Performance of the NAS Parallel Benchmarks on PVM Based Networks *

S. White, A. Ålund† and V. S. Sunderam
Department of Mathematics and Computer Science
Emory University, Atlanta, GA 30322, USA

Report RNR-94-008 May 1994

Abstract

The NAS parallel benchmarks are a set of applications that embody the key computational and data-movement characteristics of typical processing in computational aerodynamics. Five of these, the kernel benchmarks, have been implemented on the PVM system, a software system for network-based concurrent computing, with a view to determining the efficacy of networked environments for high-performance computational aerodynamics applications, and to experimentally investigate enhancements to the software infrastructure that optimize communication performance in such environments. We present results of porting and executing the NPB kernels in three different cluster environments using low- to medium-powered workstations on Ethernet and two types of FDDI networks. Our results indicate that mediocre to very good performance could be obtained despite the communications intensive nature of the applications. In most cases, we were able to achieve performance levels within an order of magnitude of a Cray Y/MP-1 on 8-workstation clusters via optimizations to the PVM infrastructure alone, i.e. with little or no algorithmic modifications. However, our results also indicate that further improvements are possible, and that network based computing has the potential to be a viable technology for high-performance scientific computing.

*Research supported by the National Aeronautics and Space Administration under grant NAG 2-828, and the National Science Foundation, under Award No. CCR-9118787.

†Swedish Institute of Applied Mathematics, Göteborg, Sweden; work done with financial support from NUTEK, Swedish National Board for Industrial and Technical Development, under grant 5321-93-04384, project no 9304384, while visiting Emory University, Fall 1993.

1 Introduction

Heterogeneous network-based concurrent computing is gaining widespread acceptance as a methodology for high-performance scientific applications in a variety of disciplines. One such application area is computational fluid dynamics(CFD), an important area within the domain of aerospace computing. The computational, memory, and data handling requirements in CFD applications are of necessity very large, in order to accurately simulate physical aerospace phenomena. For this reason, the mainstay hardware platform has traditionally been vector supercomputers, primarily multiprocessor Cray systems. Recently however, there has been an interest in the use of parallel processing for CFD applications, motivated both by the sheer performance offered by massively parallel processors (MPP's) as well as by their superior cost-effectiveness. In addition to MPP's, networked environments are also often suitable for parallelism; while their communication capabilities are only a fraction of what MPP's provide, they possess several other significant advantages. Among these are high availability, excellent price-performance characteristics, heterogeneity that can potentially be exploited by certain applications, and the existence of stable support software and development tools.

The typical methodology for parallel or concurrent processing in networked environments is based on a software framework that executes on participating hosts on a network, and emulates a "virtual parallel machine", essentially by implementing various distributed algorithms over standard network protocols. Examples of such software systems are PVM [5], Linda [6], Express [8], P4 and Parmacs [7], and numerous other systems [9]. Applications access the virtual machine via system-provided libraries that typically support process management, message exchange, and synchronization facilities. Most environments are based on explicit parallel programming using the message passing model; thus the programmer is responsible for parallelizing the application as well as for partitioning and workload allocation, and the interchange of data among processing elements. This mode of parallel/distributed computing is currently at a stage where its effectiveness as a suitable technology for high-performance computing is being evaluated.

In this paper, we report on such an exercise – the implementation and execution of a set of CFD benchmark applications on virtual parallel machines using the PVM software system. These applications are the five "kernel" benchmarks from the NAS Parallel Benchmark suite [2], an algorithmically specified collection of programs that are representative of real codes (or portions thereof) that are in production use in the aerospace community. Despite being termed kernels, the five applications rigorously exercise the processor, memory and, in the case of distributed-memory parallel machines, the communications capacity of any given system. The work presented in this paper was undertaken for several reasons. First, the kernels taken collectively, exercise all aspects of the underlying software system (in this case, PVM), especially robustness, portability, message handling limitations, process management, and synchronization. Second, the results of such an exercise would be valuable to the computational aeroscience researchers in evaluating network environments and

software systems for their computing needs. Finally, from the point of view of distributed computing research, the kernels highlight critical issues and limitations in the techniques used, and motivate the evolution of new strategies to address them. In the remainder of the paper, we present an overview of the NAS benchmarks and the PVM system, and report on preliminary porting and performance results in two environments. We then describe enhancements to the PVM system that significantly increases communication performance, and present revised results obtained in a third network computing environment based on a high-speed fiber-optic switch. We conclude with some general remarks on our findings and outline ongoing and future work.

2 The NAS Parallel Benchmarks

The NAS Parallel Benchmarks refer to a suite of applications devised by the Numerical Aerodynamic Simulation (NAS) Program of the National Air and Space Administration (NASA) for the performance analysis of highly parallel computers. The NAS Program is keenly interested in the highest level of computer performance, charged, as they are, "to provide the Nation's aerospace research and development community by the year 2000 a high-performance, operational computing system capable of simulating an entire aerospace vehicle system within a computing time of one to several hours"[3]. As there is no consensus as to the characteristics of general purpose parallel benchmarks, the NAS benchmarks focuses on the class of applications of most interest to the aerospace community. Therefore the NAS Parallel Benchmarks are rooted in the problems of computational fluid dynamics and computational aerosciences. Nevertheless, in general terms, they are valuable in the evolution of parallel processing technology, since they are rigorous and as close to "real" applications as may be reasonably expected from a benchmarking suite.

The benchmark suite consists of five "kernels" and three simulated applications which "mimic the computation and data movement characteristics of large scale computational fluid dynamics (CFD) applications" [1]. These benchmarks are specified only algorithmically, in [1], both to avoid the problems and difficulties associated with conventional approaches to benchmarking highly parallel systems, as well as to allow a certain amount of programming freedom and high level algorithmic refinement in the porting process. The positive aspects of this philosophy are that it permits programming freedom, optimization for specific machines or environments, and is not dependent on the portability of a 'code-based" benchmark. On the negative side, implementation for a given platform requires a great deal of effort; more importantly, subjective factors *viz* programmer qualifications and expertise, are introduced into the results. Complete details of the benchmark specifications as well as substantial descriptions may be found in [1, 2]; for the sake of completeness, we outline the five kernels that were ported to PVM and used in the experiments reported in this paper. In later sections, we provide additional algorithmic details to the extent necessary to understand and interpret our work.

2.1 The EP Kernel

Kernel EP is to execute 2^{28} iterations of a loop in which a pair of random numbers are generated and tested for whether Gaussian random deviates can be made from them according to a specific scheme. The number of pairs of the Gaussians in 10 successive square annuli are tabulated. The pseudorandom number generator used in this, and in all NAS benchmarks which call for random numbers, is of the linear congruential recursion type. This kernel falls into the category of applications termed "embarrassingly parallel", based on trivial partitionability of the problem, while incurring no data or functional dependencies, and requiring little or no communication between processors. It is included in the NPB suite to establish the reference point for peak performance on a given platform.

2.2 A 3D Multigrid Solver

Kernel MG adds substantial memory system and interconnection network demands to floating point demands near the order of Kernel EP. Kernel MG is to execute four iterations of the V-cycle multigrid algorithm to obtain an approximate solution to the discrete Poisson problem $\nabla^2 u = v$ on a $256 \times 256 \times 256$ grid with periodic boundary conditions. This application rigorously exercises both short- and long-distance communication, although the communication patterns are highly structured.

2.3 Conjugate Gradient

Kernel CG is to use the power and conjugate gradient methods to approximate the smallest eigenvalue of a symmetric, positive definite, sparse matrix of order 14000 with a random pattern of nonzeros. Although CG is normally considered in the context of solving linear systems, this particular benchmark version is a variant that subsequently computes eigenvalues. The communication patterns in this kernel are long-distance and unstructured – representative of typical unstructured grid computations.

2.4 3-D Fast Fourier Transform

This benchmark, termed Kernel FT, uses FFT's on a $256 \times 256 \times 128$ complex array to solve a 3-dimensional partial differential equation. Communication patterns in this kernel are structured and long distance in nature, and this benchmark represents the essence of many "spectral" codes or eddy turbulence simulations.

2.5 Integer Sort

Kernel IS is to perform 10 rankings of 2^{23} (8388608) integer keys in the range [0, 2^{19} (524288)). The keys are equally distributed in the local memories of the parallel ma-

chine; that is, each processor is assigned the same number of keys. Communication in this benchmark is frequent and relatively low-volume, and the pattern of communication is a fully connected graph. This kernel implements a sorting technique that is important in "particle method" codes.

3 The PVM System

PVM *(Parallel Virtual Machine)* is a software system that enables explicit message-passing concurrent computing on networks of heterogeneous machines. Details of the PVM system, the programming model, and experiences with its use have been reported in the literature, e.g. [5, 12, 11, 13]; in this section, we briefly outline some salient aspects of the system.

3.1 PVM Computing model

The basic computing model in PVM, which has remained semantically unchanged during its evolution, views applications as consisting of `components`, each representing a sub-algorithm; each component is an SPMD program, potentially manifested as multiple `instances`, cooperating internally as well as with other component instances via the supported communication and synchronization mechanisms. The unit of concurrency in PVM is a process, and dependencies in the process flow graph are implemented by embedding appropriate PVM primitives for process management and synchronization within control flow constructs of the host programming language. The implementation model, also unchanged from the original version, uses the notion of a "host pool", a collection of interconnected computer systems that comprises the virtual machine, on which *daemon* processes execute and cooperate to emulate a concurrent computing system. Applications request and receive services from the daemons; the facilities supported essentially fall into the categories of process management and virtual machine configuration, message passing, synchronization, and miscellaneous status checking and housekeeping tasks.

3.2 The PVM Programming Interface

In order to use the PVM system, applications are written as cooperating sequential programs, with embedded library calls to obtain services from the underlying distributed computing infrastructure. Typically, processes are organized as a master and multiple slaves, or as SPMD programs where one process takes responsibility for process initiation, collection of results, *etc.* Such a collection of cooperating processes utilize a system-supplied identifier, usually a numeric quantity, as the basis for both partitioning of the workload, and for addressing when sending and receiving messages. This model is very similar to that found in traditional message-passing parallel systems such as the Intel MPP family – except that a more general process structure is provided, and execution platforms may be heterogeneous. In the (common) case

of standalone SPMD programs therefore, control and communication structures in a PVM version are usually identical to its message-passing MPP counterpart.

3.3 PVM for High-Performance Computing

Much of the use of PVM (and other similar systems) has thus far been confined to homogeneous networked environments, primarily workstation clusters. Moreover, applications tend to be derived from corresponding MPP versions, since the two models are quite similar. Given these circumstances, there are a number of factors that must be considered, in terms of the levels of performance that may be reasonably expected for high-performance scientific computing.

- Workstations vary widely in their processing power and memory capabilities; further, as general purpose systems, they are subject to external competition for resources.
- Even in dedicated clusters, various background activities by the operating system, filesystem, and network software diminish the guarantee of obtaining a constant level of resources.
- Shared media networks, two orders of magnitude lower in capacity than MPP interconnects, are the norm in most cluster environments. Further, these are subject to the same influences discussed above for "within host" resources.
- The absence of "correct" partitioning and load balancing schemes. In a heterogeneous system, optimal algorithms for workload allocation are mandatory for achieving high performance – but as mentioned, most applications tend to be implemented directly from a homogeneous MPP version.

These issues are highlighted and reinforced in the discussion in the remainder of the paper, where our experiments with the NPB kernels on different virtual parallel machines are described.

4 PVM Implementation of the NPB Kernels

The five NPB kernels have been ported to execute under the PVM system; an overview of the kernels and our porting experiences are presented in this section. Several iterations were necessary because the release version (including the interface syntax and semantics) of PVM underwent substantial changes during the course of our exercises — we present the implementation process in the context of version 3.2 (Version 3.2), whose programming interface is likely to remain stable. We defer discussions of performance to the next section.

4.1 Kernel EP

There are two main portions to this benchmark – the random number generator, and the main processing loop which tests successive pairs of random numbers for the property that Gaussian deviates can be built from them. Parallelization of the problem is straightforward: each member in a group of processors works independently on a subset of the random numbers and the annuli counts are communicated at the end of processing. A simple front-end PVM process was designed to start the "worker" processes, communicate to them their starting and ending indexes in the loop, and receive their results at the end of computation. A static scheduling scheme was used, based upon an initial "pilot computation" lasting 10 seconds, to determine the relative CPU capacities of the machines involved.

4.2 Kernel MG

NAS specifies a discrete Poisson problem $\nabla^2 u = v$ with periodic boundary conditions on a $256 \times 256 \times 256$ grid. v is 0 at all coordinates except for 10 specific points which are $+1.0$, and 10 specific points which are -1.0. The V-cycle multigrid algorithm involves arithmetic on vectors u, v, and a work vector r. As the Intel version in Fortran77 was cumbersome to understand and port, a C version using multigrid recursion was implemented for PVM.

In multigrid recursion, the vectors that are operated upon "shrink" in each level of recursion by 2 in each dimension; that is, from a $256 \times 256 \times 256$ volume to a $2 \times 2 \times 2$ volume. As the recursion returns the size of the vectors expand back to $256 \times 256 \times 256$. The arithmetic performed are 27-point operations, in which a point, for example, in u is updated by arithmetic on the point in r with the same coordinates *plus* on the 26 points in r which differ by one unit in all combinations of one, two, and three indexes.

Therefore any scheme to data partition this problem, whether in one dimension or all three, will involve operations at border regions where a processor must "touch" the points which have been assigned to a neighboring processor. Two schemes are typically used in this type of parallel problem: keeping a neighboring processor's border points "in shadow" along with a scheme for updating them wholesale before they go out of date; or requesting the points on demand. The former scheme was chosen for our implementation. Also, we chose to data partition this problem in only one dimension, such that processor communication fits the ring topology. Figure 1 illustrates the arrangement of processors and the communication pattern for the four processor case.

There is substantial communication in this kernel between neighboring processors, because they must repeatedly communicate the values of vectors at border regions as they change. Message sizes are larger toward the top of the recursion. For example in the first communication, processor p sends the 258×258 (there is periodicity in all dimensions) double precision reals which are the "top" of its slice of the problem to

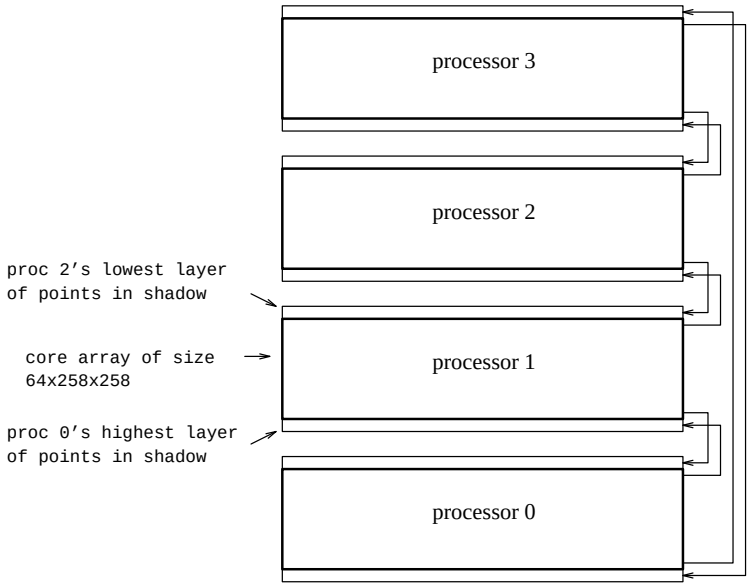

Figure 1: Kernel MG processor topology and communication pattern, four processor example

processor $p+1$, and processor $p+1$ sends the 258×258 double precision reals which are the "bottom" of its slice of the problem to processor p. The similar exchange occurs between processors p and $p-1$. Due to the periodicity, processors assigned partitions at top and bottom of the region also communicate as if their regions were physically neighboring. Total size of the communication just described, i.e. one of several border communications at the top of the recursion in each of the four multigrid iterations: over 4MB for a four processor set, over 8MB for an eight processor set, *etc.*

The most difficult aspect of the parallel implementation of this benchmark is that depending on the number of processors, near the bottom of the multigrid recursion there may be "too many" processors for the size of the vectors being worked on. A scheme where processors "leave" the computation at the lower levels of recursion and "rejoin" at the appropriate return point was implemented to overcome this obstacle.

4.3 Kernel CG

Kernel CG is to approximate the smallest eigenvalue of a symmetric positive definite sparse matrix A of order 14000. Ten iterations of the power method are employed to approximate the smallest eigenvalue; in each iteration of the power method, 25 iterations of the conjugate gradient method are used to solve the linear system $Az = x$. A is generated by an NAS-supplied Fortran routine `makea`. To ensure that the correct matrix was used, and because of time and expertise limitations, the `makea` subroutine was inherited with few modifications from the Intel version, which resulted in constraining our experiments to virtual machines with n^2 processes. The remainder of the implementation consisted of rewriting several sections of the code, especially

with regard to global communication primitives that are required to be emulated in PVM.

The primary design issue in Kernel CG is how to store the various work vectors used in the power and conjugate gradient methods. The operations on these vectors are two dot products, three vector-vector additions, and one matrix-vector multiply for each of the 25 conjugate gradient iterations in each of the 10 power method iterations. The NAS choice is to divide a vector into as many pieces as there are rows of processors (the pieces are either of equal size if the number of processor rows evenly divides 14000, or are of very nearly equal size), with each processor on a row holding an identical copy of that piece of the vector, as shown in figure 2.

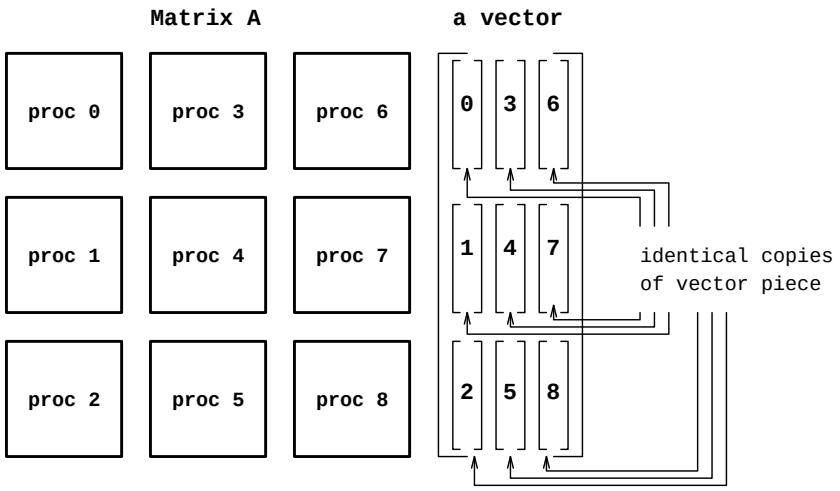

Figure 2: Matrix and vector organization in Kernel CG, nine processor case

Dot products are done in parallel: each processor on a row is assigned computation on a portion $1/P_{row}$ of the vectors in the dot product, with summation done by processor 0. Vector-vector additions are not done in parallel within a row because this would clearly incur communication costs which outweigh the overhead of each processor on a row performing the identical addition.

It is the scheme for performing the matrix-vector multiplication $q = Ap$ where efficiency is most crucial. Consider processor 2 in figure 2. It does not have the correct portion of the vector for its share of the matrix-vector multiplication; it needs the portion stored by any of the processors on the first row. Similarly, processor 6 also has the wrong vector portion for the matrix-vector multiply: it needs the portion stored by any of the processors on the third row. Therefore, before the matrix-multiplication can take place, the communication illustrated in figure 3 is required.

After the matrix-vector multiply by subblocks, processors on a column must sum their resultant vectors. But if this were done, then processors on a row would no longer have identical copies of that row's assigned portion of q. NAS takes advantage of the symmetry of A and sums the resultant vector along the row instead (figure 4).

The distribution of the vector p and the summing of the resultant vector q add up (in the 10x25 matrix-vector multiplications) to a total 130MB of communication

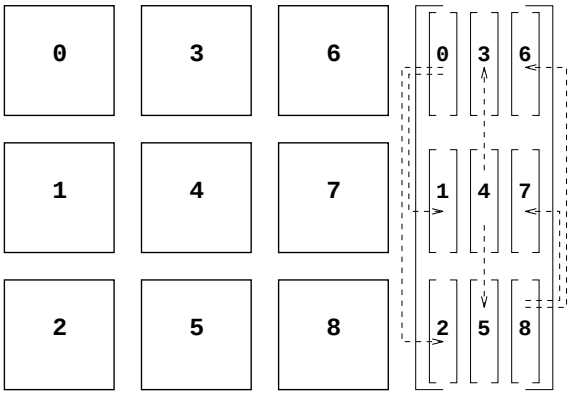

Figure 3: Communication in Kernel CG, nine processor case, before the matrix-vector multiply

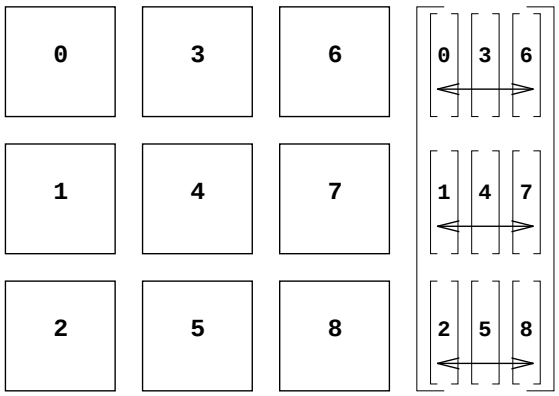

Figure 4: Kernel CG, nine processor case, sum of resultant vector along rows

in the four processor case, 250MB in the nine processor case, and 370MB in the 16 processor case.

4.4 Kernel IS

This benchmark performs 10 distributed rankings of 2^{23} (8388608) integer keys in the range [0, 2^{19} (524288)). The keys are equally distributed in the local memories of the parallel machine; that is, each processor is assigned the same number of keys (except for one processor which may have slightly fewer keys due to uneven division of N_{procs} into 2^{23}). The values of only two keys are changed between iterations, but no knowledge gained about the sequence of the keys in one iteration may be used in the next.

As the range of key values is 16 times smaller than the number of keys, and because the keys are generated by the linear congruential recursion random number generator which tends to generate more numbers in the middle of its range, it can be expected that the key values will have an uneven distribution within the key range. There may be 0 of one number, and 100 or more of another number in the specified

range. Therefore if our implementation strategy is based on assigning each processor a range of keys to rank, load balancing must be part of an optimal solution. NAS explicitly regulates the movement of the keys among processing elements; essentially, they may be duplicated but not removed from their initial placement, and the final sort at the conclusion of the 10 rankings must be done from the original mapping of keys to memory.

The Intel version of this benchmark was used to derive a PVM implementation, even though the work allocation assumes identically powered processors – incorrect in heterogeneous networks. The NAS strategy for performing the distributed ranking is as follows. We illustrate a four processor example, with data structure names depicted in figure 5. In the context of Kernel IS discussion in this and all other later chapters, "M" means 1024×1024. Each processor begins with 2M keys in the range [0, 1/2M) in the `keys` array. The goal is to fill in the `ranks` array with the global rank of each key. First, each processor calculates its distribution of key values by incrementing `distribution[i]` for each occurrence of key value `i` in the `keys` array. Each processor sends its `distribution` array to the root processor. The root processor sums these local distributions to calculate a global distribution of key values, and uses this to assign a range of key values—as balanced as possible by number of keys having values in that range—to each processor. These are the key values which a processor will be responsible for ranking. All ranges are communicated to all processors. Each processor then enters a loop where for each *other* processor in the processor set it packs a buffer of its keys that are assigned to that processor (`packed_keys`), packs each key's index in its sequence of keys (`packed_indexes`), and sends those arrays to the processor assigned to rank on that range.

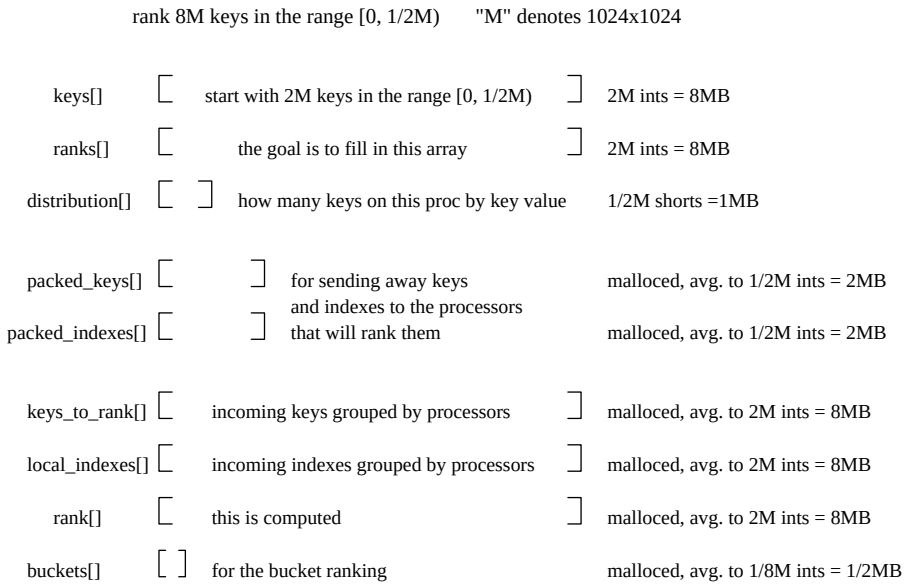

Figure 5: Kernel IS data structures, four processor example

Each processor receives incoming keys and indexes into the `keys_to_rank` and `local_indexes` arrays. The number of incoming keys each processor receives is broadcast to all processors, so that each processor knows how many keys there are in each

range of key values, and therefore where to begin assigning rank numbers. Each processor ranks approximately the same number of keys, give or take a small number due to a non-perfect breaking points in the distribution. A simple bucket rank is used to rank the keys, using the `bucket` array. On conclusion of the bucket ranking, each processor communicates the appropriate portions of the `rank` and `local_indexes` array to the native processors.

The communication required in this distributed ranking algorithm is extreme. Each processor sends key values and indexes for approximately $(N_{procs} - 1)/N_{procs}$ of its keys, and receives that many ranks and indexes back after the distributed ranking. This algorithm is clearly better suited to shared memory multiprocessors than distributed memory multicomputers. It is expected that the computation-intensive code executes in very much less time than the communications code, to the extent that on a distributed memory machine this kernel essentially reduces to a communications-system benchmark. In an attempt to reduce the communications volume in this application, we also implemented a version of Kernel IS that avoids sending indexes as the expense of additional memory and processing.

4.5 Kernel FT

The Kernel FT solves the discrete form of the PDE

$$\frac{\partial u(x,t)}{\partial t} = \alpha \nabla^2 u(x,t), x \in R^3, u \in C \text{ with initial values } u(x,0) = u_0(x)$$

using the Discrete Fourier Transform. After 3-D Fourier transformation of each side, this equation becomes

$$\frac{\partial v(z,t)}{\partial t} = -4\alpha\pi^2|z|^2 v(z,t) \text{ with the initial solution } v(z,0) = v_0(z)$$

where $v_0(z)$ is the 3-D Fourier transform of $u_0(x)$. The solution to this equation is

$$v(z,t) = e^{-4\alpha\pi^2|z|^2 t} v(z,0) = e^{-4\alpha\pi^2|z|^2 t} v_0(z)$$

where $v(x,t)$ is obtained as the inverse 3-D Fourier transform of v. The above is also true for the discrete form of the original PDE when the 3-D Discrete Fourier Transform (DFT) is applied. The serial version of Kernel FT first generates an initial $n_1 \times n_2 \times n_3$ array of complex numbers, $U(j,k,l), 0 \le j < n_1, 0 \le k < n_2, 0 \le l < n_3$ initialized with values from a pseudorandom number generator. Then 3-D DFT of U, called V, is computed using a 3-D FFT. Finally, for each $t \in [1,6]$ $W_{j,k,l}(t) = e^{-4\pi^2\alpha|\overline{j}^2+\overline{k}^2+\overline{l}^2|t} V_{j,k,l}$ is computed, where $\overline{j}$ is defined as j for $0 \le j < n_1/2$ and $j - n_1$ for $n_1/2 \le j < n_1$ and $\overline{k}$ and $\overline{l}$ similarly defined with n_2 and n_3. The inverse DFT is then applied to obtain the solution array $X_{j,k,l}$.

The PVM implementation on n_p processors organizes the $n_1 \times n_2 \times n_3$ array as n_3 $n_1 \times n_2$ dimensional arrays, with each processor responsible for n_3/n_p of these arrays. The 3-D DFT is implemented as 3 sweeps with a 1-D FFT routine, one in

each of the directions j, k, and l and the results are multiplied to form the 3-D DFT. The sweeps in the j and k directions can be done on the data initially assigned to each processor but for the sweep in the l-direction the array must first be transposed, involving communication as shown in Figure 6.

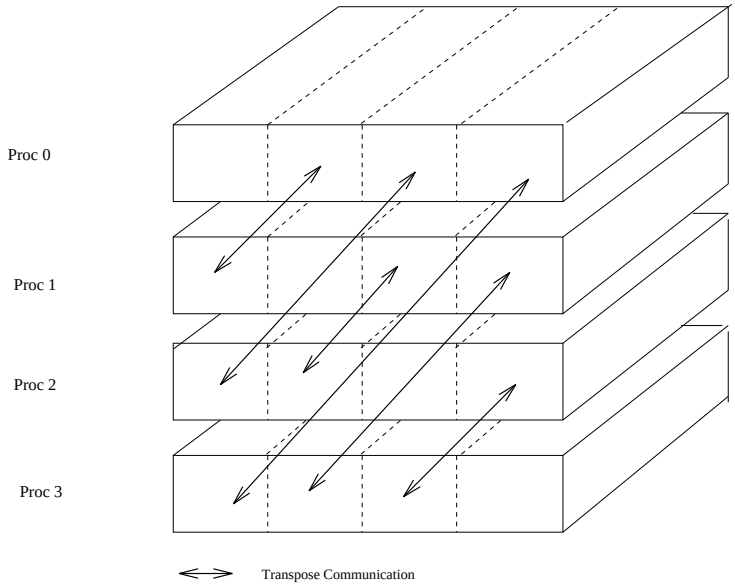

Figure 6: Kernel FT partitioning/communication, four processor example

The PVM implementation of the 3-D DFT requires two array transposes; therefore, the total communication volume is $2n_1n_2n_3(n_p - 1)/n_p$ complex numbers sent and received. For the $256 \times 256 \times 128$ problem on 8 nodes the communication volume for one 3-D DFT is 224 MByte (28 MByte / node), totalling 1568 MB for 6 timesteps. The memory requirement for this problem size is 49 MB for each of the 8 processes.

5 Preliminary Experiments

In the previous section, we have described the parallelization aspects of the five NPB kernels and outlined the structure of the PVM implementation of each. The communication structure, approximate communication volume, and memory requirements for each kernel have also been described. In this section we discuss the results of our performance experiments with these five kernels in three different PVM environments. The release version (3.2) of the PVM software, was used, and the default "daemon-based" communication scheme was selected. In the next section we describe a significant performance enhancement to the PVM communications scheme[1], and present updated results.

We wish to emphasize that the results reported herein are for *unoptimized* implementations of the NPB kernels – neither processor/memory optimizations for the

[1]Since the time of this exercise, the release version of PVM (3.2.6) has also incorporated a fast communication scheme

specific machines involved nor partitioning and scheduling optimizations believed to be essential for efficient heterogeneous execution were incorporated. A few changes that do have an impact on performance were incorporated, and these are described later, but by and large, the structure of the PVM implementations are based upon the corresponding Intel versions distributed by NAS.

5.1 Testbed Environments

The set of benchmark exercises discussed in this paper are unconventional, in the sense that they are not aimed at measuring performance of a specific machine, but rather of a methodology, *viz* network computing. In practical terms, this measures the effectiveness of the software infrastructure (PVM) and the particular network environment on which the experiments are conducted. In order to obtain an understanding of the effect of different hardware platforms while utilizing the same software system, we conducted our benchmark experiments on three environments, each with its unique characteristics. While these were selected primarily on the basis of availability, they do nevertheless represent major classes of cluster platforms, especially with respect to the communications network. They are outlined below, with an indication of their throughput and latency (process-to-process time, including PVM overhead, for a zero length message).

- Low-end workstations on low-speed shared medium networks: This environment consists of 16 diskless Sun SS1+ workstations, each with 16 MB of memory and 32 MB of virtual memory(swap space), at the Emory University Parallel Processing Lab. They are isolated from the remainder of the department/campus network, and it is possible to use them in dedicated mode – thus achieving a reasonably well controlled environment, where the only external influences are background OS and network activities. These machines are connected by 10Mbps Ethernet on which the maximum achievable throughput is about 1000 KB/s, and the minimal latency is of the order of 2 milliseconds.

- Medium-performance workstations on high-speed shared networks: This environment consists of seven IBM RS/6000 model 560 computers, and one RS/6000 model 320 computer, each with 32 MB of memory, and 64 MB of swap space, connected by FDDI. FDDI is an optical token-ring network that theoretically offers 100 Mbps, but because it is a shared-ring topology, achievable throughput is usually substantially lower. Minimal latencies are of the order of 1 millisecond with IBM RS/6000 model 560's. This platform was made available to us by the Utah Supercomputing Institute, and is subject to external loads as well as periods of unavailability.

- Medium-performance workstations on high-speed switched medium networks: Figure 7 shows the organization of a high performance heterogeneous computing testbed (termed HEAT) that has recently been installed at Sandia National Labs to which we have been given access.

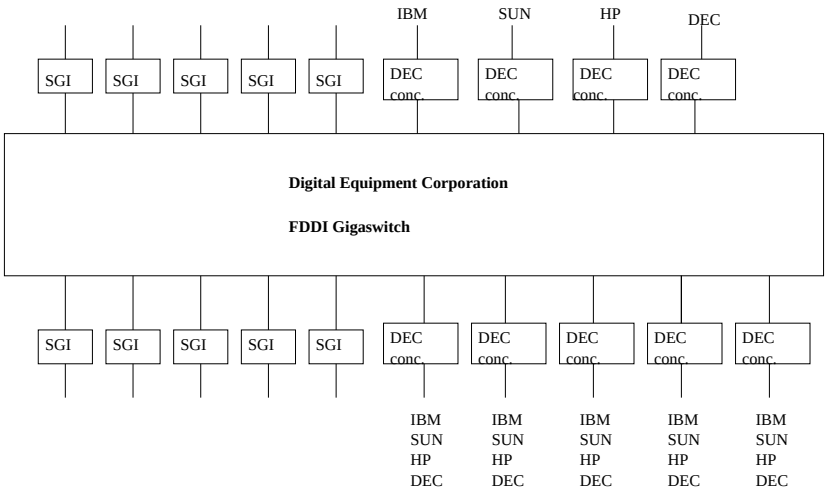

Figure 7: Sandia Labs Heat Testbed

Although the HEAT has high-end workstations, only the medium powered SGI R4000 workstations were used in our experiments, both because of their direct interconnection via the switch, and due to some instabilities and configuration factors on the other machines. These SGI workstations are configured with 64 MB of memory and 128 MB of swap space; minimal message latency is about 1 millisecond. The switch, it is believed, is capable of 100 Mbps across 11 simultaneous connections. While the HEAT testbed was not available in dedicated mode, it was possible to access it at times of low external loads.

5.2 Performance Results

In this section we present performance results for the five kernel benchmarks on each of the three testbeds, *using an unmodified version of PVM version 3.2*. The experiments were conducted during idle periods, with background CPU and network loads estimated at 5—10%, although the environments were not controlled. We present the results in tabular form, specifying the total *elapsed time* in seconds for each platform and, where appropriate, the total communication volume, the time for communication related activities alone, and the total number of messages exchanged. Also shown for comparison in each table are the reported times on a single processor of the Cray Y-MP, and those for the Intel iPSC/860 hypercube. These latter results were obtained from a recent NAS report [4], and represent the best, optimized, times available as of the time of writing of that report. We emphasize again that the PVM results are for unoptimized versions, except for a few gross changes described in the previous section. In all cases, native Fortran and/or C compilers were used with only the "-O2" optimization flag.

Platform	Time (secs)
16 SS1+ Enet	1603
8 RS6000 FDDI	342
8 SGI Gswitch	446
Cray Y-MP/1	126
i860/32	102
i860/64	51
i860/128	26

Table 1: Kernel EP on native PVM

5.2.1 Performance of Kernel EP

The Kernel EP benchmark was executed on all three environments for the "full" problem size; the version that was run includes an initial 10-second pilot computation to determine processor capacities on the basis of which workload is allocated. Table 1 lists the observed timings. While these results are as expected, two facts are noteworthy. The first is that in the absence of communication, clusters of eight workstations are capable of delivering a third of Cray Y-MP/1 performance. The other point (not explicit in the table), is that despite a dedicated environment and the use of a "one-time dynamic" load-balancing scheme, load imbalances of the order of 10–15% were observed in all environments. As far as we were aware, the environments were "dedicated"; therefore, background operating system activity and the *nice* policy are the only explanations for this behavior.

5.2.2 Performance of Kernel MG

On the SS1+ cluster, owing to memory limitations, we were only able to execute a reduced version of MG that uses a $128 \times 128 \times 128$ grid. It was possible to execute the full-sized problem on the other two environments. Two small modifications were attempted for Kernel MG. The first was to change the order of message passing to permit greater overlap of computation and communication - performance improvement of the order of only 6-9% was obtained by doing so. The second was to implement a token scheme for the shared Ethernet to reduce contention. This scheme dramatically reduced the number of Ethernet collisions (from 54% to 16% of packets output), but did not improve overall execution time – we conjecture that this is due to the relatively large overhead of the token passing scheme itself. The results of MG performance without algorithmic alterations, but including the overlap scheme, are shown in table 2.

The results shown in 2 indicate that 8-processor clusters perform more than an order of magnitude slower than a single processor of the Cray Y-MP, and 25 times

Platform	Time (secs)	Comm. Volume	Comm. Time(secs)	Number of msgs
16 SS1+ Enet	198*	96 MB	154	2704
8 RS6000 FDDI	229	192 MB	162	1808
8 SGI Gswitch	264	192 MB	112	1808
Cray Y-MP/1: 22 secs; i860/128 : 8.6 secs				

* Reduced problem size ($128 \times 128 \times 128$)

Table 2: Kernel MG on native PVM

Platform	Time (secs)	Comm. Volume	Comm. Time(secs)	Number of msgs
16 SS1+ Enet	701	370 MB	480	37920
4 RS6000 FDDI	285	130 MB	192	7116
9 SGI Gswitch	130	250 MB	101	19756
Cray Y-MP/1: 12 secs; i860/128: 7.0 secs				

Table 3: Kernel CG on native PVM

slower than a 128-processor Intel hypercube. The table also indicates that communication time accounts for upto two-thirds of the execution time.

5.2.3 Performance of Kernel CG

Table 3 shows performance results for the conjugate gradient benchmark under PVM. As previously mentioned, we encountered some difficulty in porting this application from the serial and Intel versions of the NAS-supplied code, which resulted in a probably inefficient implementation – but executes on n^2 processors (rather than on 2^n processors).

With reference to table 3, it can be seen that the parallel CG algorithm results in increased communication volume with an increase in the number of processors - a serious drawback especially for shared-network based concurrent computing. However, a small number of powerful workstations works well, as the timings for the 4-processor FDDI and 9-processor switched networks indicate. Once again, communication times dominate the overall execution times, accounting for 77% of the time in the 9-processor switched network case. As with Kernel MG, the fastest PVM configuration executes about an order of magnitude slower than a Cray Y/MP-1.

Platform	Time (secs)	Comm. Volume	Comm. Time(secs)	Number of msgs
16 SS1+ Enet	607*	150 MB	595	10115
8 RS6000 FDDI	674	560 MB	610	2491
8 SGI Gswitch	770	560 MB	720	2491
Cray Y-MP/1: 11 secs; i860/32: 26 secs i860/64: 17 secs; i860/128: 14 secs				

* Reduced problem size (2^{21} keys in the range 0 to 2^{19})

Table 4: Kernel IS on native PVM

5.2.4 Performance of Kernel IS

This benchmark is particularly ill-suited for networked environments, involving large communication volume and frequency, and minimal computation. As a result, IS performs poorly in all three environments when using the daemon-based PVM communication scheme, as shown in Table 4. In fact, communication time comprises over 90% of the overall execution time in all cases! Owing to memory limitations, the SS1+ network was only used for a reduced size version of the problem. Results are shown for the version of IS incorporating one modification, *viz.* avoiding the exchange of array indices at the expense of relatively small extra memory and processing.

5.2.5 Performance of Kernel FT

The FT kernel is most demanding in terms of communication volume. However, this communication is in the form of large messages of relatively low frequency – a characteristic that contributes to achieving high bandwidth. Memory limitations again constrained our ability to run the full problem size on the 16 SS1+ environment on which a reduced problem had to suffice. The parallel PVM version of FT was derived directly from the Intel version, with some portions obtained from the sequential NAS version. Results are shown in Table 5 – but exhibit rather disappointing performance, possibly indicating that even bandwidth-limited applications are unable to perform well under the unmodified PVM system.

5.3 General Comments

The results presented in the previous subsection are mediocre at best. However, notwithstanding the general reasons of communication bottlenecks, PVM overheads, and timeshared machines for this poor showing, we should still expect to see somewhat better performance. Although too simplistic, we may use some crude approximations to determine the extent of the performance deterioration due to communication in the following manner. For example, in the MG benchmark, 192 MB of data are

Platform	Time (secs)	Comm. Volume	Comm. Time(secs)	Number of msgs
16 SS1+ Enet	717*	420 MB	502	3450
8 RS6000 FDDI	645	1500 MB	395	826
8 SGI Gswitch	1070	1500 MB	516	826
Cray Y-MP/1: 29; i860/128: 10 secs				

* Reduced problem size ($128 \times 128 \times 128$)

Table 5: Kernel FT on native PVM

communicated in 162 seconds on shared FDDI and in 112 seconds on switched FDDI. These measurements suggest that an average throughput of 1.2 and 1.7 MB/sec on the two networks respectively – 10% and 15% of the rated capacities. Even accounting for the fact that rated network capacities are hard to achieve, this appears to be abysmally low.

6 Optimizing Message Passing

Preliminary benchmarking of the NAS kernels on PVM, as described in the previous section, suggested that only a fraction of available network capacity was being attained, thereby resulting in poor to mediocre performance of these benchmarks in cluster environments. As this project is aimed at evaluating and enhancing the software infrastructure to obtain better performance (as opposed to algorithm modifications or compiler optimizations), we focused our efforts on techniques to improve PVM communication performance.

Message passing in the PVM system is a three-stage process by default, involving "store-and-forward" by the daemons. This scheme has its advantages in terms of scalability and statelessness, but performs poorly. In an attempt to overcome this drawback, we have designed and implemented an enhanced communication scheme for PVM; in this section, we describe this mechanism and present updated performance results[2].

6.1 Fsend and Frecv

From the programming interface point of view, the enhanced message passing scheme, accessible via the `pvm_fsend()` and `pvm_frecv()` calls, permit the direct transfer of user program data without requiring buffer initialization and packing. However, data conversion can still be included if communicating between different architectures,

[2]As mentioned, a later version of PVM (3.2.6) supports similar functionality and performance via a toggled option. Thus the results presented may also be obtained using the release version of the PVM software

Platform	Throughput (kB/sec)			
	1 byte	100 bytes	10kB	1MB
SS1+ Ethernet				
Daemon	0.06	12.88	263.41	358.48
Fsend	0.49	81.79	902.42	1003.87
TTCP	0.65	130.45	965.04	1125.24
RS6000 FDDI				
Daemon	0.09	20.67	374.04	711.58
Fsend	0.82	112.79	1568.40	2285.89
TTCP	1.85	325.40	2573.00	2918.20
SGI Gigaswitch				
Daemon	0.21	38.92	406.04	483.58
Fsend	1.11	102.79	3400.42	9550.87
TTCP	2.15	500.40	9203.00	9624.20

Table 6: Point-to-point communication bandwidth

thus retaining data heterogeneity but not heterogeneity of message content. The `pvm_fsend()` and `pvm_frecv()` mechanisms are built on standard TCP internet protocols and are manifested as a separate but non-intrusive library in the PVM system. This library is tailored and optimized for each host operating system, in a manner that achieves near-optimal performance for a given host environment. The limitations on scalability remain, but currently, this is of the order of 64 hosts and above, and we are working on enhancements that would eliminate this limitation altogether.

The `pvm_fsend()` and `pvm_frecv()` library was implemented and tested on a variety of environments and networks. In particular, it has been evaluated on the platforms described in this paper, both using simple communications testing programs, as well as end-applications. Table 6 indicate the performance of this communications scheme for simple point-to-point data transfer, for a variety of message sizes. Also shown for reference, are the corresponding values for dameon-based PVM communication, and for a standalone benchmarking program, *viz.* TTCP.

From the table it can be observed that the enhanced communication scheme delivers throughputs several times as much as the daemon based communication. However, it also indicates that except for large messages, even the enhanced communication mechanism delivers only a fraction of the throughput actually attainable by software as indicated by the reference TTCP numbers which, incidentally, is of the order of 70-95% of the theoretical maxima. This exercise of evaluating communications performance and the development of an experimental enhanced communication scheme was productive for two important reasons. First, it has resulted in a robust and effective auxiliary library that performs, in some cases, several times better than the native PVM system. Second, it has highlighted the importance and benefits of tuning communications software both for specific machines and their operating systems, as

Platform	Time (secs)	Comm. Volume	Comm. Time(secs)	Number of msgs	Idle Time(secs)
16 SS1+ Enet	138* (198*)	96 MB	85 (154)	2704	48
8 RS6000 FDDI	110 (229)	192 MB	52 (162)	1808	30
8 SGI Gswitch	168 (264)	192 MB	81 (112)	1808	50
Cray Y-MP/1: 22 secs; i860/128 : 8.6 secs					

* Reduced problem size ($128 \times 128 \times 128$)

Table 7: Kernel MG on enhanced PVM

well as for different types of local area networks.

6.2 Updated Performance Results

In this section we report on performance measurements obtained by re-executing the NPB kernels on PVM, using the enhanced communications scheme[3]. In the tables, we also include the total time and communication time for the previous experiments (in parentheses) for convenient comparison. We also include an additional measure in the following set, *viz* the "maximum idle time". This is the largest idle time accumulated by any PVM process in the collection; idle time refers to the time that a process is blocked on receives, i.e. waiting for incoming messages that were a precondition to performing further computation. This metric approximately reflects the load imbalance in a given application algorithm or the parallelization thereof.

6.2.1 Performance of Kernel MG

It may be seen by comparing table 2 and 7 (or the new and old (parenthesized) times in columns 2 and 4) that the enhanced communication mechanism resulted in an improvement of 50% for the SS1+ platform, 100% for the RS6000's, and 60% for the SGI's. In two environments, all the gains were due to communication time, which was reduced by 50% to 300%. Focusing on the idle time column, it may be seen that unproductive time spent in blocked receives are a significant fraction of the total communication times, indicating that further optimizations may be possible by implementing a better load balancing scheme, or by eliminating serial bottlenecks.

6.2.2 Performance of Kernel CG

The improvements in performance for Kernel CG obtained by using enhanced PVM communications are not as dramatic as in MG for the SS1+ platform, and of the order of 35% for the other two platforms. However, communication times in the two FDDI environments are reduced by a factor of two, and as the idle time column shows,

[3]EP results are unchanged and are therefore not reported

Platform	Time (secs)	Comm. Volume	Comm. Time(secs)	Number of msgs	Idle Time(secs)
16 SS1+ Enet	605 (701)	370 MB	404 (480)	37920	390
4 RS6000 FDDI	203 (285)	130 MB	101 (192)	7116	88
9 SGI Gswitch	108 (130)	250 MB	46 (101)	19756	42
Cray Y-MP/1: 12 secs; i860/128: 7.0 secs					

Table 8: Kernel CG on enhanced PVM

Platform	Time (secs)	Comm. Volume	Comm. Time(secs)	Number of msgs	Idle Time(secs)
16 SS1+ Enet	398* (607*)	150 MB	375 (595)	10115	370
8 RS6000 FDDI	318 (674)	560 MB	171 (610)	2491	129
8 SGI Gswitch	258 (770)	560 MB	140 (720)	2491	105
Cray Y-MP/1: 11 secs; i860/32: 26 secs i860/64: 17 secs; i860/128: 14 secs					

* Reduced problem size (2^{21} keys in the range 0 to 2^{19})

Table 9: Kernel IS on enhanced PVM

potential for further reduction exists. Another interesting observation concerns the number of messages transmitted in each case in relation to the total communication volume. For example, in the 9 processor SGI platform, CG exchanges 19756 messages for a total volume of 250 MB, while MG exchanges only 1808 messages for 192 MB. The enhanced communication mechanism attempts to optimize performance for small messages as well, thereby achieving performance gains even for latency-limited applications. Notwithstanding these improved overall results, idle column with times of the order of 60% to 90% of total communications times in CG is the highlight of this table.

6.2.3 Performance of Kernel IS

Kernel IS shows a uniform twofold (or better) performance with the enhanced communication scheme, as compared to the native PVM measurements. This improvement is definitely attributable to increased communication efficiency, especially in the case of the two FDDI networks. On Ethernet, the difference is not so marked, perhaps owing to the fact that collisions (as observed by network monitoring tools) on the network continue to cause degradation. The other noteworthy aspect of this table is that idle times are a very high proportion of the total communication time, in some cases worse than those for CG and MG. It is believed that this behavior is a result of the very small amounts of processing involved in IS, as compared to the frequency, pattern, and volume of communication involved.

Platform	Time (secs)	Comm. Volume	Comm. Time(secs)	Number of msgs	Idle Time(secs)
16 SS1+ Enet	520* (717*)	420 MB	385 (502)	3450	266
8 RS6000 FDDI	412 (645)	1500 MB	220 (395)	826	112
8 SGI Gswitch	228 (1070)	1500 MB	130 (516)	826	34
Cray Y-MP/1: 29; i860/128: 10 secs					

* Reduced problem size ($128 \times 128 \times 128$)

Table 10: Kernel FT on enhanced PVM

6.2.4 Performance of Kernel FT

The results for FT are also illustrative in that they indicate the degree to which performance can be improved by using faster communication schemes for very-high message-volume applications. Execution time reductions are very significant for this benchmark, especially in the case of the switched FDDI network, where a four-fold improvement in both overall time and communication time may be observed. This marked improvement in performance and high network utilization is attributable to the fact that infrequent, large messages are the norm in this application. This effect is highly pronounced in a switched network but not as remarkable in shared networks; understandable considering that the FT benchmark is synchronous in nature – all processes perform approximately equal communication and then simultaneously exchange large messages. Another effect of exchanging large messages at relatively infrequent intervals is that idle times attributable to blocked receives and load imbalances are not as severe in this application.

6.3 Communications Efficiency

From the foregoing results, it can be seen that the enhanced communication scheme reduced communication times significantly, often achieving near-maximal communications efficiency. For example, in MG, the SS1+ system transfers 96MB in 85 seconds; an aggregate rate of 1.12MB/sec, or 95% of the theoretical maximum. The Gigaswitch version of CG transfers 250MB in 46 secs for medium size messages, also close to the best attainable in software. IS on Ethernet runs at 91% communication efficiency. FT on FDDI gets over 60aggregate utilization (twice the software maximum for a point-point raw test) due to high offered load on a shared-access network. However, there is still room for improvement, as discussed in the next section.

7 Discussion

In this paper, we have described porting exercises and performance measurements of the NAS parallel benchmark kernels to distributed computing environments based on

the PVM system. This work has shed light on a number of issues that influence high-performance concurrent computing in networked environments. Many of these issues deal with communications performance. By analyzing data transfer bottlenecks in the release version of PVM and devising enhancements to the system to overcome these drawbacks, we have obtained significantly better performance in all three network platforms *viz* shared-Ethernet, shared-FDDI, and switched FDDI. In addition, these experiments have also highlighted other factors that should be addressed in order to obtain optimal performance, especially load balancing, and we intend to investigate strategies to address them in future research.

7.1 NPB Codes on PVM

From the results presented in this paper, it may be seen that the clusters we have used are generally an order of magnitude slower than the standard yardstick for CFD codes, namely one processor of the Cray Y-MP. Eight processor clusters of RS6000's over FDDI and SGI's over switched FDDI perform about 7 to 10 times slower than a Cray Y-MP/1, with better ratios when communication volume and frequency is lower. This is to be expected of course, but certain other aspects are also worthy of note. First, it is apparent that for codes such as the NPB kernels that exchange large volumes of data, fast networks are essential for good performance. However, shared-FDDI networks perform as well as switched networks – provided the communication volume is below a certain threshold, as indicated by the superior performance of kernel MG in FDDI ring networks [4]. When communication volume is large, as in the case of FT, switched fiber networks appear to perform twice as well for the same number of processors.

It should also be kept in mind that our implementations were naive, first-cut efforts. One probable shortcoming of the PVM versions of the NPB codes is our use of the Intel codes as our porting basis. In addition to uniform partitioning and workload allocation, the assumption of a specific interconnection network with a known, dedicated bandwidth is bound to have had a negative effect on our performance results. We believe that a "first-principles" implementation of the NPB kernels for PVM, with specific tailoring of the computational algorithms as well as partioning and communication schemes to the cluster processors, the network, and the PVM system, will achieve significantly better results.

Another fact to be considered when interpreting our results concerns the processors that were used in the experiments. The switched FDDI network delivers the best communications performance, but unfortunately, we were only able to use medium powered SGI processors on this system – it is interesting to speculate on the results had we used Alpha-class workstations that are five times as powerful. Moreover, both the RS6000 network and the SGI platform were subject to external influences and various configuration and operating parameters (e.g. memory, swap space, spontaneous OS activity, *nice* policy, network service daemons *etc.*) over which we had no control.

[4]Some of this better performance is of course due to the faster processors used

These effects are manifested to some extent in the idle times reported in the previous section; an externally loaded processor lags behind the others in a parallel computation, thus causing the other processors to accumulate "receive blocking" times. It is suspected that performance improvements of the order of 20% or more over what we have reported might have been possible, if these environments had been used in an absolutely dedicated mode. In some sense however, such variability and lack of control is a *feature* of the network computing methodology, not a drawback. Therefore dynamic load balancing schemes and better partitioning methods are probably required to exploit the full potential of network computing.

7.2 Future Work

As can be seen from the foregoing discussions, several issues still remain to be addressed with regard to network computing infrastructures to improve their efficacy for high-performance computing. We intend to continue to work on the following critical aspects as part of our research in heterogeneous concurrent computing.

- Profiling and Visualization: Interpretation of the experiments presented in this paper has been done primarily on the basis of the end-results. In order to gain deeper insight into the behavior of the applications and software framework, graphical visualization and profiling would be extremely valuable. We are currently working on enhancements to the PVM system to permit visualization of network applications using the ParaGraph tool [19]. The major effort required in this regard is necessitated by the asynchrony and drift of system clocks in typical cluster environments that are manifested as causality violations in program trace files; we are developing an algorithm to adjust for these "tachyons" in a manner that preserves the time-sequence of events and only minimally alters time scales.

- Partitioning and Workload Allocation: High and dynamic variability in the amount of resources available in different parts of networked environments (different processors, network connections, software) strongly suggest that alternative partitioning and load balancing schemes are a pre-requisite to optimal performance. We are beginning the design of an object-based methodology whereby smaller granularity workload units may be generated and dynamically allocated according to instantaneous conditions that prevail in a cluster or heterogeneous network. We intend to manifest these objects as threads or light-weight processes whereby multiple threads would be sited within a single process; this strategy is expected to achieve better latency hiding as well as greater overlap of computation and communication, in addition to more adaptable workload allocation.

- Alternative Protocol Architectures: In the present implementation of PVM, standard network protocols (i.e. TCP/UDP/IP)are used; these are thought to be overly general and in many respects, unsuitable for parallel distributed

computing. We are evolving an alternative protocol scheme where PVM and application programs directly access the data link interface of high-speed local networks, thus bypassing much of the protocol processing and resultant overheads. In a preliminary implementation of this scheme, we have obtained latencies that are less than one-half of the minimal obtainable while using TCP/IP, and throughput improvements of the order of 35%. We intend to develop this protocol suite [18] as a library and modify the PVM system to operate over this suite when possible.

8 Acknowledgements

We are grateful to Rod Fatoohi and Horst Simon of NASA Ames for their assistance in understanding the NAS NPB kernels and their help in our porting efforts. We appreciate the assistance and cooperation of Edgar Leon of Emory University who constructed, maintained, and supported the 16 SS1+ cluster, and to Michael Pernice of the University of Utah for permitting access to an already busy FDDI cluster. We express our thanks to Ray Cline, Rob Armstrong, Rich Guy, and Cathy Houf of Sandia National Laboratories for access to the state-of-the-art HEAT testbed and for their willingness to cater to some of our special requirements.

References

[1] D. Bailey, J. Barton, T. Lasinski, and H. Simon, eds. "The NAS Parallel Benchmarks". NASA TM 103863, Moffett Field, California: NASA Ames Research Center, July 1993.

[2] D. Bailey, E. Barszcz, et al. "The NAS Parallel Benchmarks." *International Journal of Supercomputer Applications*, Vol. 5, No. 3, pp.63-73, Fall 1991.

[3] NAS Systems Division, *Numerical Aerodynamic Simulation Program Plan*. Moffett Field, California: NASA Ames Research Center, 1988.

[4] D. Bailey, E. Barszcz, L. Dagum, and H. Simon, "NAS Parallel Benchmark Results", Report RNR-94-006, NASA Ames, Moffett Field, California March, 1994.

[5] V. S. Sunderam, "PVM : A Framework for Parallel Distributed Computing", *Journal of Concurrency: Practice and Experience*, **2**(4), pp. 315-339, December 1990.

[6] N. Carriero and D. Gelernter, "Linda in Context", *Communications of the ACM*, **32**(4), pp. 444-458, April 1989.

[7] R. Butler and E. Lusk, "User's Guide to the P4 Programming System", Argonne National Laboratory, Technical Report ANL–92/17, 1992.

[8] A. Kolawa, "The Express Programming Environment", *Workshop on Heterogeneous Network-Based Concurrent Computing*, Tallahassee, October 1991.

[9] Louis Turcotte, "A Survey of Software Environments for Exploiting Networked Computing Resources", MSSU-EIRS-ERC-93-2, Engineering Research Center, Mississippi State University, February 1993.

[10] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. "A machine-independent communication library", In J. Gustafson, editor, *The Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, pp. 565-568, P.O. Box 428, Los Altos, CA, 1990. Golden Gate Enterprises.

[11] G. A. Geist and V. S. Sunderam, "Network Based Concurrent Computing on the PVM System", *Journal of Concurrency: Practice and Experience*, **4**(4), pp. 293-311, June 1992.

[12] B. Schmidt and V. S. Sunderam, "Empirical Analysis of Overheads in Cluster Environments", *Journal of Concurrency: Practice and Experience*, (to appear), 1993.

[13] A. Beguelin, J. Dongarra, G. Geist, R. Manchek, and V. Sunderam, "Solving Computational Grand Challenges Using a Network of Supercomputers", *Proceedings of the Fifth SIAM Conference on Parallel Processing*, D. Sorensen, ed., SIAM, Philadelphia, 1991.

[14] G. A. Geist and V. S. Sunderam, "The Evolution of the PVM Concurrent Computing System", *Proceedings – 26th IEEE Compcon Symposium*, pp. 471-478, San Francisco, February 1993.

[15] J. Leon, et. al., "Fail Safe PVM: A Portable Package for Distributed Programming with Transparent Recovery", School of Computer Science Technical Report, Carnegie-Mellon University, CMU-CS-93-124, February 1993.

[16] J. Dongarra, et. al., *Abstracts: PVM User's Group Meeting*, University of Tennessee, Knoxville, May 1993.

[17] C. Hartley and V. S. Sunderam, "Concurrent Programming with Shared Objects in Networked Environments", *Proceedings – 7th Intl. Parallel Processing Symposium*, pp. 471-478, Los Angeles, April 1993.

[18] V. S. Sunderam, "DCL: Protocols and Primitives for Distributed Concurrent Computing", *Proceedings – International Conference on Computing and Information*, (Eds. D. Krumme et. al.), Sudbury, Ontario, May 1993.

[19] M. T. Heath and J. A. Etheridge, "Visualizing the Performance of Parallel Programs", IEEE Software, 8(5), pp. 29-39, September 1991.