INTEGRATED PVM FRAMEWORK SUPPORTS HETEROGENEOUS NETWORK COMPUTING

Jack Dongarra, G. A. Geist, Robert Manchek, and V. S. Sunderam

Parallel Virtual Machine (PVM) software lets scientists exploit collections of networked machines when performing complex computations

Two developments promise to revolutionize scientific problem solving. The first is the development of massively parallel computers. Massively parallel systems offer the enormous computational power needed for solving Grand Challenge problems. Unfortunately, software development has not kept pace with hardware advances. In order to exploit fully the power of these massively parallel machines, new programming paradigms, languages, scheduling and partioning techniques, and algorithms are needed.

The second major development affecting scientific problem solving is distributed computing. Many scientists are discovering that their computational requirements are best served not by a single, monolithic machine, but by a variety of distributed computing resources, linked by high-speed networks.

Heterogeneous network computing offers several advantages. By using existing hardware, the cost of this computing can be kept very low. Performance can be optimized by assigning each individual task to the most appropriate architecture. Network computing also offers the potential for partioning a computing task along lines of service functions. Typically, network computing environments possess a variety of capabilities; the ability to execute subtasks of a computation on the processor most suited to a particular function both enhances performance and improves utilization.

Another advantage in network-based concurrent computing is the ready availability of development and debugging tools, and the potential fault tolerance of the network(s) and processing elements. Typically, systems that operate on loosely coupled networks permit the direct use of editors, compilers, and debuggers that are available on individual machines. These individual machines are quite stable, and substantial expertise in their use is readily available. These factors translate into reduced development and debugging time for the user, reduced contention for resources, and possibly more effective implementations of the application.

Yet another attractive feature of loosely coupled computing environments is the potential for user-level or program-level fault tolerance that can be implemented with little effort in either the application or the underlying operating system. Most multiprocessors do not support such a facility; hardware or software failures in one of the processing elements often lead to a complete crash.

Despite the advantages of heterogeneous network computing, however, many issues remain to be addressed. Of especial importance are issues relating to the user interface, efficiency, compatibility, and administration. In some cases, individual researchers have attempted to address these issues by developing ad hoc approaches to the implementation of concurrent applications. Recognizing the growing need for a more systematic approach, several research groups have recently attempted to develop programming paradigms, languages, scheduling and partitioning techiques, and algorithms.

Our approach is more pragmatic. We discuss the development of an *integrated framework for heterogeneous network computing*, in which a collection of interrelated components provides a coherent high-performance computing environment. In particular, we analyze several of the design features of the Parallel Virtual Machine (PVM) (see Fig. 1).

In this article, we look briefly at the general field of heterogeneous network computing, and discuss some of the research issues that must be addressed before network-based heterogeneous computing can become truly effective. Next, we focus on the PVM system, which is designed to help scientists write programs for such heterogeneous systems. Finally, we discuss a recent extension of PVM

Jack Dongarra is both Distinguished Professor of Computer Science at the University of Tennessee (UT) and Distinguished Scientist in the Mathematical Sciences Section at Oak Ridge National Laboratory. Al Geist is a computer scientist in the Mathematical Sciences Section of Oak Ridge National Laboratory. Robert Manchek is a Research Associate at the University of Tennessee, Knoxville, TN. Vaidy Sunderam is a Professor in the Department of Mathematics and Computer Science at Emory University, GA.

that further assists in the implementation of concurrent applications.

Connecting heterogeneous computers

In the past, researchers have conducted experiments linking workstations that operate at speeds of approximately 1–10 MIPS. Such experiments have included remote execution, computer farms, and migration of computations.

More recently, experiments have focused on linking higher-performance workstations (those performing approximately 10–100 Mflops) together with multiprocessors and conventional supercomputers.

To exploit these multiple computer configurations fully, researchers have developed various software packages that enable scientists to write truly heterogeneous programs. Such software packages include Express, P4, Linda, and PVM. Each of these packages is layered over the native operating systems, exploits distributed concurrent processing, and is flexible and general-purpose; all exhibit comparable performance. The differences lie in their programming model, implementation schemes, and efficiency.

The proceedings of recent conferences, as well as informal discussions, seem to indicate that specialists in high-performance scientific computing are focusing most

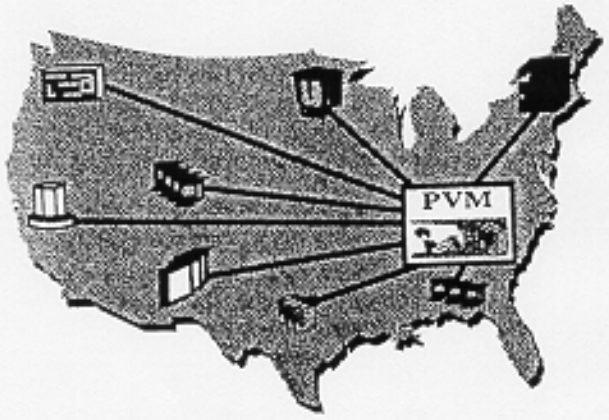

Fig. 1: With PVM software, a heterogeneous collection of networked serial, parallel, and vector computers appears as one large computer. The software supplies C and Fortran routines for asynchronous message passing and process control. It is small and portable, but able to handle large-scale scientific applications.

of their attention on the four software packages mentioned above.

We present brief outlines of each of the first three of these packages before beginning a detailed description of PVM. We wish to emphasize, however, that these systems are by no means the only software packages in existence, and that the descriptions that follow are not detailed and formal critiques, but rather brief synopses abstracted from our understanding of the systems and the developers' own articles or communications.

Linda

Linda[1] is a concurrent programming model that has evolved from a Yale University research project. The primary concept in Linda is that of the "tuple-space," an abstraction by means of which cooperating processes communicate. This central theme of Linda has been proposed as an alternative paradigm to the two traditional methods of parallel processing, namely those based on shared memory and message passing. The tuple-space concept is essentially an abstraction of distributed shared memory, with one important difference (i.e., that tuple-spaces are associative), and several minor distinctions (destructive and nondestructive reads, and different coherency semantics, are possible). Applications use the Linda model by explicitly embedding, within cooperating sequential programs, constructs that manipulate (insert/retrieve tuples) the tuple space.

From the point of view of applications, Linda[2] is a set of programming language extensions for facilitating parallel programming. The Linda model is a scheme built upon an associative memory referred to as tuple-space. It provides a shared memory abstraction for process communication without requiring the underlying hardware to physically share memory. The Linda environment is illustrated in Fig. 2.[1]

Tuples are collections of fields logically "welded" to form persistent storage items. These collections of fields are the basic tuple-space storage units. Parallel processes exchange data by generating, reading, and consuming them. To update a tuple, it is removed from tuple-space, modified, and returned to tuple-space. Restricting tuple-space modification in this manner creates an implicit locking mechanism, ensuring proper synchronization of multiple accesses.

The following are the four basic operations, or primitives, which are added to a language to produce a Linda dialect (see Fig. 2).

(1) *rd*(*t*) performs a nondestructive read from tuple-space. If the desired tuple *t* is not found, the invoking process is suspended until an appropriate tuple is created by another process.

(2) *in*(*t*) behaves in a fashion similar to rd(), except that the read is destructive and the tuple is consumed.

(3) *out*(*t*) writes a tuple *t* to tuple-space.

(4) *eval*(*expression*) writes a tuple to tuple-space after arguments in the *expression* are evaluated by creating new processes that perform their tasks independently.

Tuples are selected by the rd() or in() primitives on the basis of their field values. There are no tuple addresses in an associative memory. Consider the following tuple:

out("a string", 15.01, 17, "another string")

A variety of access routes to this tuple are possible, e.g., any one of the following operations suffices:

rd("a string", ?fval, ?ival, ?strval)

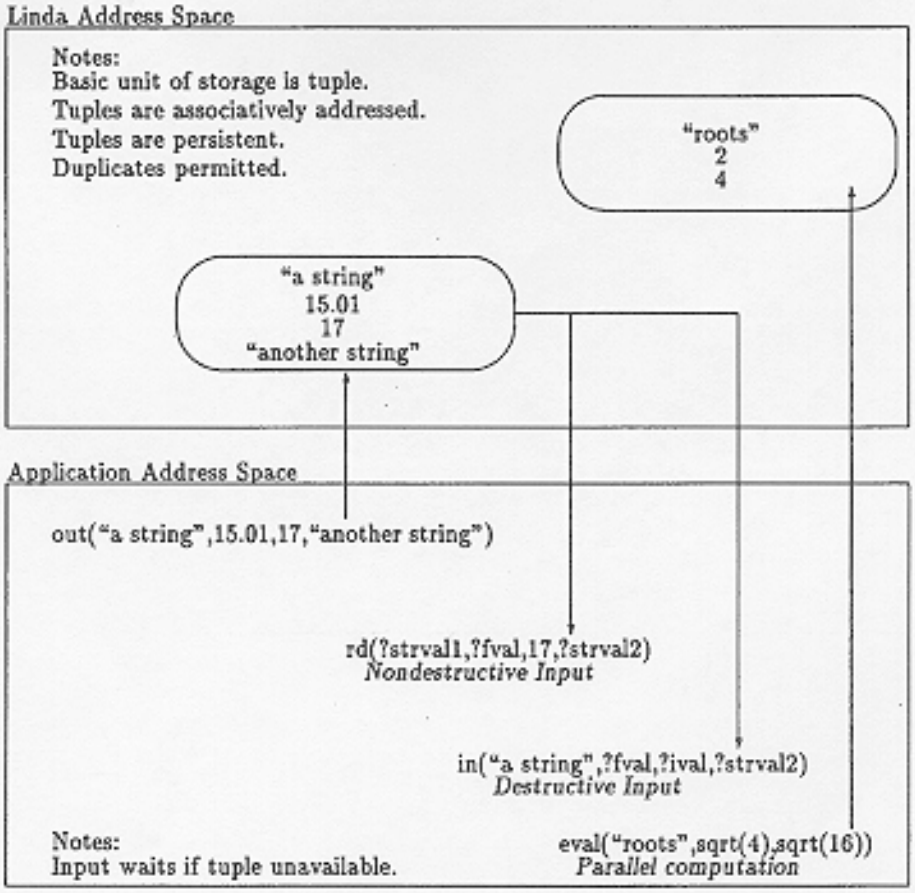

Fig. 2: Cooperating processes communicate in the "tuple-space" of the programming environment Linda.

rd(?strval, 15.01, ?ival, "another string")

rd(?strval-1, ?fval, 17, ?strval-2)

The "?" operator designates a value returned from a matching tuple. Fields marked by the operator do not participate in the (associative-memory) matching process. Any of the three example rd( ) operations results in a nondestructive reading of the original tuple. If the operation were an in( ), the tuple would be removed from tuple-space.

To illustrate the eval( ) primitive, consider the following:

eval("roots",sqrt(4),sqrt(16)).

Using Linda terminology, this creates a *live* tuple. The square-root operations are performed independently of the originating process, with the (two) numeric results combined to form a three-element tuple saved in tuple-space. The eval( ) primitive is a mechanism capable of creating fine-grain parallelism.

The "Linda System" usually refers to a specific (sometimes portable) implementation of software that supports the Linda programming model. System software that establishes and maintains tuple spaces is provided and is used in conjunction with libraries that interpret and execute Linda primitives appropriately. Depending on the environment (shared-memory multiprocessors, message-passing parallel computers, networks of workstations, etc.), the tuple-space mechanism is implemented using different techniques and with varying degrees of efficiency. Recently, a new system technique has been proposed, at least nominally related to the Linda project. This scheme, termed "Piranha," proposes an active approach to concurrent computing—the idea being that computational resources (viewed as active agents) seize computational tasks from a well-known location, based on availability and suitability. Again, this scheme may be implemented on multiple platforms, and manifested as a "Piranha system" or "Linda-Piranha system."

P4 and Parmacs

P4 is a library of macros and subroutines developed at Argonne National Laboratory for programming a variety of parallel machines. The P4 system supports both the shared-memory model (based on monitors) and the distributed-memory model (using message-passing). For the shared-memory model of parallel computation, P4 provides a set of useful monitors, as well as a set of primi-

tives from which monitors can be constructed. For the distributed-memory model, P4 provides typed send and receive operations, and allows for the creation of processes according to a text file describing group and process structure. P4 is intended to be portable, simple to install and use, and efficient. It can be used to program networks of workstations, advanced parallel supercomputers such as the Intel Touchstone Delta and the Alliant Campus HiPPI-based system, and single shared-memory multiprocessors. It has already been installed on most uniprocessor workstations, shared-memory multiprocessors, and several high-performance parallel machines.

Process management in the P4 system is based on a configuration file that specifies the host pool, the object file to be executed on each machine, the number of processes to be started on each host (intended primarily for multiprocessor systems), and other auxiliary information. An example of a configuration file is

```
# start one slave on each of sun2 and sun3
local 0
sun2 1 /home/mylogin/p4pgms/sr_test
sun3 1 /home/mylogin/p4pgms/sr_test
```

Two issues are worth noting with regard to the process-management mechanism in P4. First, there is the notion of a "master" process and of "slave" processes; multilevel hierarchies may be formed to implement a "cluster" model of computation. Second, the primary mode of process creation is static, via the configuration file; dynamic process creation is possible only by means of a statically created process that must invoke a special P4 function, which spawns a new process on the local machine. However, despite these restrictions, a variety of application paradigms may be implemented in the P4 system in a fairly straightforward manner.

Message-passing in the P4 system is achieved through the use of traditional `send` and `recv` primitives, parameterized almost exactly as in other message-passing systems. Several variants are provided for semantics such as heterogeneous exchange and blocking or nonblocking transfer. A significant proportion of the burden of buffer allocation and management, however, is left to the user. Apart from basic message passing, P4 offers a variety of global operations, including broadcast, global maxima and minima, and barrier synchronization.

Shared-memory support via monitors is a facility that distinguishes P4 from other systems. However, this feature is not the same as distributed shared memory, but rather is a portable mechanism for shared-address-space programming in true shared-memory multiprocessors. The abstraction provided by P4 for managing data in shared memory is `monitors`.[3] P4 provides several useful monitors (`p4_barrier_t`, `p4_getsub_monitor_t`,`p4_askfor_monitor_t`), as well as a general monitor type to help users in constructing their own monitors (`p4_monitor_t`).

P4 also supports a variety of auxiliary and support functions, for timing purposes and for debugging. The debugging functions are essentially printing facilities that identify the source of a debugging message. A choice of different "levels" of debugging allows the user to control the volume of debugging information that is printed. Finally, the P4 system also contains a package known as ALOG for creating logs of time-stamped events. This package is of general utility outside of P4. The timestamps are obtained from various timers, with microsecond-level resolution on various machines. These log files are primarily intended for use with a separate tool termed `Upshot`[4] that visually depicts events and the ordering of these events from a P4 application run.

The Parmacs project is closely related to P4. Essentially, Parmacs is a set of macro extensions developed at Gesellschaft für Mathematik und Datenvorarbeitung (GMD).[5] It originated in an effort to provide Fortran interfaces to the P4 system, but it is now a significantly enhanced package that provides a variety of high-level abstractions, mostly dealing with global operations. Parmacs provides macros for logically configuring a set of P4 processes; for example, the macro `torus` produces a suitable configuration file for use by P4 that results in a logical process configuration corresponding to a three-dimensional (3-D) torus. Other logical topologies, including general graphs, may also be implemented, and Parmacs provides macros used in conjunction with `send` and `recv` to achieve topology-specific communications within executing programs.

Express

In contrast to the other parallel-processing systems described in this section, Express[6] toolkit is a collection of tools that individually address various aspects of concurrent computation. The toolkit was developed, and is marketed commercially, by ParaSoft Corporation, a company that was started by some members of the Caltech concurrent-computation project. A second distinction is that Express supports PCs in addition to the usual high-performance computing platforms and workstations.

The philosophy behind computing with Express is based on beginning with a sequential version of an application, and following a recommended development life cycle that culminates in a parallel version tuned for optimality. Typical development cycles begin with the use of VTOOL, a graphical program that allows the progress of sequential algorithms to be displayed in a dynamic manner. Updates and references to individual data structures can be displayed, to demonstrate algorithm structure explicitly and provide the detailed knowledge necessary for parallelization.

Related to the above is FTOOL, which provides in-depth analysis of a program, including variable-use analysis, flow structure, and feedback regarding potential parallelization. FTOOL operates on both sequential and parallel versions of an application. A third tool called ASPAR is then used; this is an automated parallelizer that converts sequential C and Fortran programs for parallel

or distributed execution using the Express programming models.

The core of the Express system is a set of libraries for communication, input/output (I/O), and parallel graphics. The communication primitives are akin to those found in other systems, and include a variety of global operations and data-distribution primitives. Extended I/O routines allow parallel input and output, and a similar set of routines is provided for graphical displays from multiple concurrent processes. Express also contains the NDB tool, a parallel debugger that uses commands based on the popular "dbx" interface. These debugging commands can be issued to single processors or groups of nodes simultaneously.

Finally, Express contains a set of "back-end" tools intended to assist performance monitoring and tuning. CTOOL analyzes high-level overhead issues such as the relative amount of time spent computing, performing I/O, and communicating between processors. ETOOL shows the relationships between various computing elements, and may be used to understand overheads and cause-and-effect relationships between actions in different processors. XTOOL profiles CPU usage on a per-processor basis, and may be used at different levels of granularity.

Ongoing trends

In the next section of this paper, we focus on the basic features of PVM and discuss our experiences with that system. PVM and the other systems described above have evolved over the past several years, but none of them can be considered to be fully mature. The field of network-based concurrent computing is relatively young, and research on various aspects is in progress. Although basic infrastructures have been developed, necessary refinements are still evolving. Some of the ongoing research projects related to heterogeneous network-based computing are outlined briefly here.

Standalone systems delivering several tens of millions of operations per second are commonplace, and continuing increases in power are predicted. For network computing systems, this presents many challenges. One consideration involves scaling to hundreds and perhaps thousands of independent machines; specialists conjecture that functionality and performance equivalent to to those of massively parallel machines can be supported on cluster environments. A project at Fermilab has demonstrated the feasibility of scaling to hundreds of processors for some classes of problems (see article on p. xx in this issue). Research into protocols to support scaling and other system issues is currently under investigation. Furthermore, under the right circumstances, the network-based approach can be effective in coupling several similar multiprocessors, resulting in a configuration that might be economically and technically difficult to achieve with hardware.

Applications with large execution times will benefit greatly from mechanisms that make them resilient to failures. Currently, few platforms (especially among multiprocessors) support application-level fault tolerance. In a network-based computing environment, application resilience to failures can be supported without specialized enhancements to hardware or operating systems. Research is in progress to investigate and develop strategies for enabling applications to run to completion, in the presence of hardware, system-software, or network faults. Approaches based on checkpointing, shadow execution, and process migration are being investigated.

The performance and effectiveness of network-based concurrent-computing environments depend to a large extent on the efficiency of the support software and on the minimization of overheads. Experiences with the PVM system (see below) have identified several key factors in the system that are being further analyzed and improved so as to increase overall efficiency. Efficient protocols to support high-level concurrency primitives are a subgoal of the work being performed in this area.

Particular attention is being given to exploiting the full potential of fiberoptic connections. In preliminary experiments on an experimental fiberoptic network, several important issues have been identified. For example, with fiberoptics the operating-system interfaces network reliability characteristics, and factors such as maximum packet size are significantly different from those for Ethernet. When the concurrent-computing environment is executed on a combination of both types of network, the system algorithms have to be modified to cater to these differences in an optimal manner, and with minimized overheads.

Another issue to be addressed concerns data conversions that are necessary in networked heterogeneous systems. Heuristics that perform conversions only when necessary, and while minimizing overheads, have been developed, and their effectiveness is being evaluated. Recent experiences with a Cray-2 have also identified the need to handle differences in word size and precision when operating in a heterogeneous environment; general mechanisms to deal with arbitrary precision arithmetic (when desired by applications) are also being developed. A third consideration involves the efficient implementation of inherently expensive parallel-computing operations such as barrier synchronization. Particularly in an irregular environment (where interconnections within hardware multiprocessors are much faster than network channels, both in terms of bandwidth and latency), such operations can cause bottlenecks and severe load imbalances. Other distributed primitives for which algorithm development and implementation strategies are being investigated include polling, distributed fetch-and-add, global operations, automatic data decomposition and distribution, and mutual exclusion.

PVM

PVM[7] was produced by the Heterogeneous Network Project—a collaborative effort by researchers at Oak Ridge National Laboratory (ORNL), the University of Tennessee, and Emory University—specifically to facili-

tate heterogeneous parallel computing. PVM was one of the first software systems to enable machines with widely different architectures and floating-point representations to work together on a single computational task. PVM can be used on its own, or as a foundation upon which other heterogeneous network software can be built.

The PVM package is small (about 1 Mbyte 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, installation does not require special privileges on any of the machines, and can thus be performed by any user.

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.

Levels of heterogeneity

PVM supports heterogeneity at the application, machine, and network level. At the application level, subtasks can exploit the architecture best suited to their solution. At the machine level, computers with different data formats are supported, as well as different serial, vector, and parallel architectures. At the network level, different network types can make up a Parallel Virtual Machine, e.g., Ethernet, FDDI, token ring. With PVM, a user-defined collection of serial, parallel, and vector computers appears as one large distributed-memory computer; we use the term "virtual machine" to designate this logical distributed-memory computer. The hardware that composes the user's personal PVM may include any Unix-based machine on which the user has a valid login, and which is accessible over some network.

Individual PVM users can configure their own parallel virtual machines, which can overlap with other users' virtual machines. Configuring a personal parallel virtual machine involves simply listing the names of the machines in a file that is read when PVM is started. Applications, which can be written in Fortran 77 or C, can be parallelized by using message-passing constructs common to most distributed-memory computers. By sending and receiving messages, multiple tasks of an application can cooperate to solve a problem in parallel.

PVM supplies the functions for automatically starting up tasks on the virtual machine and for allowing the tasks to communicate and synchronize with each other. In particular, PVM handles all message conversions that may be required if two computers use different data representations. PVM also includes many control and debugging features in its user-friendly interface. For instance, PVM ensures that error messages generated on some remote computer get displayed on the user's local screen.

Components of PVM

The PVM system is composed of two parts. The first part is a daemon, called *pvmd* 3, which resides in all the computers that make up the virtual computer. (An example of a daemon program is *sendmail*, which handles all the incoming and outgoing electronic mail on a Unix system.) *pvmd* 3 is designed so that any user with a valid login can install this daemon on a machine. When a user wishes to run a PVM application, she or he executes *pvmd* 3 on one of the computers which, in turn, starts up *pvmd* 3 on each of the computers making up the user-defined virtual machine. A PVM application can then be started from a Unix prompt on any of these computers.

The second part of the system is a library of PVM interface routines. This library contains user-callable routines for passing messages, spawning processes, coordinating tasks, and modifying the virtual machine. To use PVM, application programs must be linked with this library.

Applications

Application programs that use PVM are composed of subtasks at a moderately high level of granularity. The subtasks can either be generic serial codes or br 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 executed; 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.

Applications access PVM resources via calls to PVM routines through a PVM 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 event synchronization.

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

PVM applications

Over the past few years, PVM applications have been developed in the following areas:

- Materials science
- Global climate modeling
- Atmospheric, oceanic, and space studies
- Meteorological forecasting
- 3-D groundwater modeling
- Weather modeling
- Superconductivity

- Molecular dynamics
- Monte Carlo CFD
- 2-D and 3-D seismic imaging
- 3-D underground flow fields
- Particle simulation
- Distributed AVS flow visualization

These implementations have been realized on various platforms.

Recently, ORNL material scientists and their collaborators at the University of Cincinnati, the Science and Engineering Research Council at Daresbury, UK, and the University of Bristol, UK, have been developing an algorithm for studying the physical properties of complex substitutionally disordered materials. Physical systems and situations in which substitutional disorder plays a critical role in determining material properties include high-strength alloys, high-temperature superconductors, magnetic phase transitions, and metal/insulator transitions. The algorithm under development is an implementation of the Korringa, Kohn, and Rostoker coherent-potential-approximation (KKR-CPA) method for calculating the electronic properties, energetics, and other ground-state properties of substitutionally disordered alloys.[8] The KKR-CPA method extends the usual implementation of density-functional theory (LDA-DFT)[9] to substitutionally disordered materials.[10] In this sense, it is a completely first-principles theory of the properties of substitutionally disordered materials, requiring as input only the atomic numbers of the species making up the solid.

The KKR-CPA algorithm contains several locations where parallelism can be exploited. These locations correspond to integrations in the KKR-CPA algorithm. The evaluation of integrals typically involves the independent evaluation of a function at different locations and the merging of these independent evaluations into a final value. The integration over energy is then parallelized. The parallel implementation is based on a master/slave paradigm that reduces memory requirements and synchronization overhead. In the implementation, one processor is responsible for reading the main input file, which contains the number of nodes to be used on each multiprocessor, as well as the number and type of workstations, the problem description, and the location of relevant data files. This master processor also manages dynamic load balancing of the tasks through a simple pool-of-tasks scheme.

Using PVM, the KKR-CPA code has achieved over 200 Mflops utilizing a network of ten IBM RS/6000 workstations. Given this capability, the KKR-CPA code is being used as a research code to solve important materials-science problems. Since its development, the KKR-CPA code has been used in various ways: to compare the electronic structure of two high-temperature superconductors, $Ba(Bi_{0.3}Pb_{0.7})O_3$ and $(Ba_{0.6}K_{0.4})BiO_3$; to explain anomalous experimental results from the high-strength alloy NiAl; and to study the effect of magnetic multilayers in CrV and CrMo alloys in connection with their possible use in magnetic storage devices.

The goal of the groundwater-modeling group at ORNL is to develop state-of-the-art parallel models for today's high-performance computers, which will enable researchers to model flow with higher resolution and greater accuracy than ever before. As a first step, researchers at ORNL have developed a parallel 3-D finite-element code called PFEM that models water flow through both saturated and unsaturated media. PFEM solves the system of equations

$$F\frac{\partial h}{\partial t} = \nabla \cdot [K_s K_r (\nabla h + \nabla z)] + q,$$

where h is the pressure head, t is time, K_s is the saturated hydraulic conductivity sensor, K_r is the relative hydraulic conductivity or relative permeability, z is the potential head, q is the source/sink, and F is the water capacity ($F = d\theta/dh$, with θ the moisture content) after neglecting the compressibility of the water and of the media.

Parallelization was accomplished by partitioning the physical domain and statically assigning subdomains to tasks. The present version uses only static load balancing and relies on the user to define the partitioning. In each step of the solution, the boundary region of each subdomain is exchanged with that of its neighboring regions.

Originally developed on an Intel iPSC/860 multiprocessor, a PVM version of PFEM was straightforward to create, taking an undergraduate student less than three weeks to complete. The PVM version of PFEM has been delivered to several members of the groundwater-modeling group for validation testing using networks of workstations while researchers await the availability of parallel supercomputers.

Status and availability

PVM was publicly released in March 1991 and has gone through a number of updates. Version 3.0 of the software has been tested with various combinations of the following machines: Sun 3, SPARCstation, Microvax, DECstation,

Workstations	Parallel Computers
Sun3	Intel Paragon
SPARCstation	Thinking Machines CM5
MicroVAX	Sequent Symmetry
DECstation	Intel iPSC/860
IBM RS/6000	Thinking Machines CM2
NEXT	Alliant FX/8
Silicon Graphics IRIS	Cray YMP and C90
HP 9000	Fujitsu VP2000
386/486 Unix boxes	Convex
	IBM 3090
	KSR-1

Table. Portability of PVM Source Code across UNIX workstations and parallel computers.

IBM RS/6000, HP-9000, Silicon Graphics IRIS, NeXT, Sequent Symmetry, Alliant FX, IBM 3090, Intel iPSC/860, Thinking Machines CM-2, KSR-1, Convex, and Cray Y-MP (see table).

Version 3.0 has a number of improvements over the previous Version 2.4. The new features are itemized below.

- Runs on Multiprocessors. Version 3.0 can operate on Paragon, CM-5, etc., using efficient vendor-specific calls.
- Dynamic Process Groups. Version 3.0 allows user-defined grouping.
- Dynamic Configuration. Users are to able to add and delete hosts.
- Multiple Message Buffers. Allows easier development of PVM math libraries, graphical interfaces, etc.
- Receive by source or type (or user-defined context) spawn multiple tasks with options for debugging and tracing, as well as packing and unpacking messages with stride, and more query functions.
- New naming convention for routines retains backwards compatibility with PVM2.4.

PVM is available through *netlib*. To obtain a description of PVM's features, such as a copy of the PVM User's Guide or source code, one simply sends e-mail to `netlib@ornl.gov` with the message `send index from pvm3`.

Future directions

The Heterogeneous Network Project is currently building a second package, called HeNCE (for Heterogeneous Network Computing Environment),[11] on top of PVM.

HeNCE simplifies the tasks of writing, compiling, running, debugging, and analyzing programs on a heterogeneous network. The goals are to make network computing accessible to scientists and engineers without the need for extensive training in parallel computing, and to enable users to use resources best suited for a particular phase of the computation.

In HeNCE, the programmer is responsible for explicitly specifying parallelism by drawing graphs that express the dependences and control flow of a program (see Fig. 3). HeNCE supplies a class of graphs as a usable, yet flexible, way for the programmer to specify parallelism. The user inputs the graph directly, 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 dependences 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 after their predecessors, as defined by dependence arcs, have been completed. Functions are mapped to machines based on a user-defined cost matrix.

The focus of this work is to provide a paradigm and graphical support tool for programming a heterogeneous

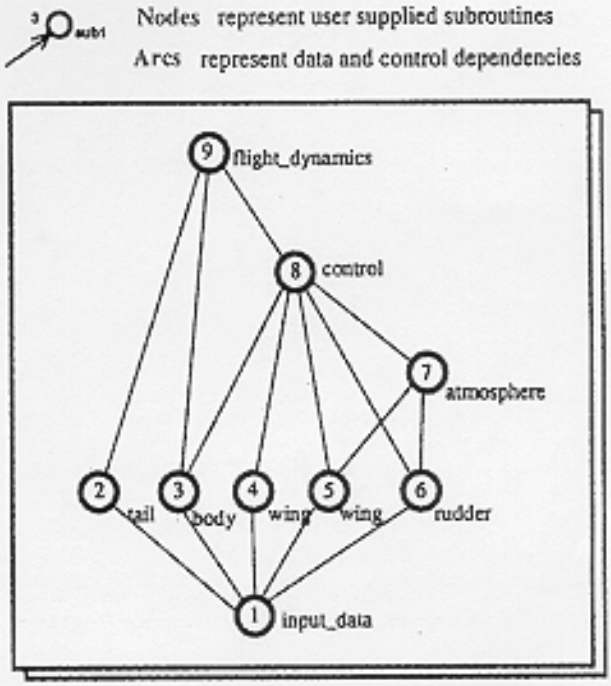

Fig. 3: Graph language constructs in HeNCE allow user to specify parallelism by drawing a graph of the application of algorithm.

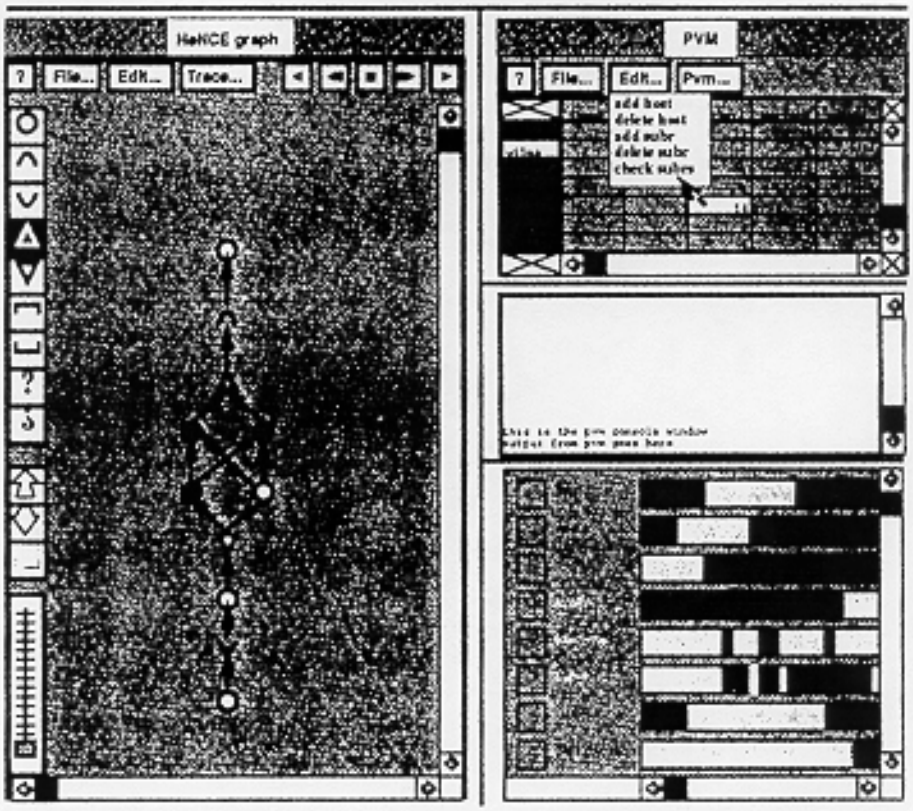

Fig. 4: Window-based environment supports programming in HeNCE.

network of computers as a single resource. HeNCE is the graphically-based parallel programming paradigm. In HeNCE, the programmer explicitly specifies the 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, as shown in Fig. 4.

An initial version of HeNCE has recently been made available through *netlib*. To obtain a description of its features, one should send e-mail to `netlib@ornl.gov` with the message `send index from hence`.

Both PVM and HeNCE offer researchers powerful means for attacking scientific computational problems through heterogeneous network computing. Continued research and development will ensure that this new area meets the needs of scientific computing in the 1990s and thereafter.

References

1. L. Patterson *et al.*, ACM SIGAPP, Indianapolis, 1993 (unpublished).
2. D. Gelernter, IEEE Trans. Comput. 19(8), 12 (1986).
3. J. Boyle *et al.*, *Portable Programs for Parallel Processors* (Holt, Rinehart, and Winston, New York, 1987).
4. V. Herrarte and E. Lusk, Argonne National Laboratory Report No. ANL-91/15, 1991 (unpublished).
5. R. Hempel, GMD Report, 1991 (unpublished).
6. A. Kolawa, in Proceedings of the Workshop on Heterogeneous Network-Based Concurrent Computing, 1991 (unpublished).
7. A. Beguelin *et al.*, Oak Ridge National Laboratory Report No. ORNL/TM-11826, 1991 (unpublished).
8. G. M. Stocks, W. M. Temmerman, and B. L. Györffy, Phys. Rev. Lett. **41**, 339 (1978).
9. Ulf von Barth, in *Density Functional Theory for Solids*, edited by P. Phariseau and W. Temmerman, NATO Advanced Study Institute, Series B: Physics (Plenum, New York, 1984).
10. D. D. Johnson *et al.*, Phys. Rev. B **41**, 9701 (1990).
11. A. Beguelin *et al.*, in *Proceedings of the Fifth SIAM Conference on Parallel Processing, Philadelphia, PA, 1991*, edited by Danny Sorensen (SIAM, Philadelphia, 1991).

Forum develops standard interface for message passing

During the past year there has been quite a bit of activity in the community to develop a standard interface for message passing.[1] The advantages of a message-passing-interface (MPI) standard would include portability and ease of use. In a distributed-memory communication environment, in which the higher-level routines and/or abstractions are built upon lower-level message-passing routines, standardization is particularly desirable. Furthermore, definiton of an MPI standard would provide vendors with a clearly defined base set of routines that they could implement efficiently, or in some cases provide hardware support for, thereby enhancing scalability. The standards activity goes by the name MPI Forum and includes the major hardware and software vendors, as well as researchers from universities and laboratories around the world.

Simply stated, the goal of the MPI Forum is to develop a standard for writing message-passing programs. The MPI should be a practical, portable, efficient, and flexible standard for message passing.

A complete list of goals of the MPI Forum follows.

- Design an application programming interface (not necessarily for compilers or a system implementation library).
- Allow efficient communication: Avoid memory-to-memory copying and allow overlapping of computation and communication and offloading to communication coprocessor, where available.
- Allow (but not mandate) extensions for use in heterogeneous environments.
- Allow convenient C, Fortran 77, Fortran 90, and C++ bindings for interface.
- Provide a reliable communication interface: Users need not cope with communication failures. Such failures are dealt by the underlying communication subsystem.
- Focus on a proposal that can be agreed upon in six months.
- Define an interface that is not too different from current practice, as in PVM, Express, Parmacs, etc.
- Define an interface that can be quickly implemented on many vendors' platforms, with no significant changes in the underlying communication and system software.
- Include only those functions in the MPI that are really necessary.

This standard is intended for use by anyone who wants to write portable message-passing programs in Fortran 77 and/or C. Potential users include individual application programmers, developers of software designed to run on parallel machines, and creators of higher-level programming languages, environments, and tools. In order to be attractive to this wide audience, the standard must provide a simple, easy-to-use interface for the basic user, while not semantically precluding the high-performance message-passing operations available on advanced machines.

The MPI standard is expected to include some or all of the following features.

- Point-to-point communication in a variety of modes, including modes that allow fast communication and heterogeneous communication
- Collective operations
- Process groups
- Communication contexts
- A simple way to create processes for the SPMD model
- Bindings for both Fortran and C
- A model implementation
- A formal specification

One of the objectives of the activity is to have a definition completed by Summer 1993. If you are interested in finding out more about the MPI effort, contact David Walker (walker@msr.epm.ornl.gov) at Oak Ridge National Laboratory.

Reference

1. Jack J. Dongarra, Rolf Hempel, Anthony J. G. Hey, and David W. Walker, Oak Ridge National Laboratory Report No. ORNL TM-12231, 1991 (unpublished).