Tools for Heterogeneous Network Computing [*]

Adam Beguelin[†], Jack Dongarra[‡], Al Geist[§], Robert Manchek[¶], Keith Moore[¶], and Vaidy Sunderam[‖]

December 13, 1996

Abstract

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 PVM and HeNCE, tools and methodology under development that assists a programmer in developing programs to execute on a networked group of heterogeneous machines.

HeNCE is implemented on top of a system called PVM (Parallel Virtual Machine). 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. While PVM provides the low level tools for implementing parallel programs, HeNCE provides the programmer with a higher level abstraction for specifying parallelism.

1 Introduction

Heterogeneous networks of computers are becoming commonplace in high-performance computing. Systems ranging from workstations to supercomputers are linked together by high speed networks. Until recently each computing resource on the network remained a separate unit, but now over 100 institutions worldwide are writing and running truly *heterogeneous* programs utilizing multiple computer systems to solve applications through the use of a software package called PVM.

PVM stands for Parallel Virtual Machine [13, 11, 3]. PVM is designed from the ground up with heterogeneity and portability as primary goals. As such it is one of the first software systems that allows machines with wildly different architectures and floating point representations to work together on a single computational task.

The Heterogeneous Network Project, being worked on by researchers at Oak Ridge National Laboratory, the University of Tennessee, and Emory University, is involved in the research and development of two software packages specifically designed to facilitate heterogeneous parallel computing. The first package is PVM, which can be used on

[*]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 in part by the the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615.

[†]Carnegie-Mellon University

[‡]University of Tennessee and Oak Ridge National Laboratory

[§]Oak Ridge National Laboratory

[¶]University of Tennessee

[‖]Emory University

its own or as a foundation upon which other heterogeneous network software can be built. The second package is called HeNCE, which stands for Heterogeneous Network Computing Environment [2, 1]. HeNCE is being built on top of PVM with the intention of simplifying the task of writing, compiling, running, debugging, and analyzing programs on a heterogeneous network. The goal is to make network computing accessible to scientists and engineers without the need for extensive training in parallel computing and allowing them to use resources best suited for a particular phase of the computation.

What follows is a description of the basic features of these two packages.

2 PVM

PVM is a software package that permits a heterogeneous collection of serial, parallel and vector computers hooked together by a network to appear as one large computer. Thus, PVM allows a user to exploit the aggregate power of workstations and supercomputers distributed around the world to solve computational grand challenges.

The user views PVM as a loosely coupled distributed-memory computer programmed in C or Fortran with message-passing extensions. The hardware that composes the user's personal PVM may be any UNIX based machine on which the user has a valid login and is accessible over some network.

PVM may be configured to contain various machine architectures including sequential processors, vector processors, multicomputers, etc. The present version of the software has been tested with various combinations of the following machines: Sun3, SPARCstation, Microvax, DECstation, IBM RS/6000, HP-9000, Silicon Graphics IRIS, NeXT, Sequent Symmetry, Alliant FX, IBM 3090, Intel iPSC/860, Thinking Machines CM-2 and CM-5, KSR-1, Convex, Cray Y-MP, and Fujitsu VP-2000. In addition, users can port PVM to new architectures by simply modifying a generic 'makefile' supplied with the source and recompiling.

Using PVM, each user can configure his own parallel virtual computer, which can overlap with other users' virtual computers. Configuring a personal parallel virtual computer involves simply listing the names of the machines in a file that is read when PVM is started. Several different physical networks can co-exist inside a virtual machine. For example, a local ethernet, HIPPI, and a fiber optic network can all be a part of a user's virtual machine. While each user can have only one virtual machine active at a time, PVM is multitasking so several applications can run simultaneously on a parallel virtual machine.

The PVM package is small (less than 400 Kbytes of C source code) and easy to install. It needs to be installed only once on each machine to be accessible to all users. Moreover, the installation does not require special privileges on any of the machines and thus can be done by any user.

Application programs that use PVM are composed of subtasks at a moderately large level of granularity. The subtasks can be generic serial codes, or they can be specific to a particular machine. In PVM, resources may be accessed at three different levels: the *transparent* mode in which subtasks are automatically located at the most appropriate sites, the *architecture-dependent* mode in which the user may indicate specific architectures on which particular subtasks are to execute, and the *machine-specific* mode in which a particular machine may be specified. Such flexibility allows different subtasks of a heterogeneous application to exploit particular strengths of individual machines on the network.

The PVM user-interface requires that all message data be explicitly typed. PVM performs machine-independent data conversions when required, thus allowing machines with different integer and floating point representations to pass data. Applications access PVM resources via a library of standard interface routines. These routines allow the initiation and termination of processes across the network as well as communication and synchronization between processes. Communication constructs include those for the exchange of data structures as well as high-level primitives such as broadcast, barrier synchronization, and rendezvous.

Application programs under PVM may possess arbitrary control and dependency structures. In other words, at any point in the execution of a concurrent application, the processes in existence may have arbitrary relationships between each other and, further, any process may communicate and/or synchronize with any other.

3 HeNCE

While PVM provides low-level tools for implementing parallel programs, HeNCE provides the programmer with a higher level environment for using heterogeneous networks. The HeNCE philosophy of parallel programming is to have the programmer explicitly specify the parallelism of a computation and to automate, as much as possible, the tasks of writing, compiling, executing, debugging, and analyzing the parallel computation. Central to HeNCE is an X-Window interface that the programmer uses to perform these functions. (see Figure 2).

The HeNCE environment contains a **compose** tool that allows the user to explicitly specify parallelism by drawing a graph of the parallel application. If an X-window interface is not available, then textual graph descriptions can be input.

Each node in a HeNCE graph represents a procedure written in either Fortran or C. HeNCE is designed to enhance procedure reuse. The procedure can be a subroutine from an established library or a special purpose subroutine supplied by the user. Arcs between nodes represent data dependency and control flow. A dependency arc from one node to another represents the fact that the tail node of the arc must run before the head of the arc. Data is sent to a node from its ancestors in the graph (usually its parents).

In addition to simple nodes, four types of control constructs are available in the HeNCE graph language. One represents looping; a second represents conditional dependency; a third represents a fan-out to a variable number of identical subgraphs; and a fourth represents pipelining. The graph can contain loops around subgraphs that execute a variable number of times based on the expression in the loop construct. Using a conditional construct, a section of the graph can be executed or bypassed based on an expression that will be evaluated at run time. A variable fan-out (and subsequent fan-in) construct is available while composing the graph. The width of the fan-out is specified as an expression that is evaluated at run time. This construct is similar to a parallel-do construct found in several parallel Fortrans. In pipelined sections, when a node finishes with one set of input data, it reruns with the next piece of pipelined data.

Once the dynamic graph is specified, a **configuration** tool in the HeNCE environment can be used to specify the configuration of machines that will compose his parallel virtual machine. The configuration tool also assists the user in setting up a cost matrix. The cost matrix allows the user to describe which machine can perform which task and can give priority to certain machines. HeNCE will use this cost matrix at run time to determine the most effective machine on which to execute a particular procedure in the graph.

The HeNCE environment also contains a **build** tool to perform three tasks. First, by analyzing the graph, HeNCE automatically generates the parallel program using PVM calls for all the communication and synchronization required by the application. Second, by knowing the desired PVM configuration, HeNCE automatically compiles the node procedures for the various heterogeneous architectures. Finally, the build tool installs the executable modules on the particular machines in the PVM configuration.

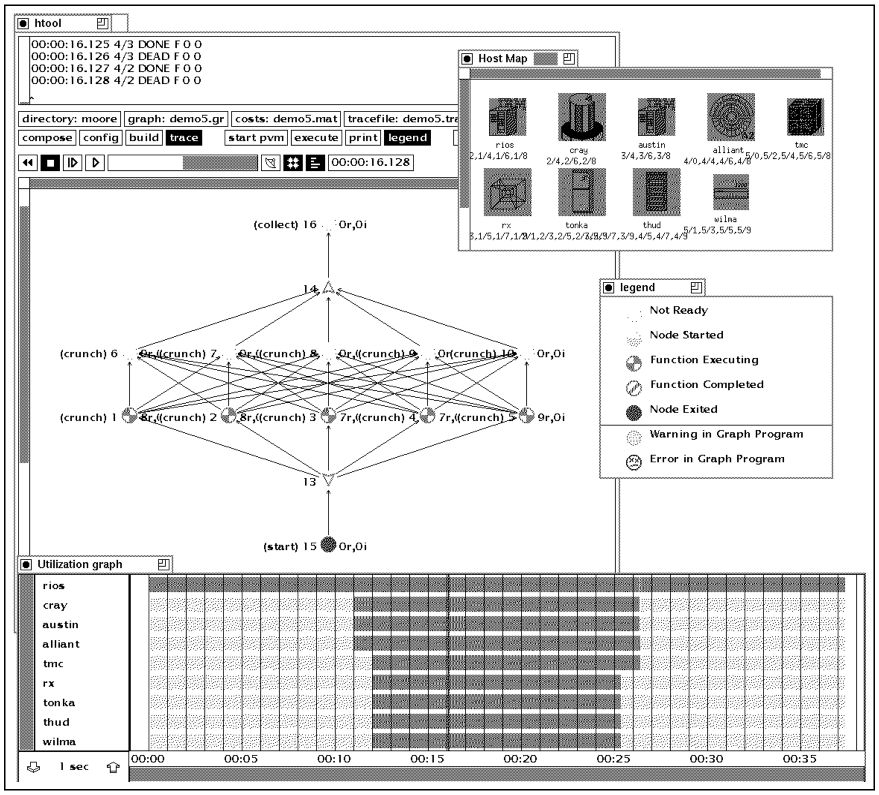
The **execute** tool in the HeNCE environment starts up the requested virtual machine and begins execution of the application. During execution, HeNCE automatically maps procedures to machines in the heterogeneous network based on the cost matrix and the HeNCE graph. Trace and scheduling information that is saved during the execution can be displayed in real time or replayed later.

The HeNCE environment has a **trace** tool that allows visualization of the parallel run. The trace tool is X-window based and consists of two windows. One window shows a representation of the network and machines underlying PVM. In this window icons of the active machines are illuminated with different colors depending on whether they are computing or communicating. Under each icon is a list of the node procedures mapped to this machine at any given instant. The second window displays the user's graph of the application, which changes dynamically to show the actual paths and parameters taken during a run. The nodes in the graph change colors to indicate the various activities going on in each procedure.

4 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.



```
00:00:16.125 4/3 DONE F 0 0
00:00:16.126 4/3 DEAD F 0 0
00:00:16.127 4/2 DONE F 0 0
00:00:16.128 4/2 DEAD F 0 0
```

directory: moore | graph: demo5.gr | costs: demo5.mat | tracefile: demo5.tr

00:00:16.128

FIG. 1. *Trace mode of the HeNCE graphical interface.*

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. 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 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.

5 Summary

The focus of this work is to provide a paradigm and graphical support tool 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 modes 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 steps 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.

6 Future Work

Both the paradigm and the tool 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 tool. The editor 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 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.

7 Availability

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”. Instructions on how to receive the various parts of the PVM and HeNCE systems will be sent by return mail.

8 References

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, 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 Birman and Keith Marzullo. Isis and the META project. *Sun Technology*, pages 90–104, Summer 1989.

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

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

[9] 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.

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

[11] 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.

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

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

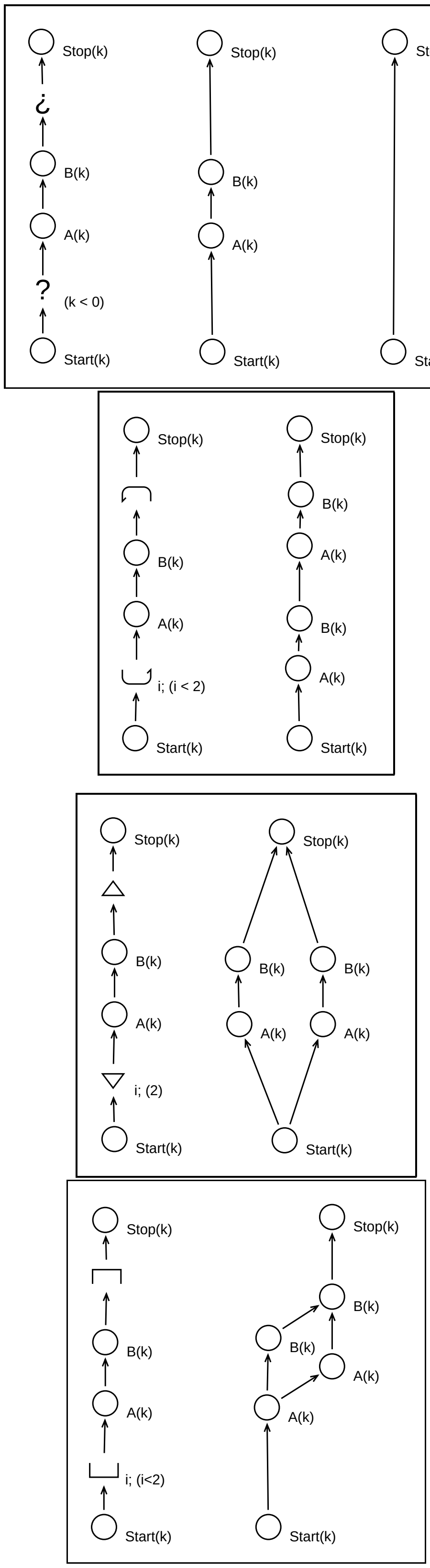


FIG. 2. *HeNCE loop, fan, pipe, and conditional graph constructs.*