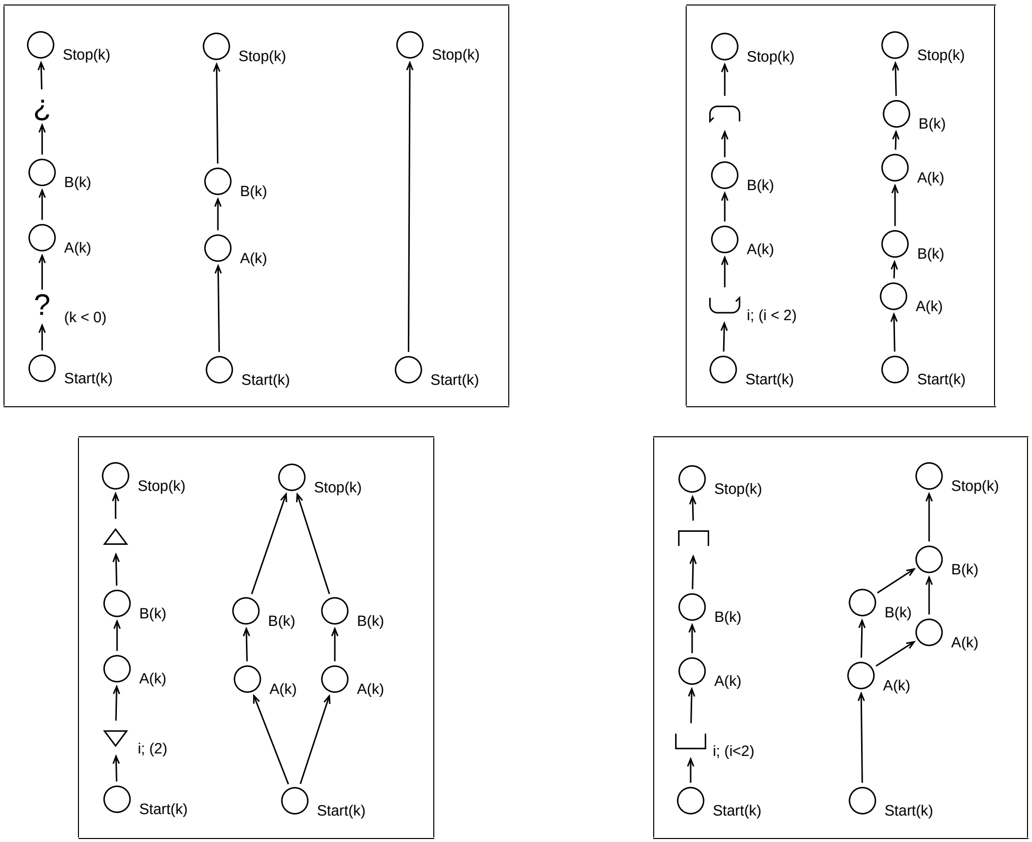

Figure 2: HeNCE loop, fan, pipe, and conditional graph constructs.

user to interactively create graphs by drawing nodes and connecting them with arcs. (Figure 4 shows the HeNCE tool in compose mode.) For each node in the graph the user must specify a procedure which will be called when that node's dependencies have been satisfied. The user attaches procedure names and parameter lists to each subroutine node. The subroutine name tells HeNCE which subroutine to execute when the dependencies for that node have been satisfied. The parameter list indicates which parameters are needed to execute the attached subroutine. For each subroutine there is also a subroutine declaration which indicates the types and input/output characteristics of each parameter. Note that while a subroutine may be attached to more than one node in the graph, it only needs to be declared once.

The names of the parameters attached to a subroutine node are important. The names indicate which data items are required to execute a subroutine. For instance if a subroutine S, attached to node N, needs an integer parameter A, then hence will need to find A among the predecessors of N in order to invoke S with the correct value. This matching is done on the parameter name by searching back through the graph. The user does not need to explicitly write any code to handle parameter passing. Once a subroutine and its parameters are described to HeNCE through a subroutine declaration, HeNCE automatically distributes the parameters at runtime.

Building HeNCE executables In order to produce executables for a HeNCE program the user must provide the annotated HeNCE graph as previously described, plus the source code for the subroutines to be executed. The HeNCE build mode automatically generates wrapper code for each subroutine in the graph. This wrapper code is tailored for each subroutine. It obtains the subroutine's parameters via HeNCE library routines then invokes the subroutine. (The HeNCE library in turn uses PVM for process initialization and communication.) The wrapper also generates trace events which can be used for de-

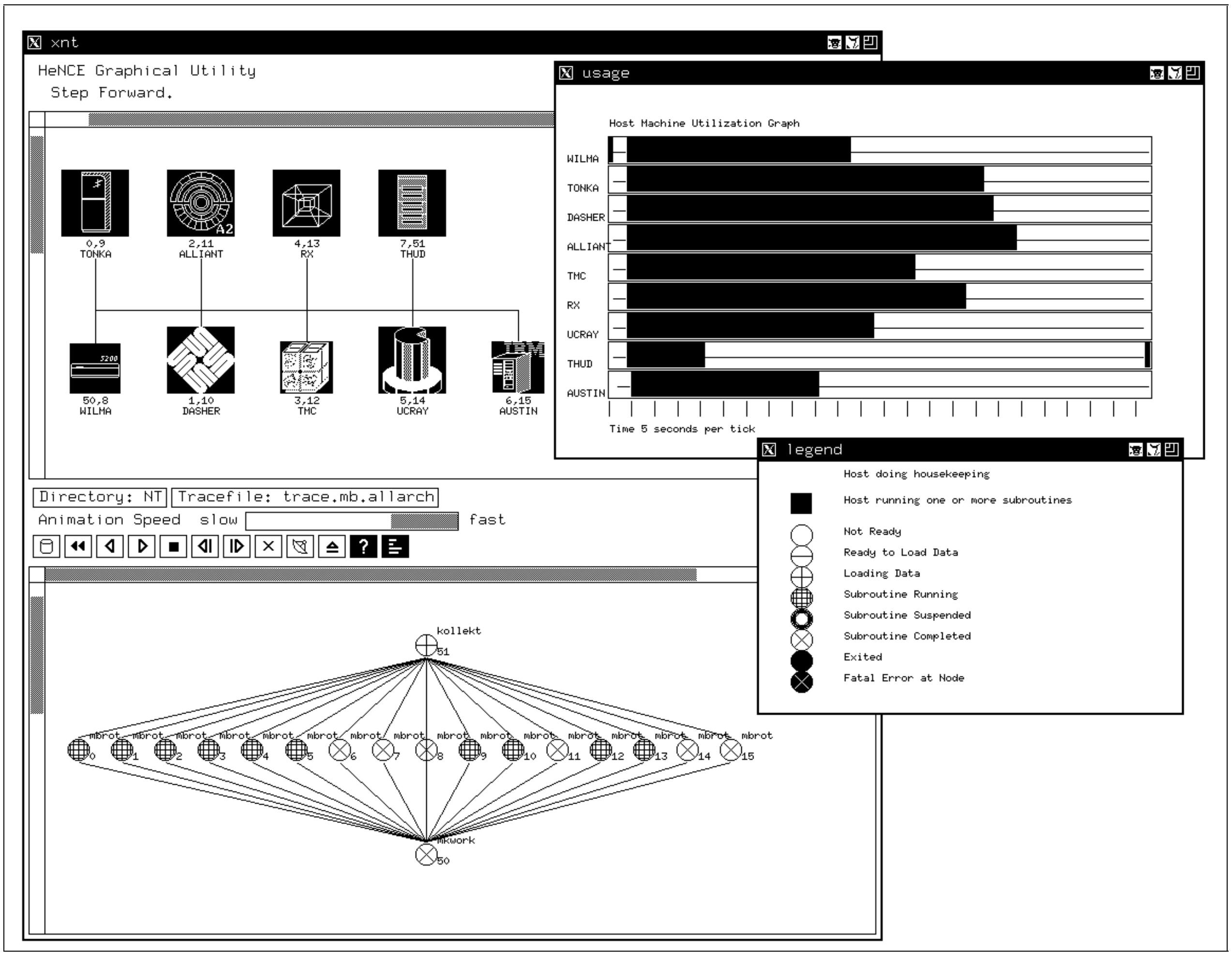

Figure 1: Trace mode of the HeNCE graphical interface.

the programmer attaches a subroutine to a node in the graph. By automatically passing parameters, HeNCE programs are easier to build from pieces of code that have already been written. Thus, re-usability is enhanced. Based on the user input graph, HeNCE automatically distributes the parameters to the subroutines at runtime using PVM for data transmission and conversion.

Now that we have sketched out the basic paradigm used to specify parallelism in HeNCE, we can describe, in some detail, the graphical tools we provide to support such a paradigm.

3 The HeNCE Tool

A graphical user interface is provided for writing HeNCE programs. This tool provides an interface for creating HeNCE graphs, configuring virtual machines, visualizing trace information, compiling, executing, and analyzing HeNCE programs. Figure 1 shows the trace mode of current graphical interface prototype. The tool makes the use of HeNCE and PVM facilities easier, however, for portability reasons it is still possible to utilize HeNCE and PVM without the graphical interface. All essential facilities have textual counterparts. The major functions and of HeNCE and their dependencies are sketched in Figure 3.

Editing a HeNCE graph The editor component, or compose mode, of the HeNCE tool allows the 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. Express [9], like Linda, is also a commercial product. Express provides libraries and tools for writing programs for distributed memory multicomputers. PVM [10] is used as the infrastructure for HeNCE because it provides the right combination of attributes important to HeNCE: support for heterogeneity, ease of installation, minimal system resource requirements, portability, lack of commercial restrictions, and a straightforward programming interface.

2 The HeNCE Paradigm

In HeNCE, the programmer is responsible for explicitly specifying parallelism by drawing graphs which express the dependencies and control flow of a program. HeNCE provides a class of graphs as a usable yet flexible way for the programmer to specify parallelism. The user directly inputs the graph using a graph editor which is part of the HeNCE environment. Each node in a HeNCE graph represents a subroutine written in either Fortran or C. Arcs in the HeNCE graph represent dependencies and control flow. An arc from one node to another represents the fact that the tail node of the arc must run before the node at the head of the arc. During the execution of a HeNCE graph, procedures are automatically executed when their predecessors, as defined by dependency arcs, have completed. Functions are mapped to machines based on a user defined cost matrix.

There are six types of constructs in HeNCE graphs; subroutine nodes, simple dependency arcs, conditional, loop, fan, and pipe constructs. Subroutine nodes represent a particular subroutine and parameter list that will be invoked during the execution of the program graph. A subroutine node has no state other than its parameter list. That is, it cannot read any global information from other subroutine nodes, nor can can it write any global variables (outside its parameter list) that will be read by other subroutine nodes. Dependency arcs represent dependencies between subroutine nodes in a HeNCE graph. The bottom window of the trace tool in Figure 1 shows a simple HeNCE graph containing only subroutine nodes and dependency arcs. This graph represents a simple fractal computation. (In the convention for drawing HeNCE graphs, they are shown to execute from bottom to top.) The initialize node, at the bottom of the graph, reads input parameters describing which part of the complex plane to use for the computation. After the initializing node (mkwork) completes, the dependency arcs from it to the compute nodes are satisfied and they may begin execution. In this case they are invoked on different parts of the complex plane. Once all of the compute nodes are finished the display procedure (kollekt) can execute and display the resulting fractal.

In addition to simple dependency arcs, HeNCE provides constructs which denote four different types of control flow: conditionals, loops, fans, and pipes. These four constructs can be thought of as graph rewriting primitives. Figure 2 illustrates these four constructs, showing each construct and how it may modify the graph. These constructs add subgraphs to the current program graph based upon expressions which are evaluated at runtime. Using the conditional construct the programmer may specify a subgraph to be executed conditionally. If the boolean expression attached to the begin-conditional node evaluates to true then the subgraph contained between the begin- and end-conditional nodes is added to the program graph. If the expression evaluates to false then the contained subgraph is not added. The loop construct is similar to the conditional in that it specifies a subgraph to be conditionally executed. However, the loop construct also allows iteration on a subgraph as a loop body. In other words, the subgraph making up the loop body is repeatedly added to the program graph based upon a boolean expression that is evaluated each time through the loop. The fan construct in HeNCE allows the programmer to specify a parallel fanning out and in of control flow to a dynamically created number of subgraphs. The integer expression attached to the begin-fan node is evaluated to determine how many subgraphs will be created. Each subgraph created by the fan construct executes in parallel. The pipe construct in HeNCE provides for pipelined execution of a subgraph. The expression attached to the begin-pipe node indicates whether another data item is to be piped though the subgraph. If the expression evaluates to true then another subgraph is added to the graph in order to execute the additional data item in a pipelined fashion. Thinking of these constructs as graph rewriting primitives not only provides a mechanism for specifying parallelism but also a natural way of viewing the dynamic parallelism as a graph unfolds at runtime.

The parameter passing interface is one of the strengths of HeNCE. HeNCE programmers need only specify which parameters are to be used to invoke each subroutine node. These parameters are specified when memory multicomputer then another algorithm may be used that has been tuned for that architecture. At runtime HeNCE dynamically chooses which machine executes a node. Based on programmer input, the correct procedure is used for a target 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 a user defined collection of machines (called a parallel virtual machine). Using HeNCE, the programmer specifies the various machines on which the program is to run. This collection of machines can be widely varying. A node in the parallel virtual machine could be anything from a single processor workstation to a 64K processor CM-2. The network connecting these nodes is assumed to have high latency and low bandwidth, thus 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 execution. Various statistics recorded during program execution may also be stored and replayed post mortem. Some debugging support is available, mainly consisting of user defined messages displayed during a trace animation.

HeNCE is implemented on top of a system called PVM (Parallel Virtual Machine) [1, 12]. PVM is a software package that allows the utilization of a heterogeneous network of parallel and serial computers as a single computational resource [2]. PVM provides facilities for spawning, communication, and synchronization of processes over a network of heterogeneous machines [3]. While PVM provides the low level tools for implementing parallel programs, HeNCE provides the programmer with a higher level abstraction for specifying parallelism.

In this paper we will expand upon the points introduced here. First is an explanation of the HeNCE paradigm and how procedures and graphs are interfaced, followed by a discussion of the HeNCE graphical interface. Finally, a summary is given of the project's current status with instructions for obtaining available software.

Related Work Several research projects have goals which overlap with those of HeNCE. Schedule, Phred, and Paralex are a few examples.

Schedule [8] is a graph based parallel programming tool 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 advantages of the available shared memory. If fact, a HeNCE node executing on a shared memory machine could actually utilize the Schedule primitives.

Phred [4, 5] is also similar to HeNCE. Phred graphs are more complicated than those of HeNCE; they contain separate control and data flow subgraphs. The graph constructs of HeNCE are based on similar constructs in Phred. However, the emphasis of Phred is not heterogeneity but determinacy.

Paralex 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. The HeNCE programmer must specify the types and sizes of the elements in a procedure's parameter list. 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 that 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. HeNCE graphs are richer than those of Paralex. HeNCE provides dynamically spawned subgraphs, pipelining, loops and conditionals. Pipelining is provided in Paralex but only for the entire program graph.

There are several platforms on which HeNCE could have been built. Isis [6] is a parallel programming toolkit for fault tolerant parallel computing over a network of heterogeneous machines. Isis is a large system. It requires significant system resources and a system administrator to properly install. The Cosmic environment [11] is a publicly available parallel programming environment targeted toward tightly coupled homogeneous groups of local memory MIMD machines or multicomputers. Network Linda is a commercial implementation of the Linda primitives [7] which runs

Graphical Development Tools for Network-Based Concurrent Supercomputing *

Adam Beguelin and Jack J. Dongarra
Oak Ridge National Laboratory and University of Tennessee

G.A. Geist
Oak Ridge National Laboratory

Robert Manchek
University of Tennessee

V. S. Sunderam
Emory University

Abstract

This paper describes an X-window based software environment called HeNCE (Heterogeneous Network Computing Environment) designed to assist scientists in developing parallel programs that run on a network of computers. HeNCE is built on top of a software package called PVM which supports process management and communication between a network of heterogeneous computers. HeNCE is based on a parallel programming paradigm where an application program can be described by a graph. Nodes of the graph represent subroutines and the arcs represent data dependencies. HeNCE is composed of integrated graphical tools for creating, compiling, executing, and analyzing HeNCE programs.

1 Introduction

Wide area computer networks have become a basic part of today's computing infrastructure. These networks connect a variety of machines, presenting an enormous computing resource. In this project we focus on developing methods and tools which allow a programmer to tap into this resource. In this paper we describe HeNCE, a tool and methodology under development that assists a programmer in developing programs to execute on a networked group of heterogeneous machines.

The HeNCE philosophy of parallel programming is to have the programmer *explicitly* specify the parallelism of a computation and to aid the programmer as much as possible in the tasks of designing, compiling, executing, debugging, and analyzing the parallel computation. Central to HeNCE is a graphical interface which the programmer uses to perform these tasks.

In HeNCE the programmer specifies graphs, a variant of directed acyclic graphs or DAGs, where the nodes in the graphs represent procedures and the edges denote dependencies. These program graphs are input by the programmer using a feature of the graphical interface. (There is also a textual interface where a programmer may create a program graph using a conventional 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 HeNCE tool provides facilities for editing and compiling these procedures on the various architectures of the user defined virtual machine.

An essential 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

*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.