Experience with Parallel Computing on the AN2 Network

Daniel J. Scales*
Computer Systems Laboratory, Stanford University

Michael Burrows and Chandramohan A. Thekkath
DEC Systems Research Center

Abstract

Technology trends make it attractive to use workstations connected by a local area network as a multicomputing platform for parallel applications. Achieving acceptable application performance in such a workstation cluster using commodity components requires good support from system software and network hardware. This paper describes our experience with parallel programming in a workstation cluster and the implications for operating systems and network adaptor design. Our cluster consists entirely of commercially available hardware: 24 Alpha workstations connected by a 155 Mbits/s AN2 ATM network. We present results from running several parallel applications on this cluster. Our cluster demonstrates both excellent low-level communication performance and good overall application performance that compares favorably with dedicated multicomputers like the IBM SP2.

1. Introduction

In this paper, we describe our experience with parallel computation on a set of high-performance Alpha workstations connected by an AN2 ATM network [2, 22]. Advances in processor and network technology make it increasingly desirable to use such workstation clusters as a loosely coupled multicomputer.

Both medium- and coarse-grained parallel scientific applications can potentially benefit from the characteristics of a workstation cluster. However, building a cluster capable of running such programs efficiently can be difficult, because workstation operating systems, network adaptors, and local-area networks are not usually designed for such a workload. Typically, operating systems and networks are designed to work well in a distributed system where services are provided by servers that communicate with their clients using mechanisms like TCP/IP or remote procedure call (RPC). In contrast, parallel applications typically favor simpler models of computation that involve partitioning data among processors and providing more direct access to data. To support such applications efficiently on our system, we exploit innovative features in each major component of our cluster: the AN2 switch, the network adaptor, and the host operating system. We describe these features in subsequent sections.

The runtime environment we support in our cluster is PVM (Parallel Virtual Machine) [21], a message-passing standard that is widely used in scientific parallel programs. Rewriting large scientific applications is often a non-trivial undertaking that can prevent the potential of workstation clusters from being fully exploited. By supporting a runtime system that is already in widespread use, we were able to run several realistic applications and understand the impact of our hardware and software in the light of this experience. An alternative we considered was to use more specialized communication primitives, such as Remote Memory [20], Active Messages [27], etc. While offering good performance, this alternative has the disadvantage that we would have to rewrite many applications to use these primitives.

1.1. Related work

The idea of running parallel programs on workstation clusters has existed for many years. A few of the early examples are Spector's work on the Alto [20], the Multisatellite star work in the V system [11], and the Nectar project at CMU [3]. There have also been several software systems that support a shared memory or a shared object abstraction on workstation clusters for executing parallel programs. Some examples are IVY [16], Amber [10], Munin [9], Orca [4], and Treadmarks [15]. Current research in using workstation clusters to run parallel applications includes work at Princeton [6], University of Wisconsin [13], University of Washington [24], Berkeley [1], and Cornell [26]. In the past ten years, there has also been con-

*Author's current address: Digital Equipment Corporation Western Research Laboratory

siderable effort in designing hardware and software interfaces to minimize the cost of network access [7, 12, 14, 19, 23].

Our primary goal was to build a working system of reasonable size using high-performance workstations and to gain some experience with running realistic parallel programs. We did not expect our system to give the best possible performance but hoped to achieve good cost/performance tradeoffs relative to commercially available message passing machines. In the rest of the paper we report on the current status of the system that we have built and our experience using it.

2. The environment

Our environment consists of eight 225-MHz DEC 3000 Model 700 machines and sixteen 233-MHz DEC AlphaStation 400 4/233 machines. The eight DEC 3000/700 machines are connected by a single AN2 ATM switch. The remaining 16 AlphaStations are office workstations belonging to our colleagues and are located throughout our building. These machines are interconnected through another switch.

The workstations run the Digital UNIX 3.2 operating system with local modifications to the ATM network adaptor driver. In AN2, data is carried over SONET fiber-optic links using 53-byte cells. Hosts on the network that wish to communicate with each other establish connections called *virtual circuits* (VCs), as specified by the ATM protocols.

2.1. The network switch

AN2 switches implement a credit-based flow control mechanism. A downstream switch sends back credits to the upstream switch for each cell it forwards on a particular VC. The upstream switch will not forward cells for a VC unless it has received credits for cells forwarded previously. This credit-based scheme has the advantage that cells will not be dropped in the network due to congestion, which is the main source of data loss in modern fiber-optic LANs with low-error links. For the moment, until experience indicates otherwise, we therefore feel justified in treating data loss within our workstation cluster as an extremely rare occurrence, and regard it as a catastrophic event. As described in Section 3, we exploit this flow control to eliminate unnecessary overheads in the communication protocol.

2.2. The network adaptor

Depending on the host system bus, the Alpha machines we use are connected to AN2 using either the OTTO TURBOchannel adaptor or the OPPO PCI adaptor. (Though there are some implementation differences in the two adaptors, for the purposes of this paper, the two can be treated as functionally identical.) The network adaptor has some features that facilitate an efficient implementation of PVM. In particular, it can demultiplex incoming data efficiently based on the virtual circuit of the incoming cell and perform scatter DMA to an arbitrary region of host memory. As described in Section 3.1, the message semantics of PVM are well suited to exploit this facility of the adaptor.

The host transmits and receives *packets* on the cell-based ATM network. These packets are suitably formatted, checksummed, fragmented, and reassembled by the adaptor according to standard ATM Adaptation Layer (AAL5) specifications. The adaptor and host communicate via several data structures that are shared between them. The three key data structures that concern us are the *receive ring*, the *transmit descriptors*, and the *report ring*. The first two data structures are maintained inside the adaptor, while the report ring is in host memory.

The receive ring is a circular list of *packet descriptors*, each of which consists of a list of *fragment descriptors*. Each fragment descriptor describes the memory layout of a host buffer that can hold a fragment of an incoming packet. These fragment descriptors can be pre-initialized by the host to point to pinned receive buffers for that particular VC. Every virtual circuit has a receive ring associated with it; when the adaptor receives a packet on a connection, it uses the virtual circuit identifier in the cells of the incoming packet to locate the appropriate ring of receive packet descriptors. The adaptor uses the next free packet descriptor in the ring to initiate DMA from the network to the receive buffer. Our workstations have caches that are kept coherent in the presence of DMA.

The behavior of the transmit descriptors is similar. When the host has a packet to send, it initializes a descriptor with the host memory address and length information for each of the packet fragments. Then, the host writes an adaptor register, causing the adaptor to access the descriptors and perform DMA directly between host memory and the network.

When the adaptor has transmitted or received a packet, it writes a status report into a ring maintained in host memory and interrupts the host. Host software can then examine the report and do the necessary bookkeeping, such as freeing up the packet memory on transmit or notifying the receiver process about packet arrival. There is a single report ring containing reports for all VCs on a host.

One technique that would minimize the cost of using the adaptor is for the operating system to map the adaptor into each application's address space. The application could then program the adaptor to transmit and receive data directly from its address space. This technique is not practical because our adaptor hardware does not support protected mapping of the device into multiple address spaces. In the next section, we describe the changes we put into the operating system to share the adaptor efficiently among multiple ap-

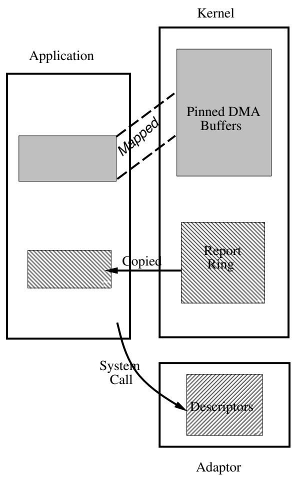


Figure 1. Layout of Buffers, Descriptors, and Report Rings

plications. With a small amount of software we were able to support direct data transfer between a user address space and the network without compromising safety or adding to the complexity of the adaptor.

2.3. The host operating system

In the stock Digital UNIX operating system, network data is transferred to and from kernel DMA buffers. To access the data, programs must call into the kernel, which then copies data between user-level and DMA buffers. In addition, programs make system calls to check for incoming messages. To improve performance, we modified the operating system to eliminate extra data copies and the high overhead of message notifications.

The primary changes we made to the operating system were in the device driver for the adaptor and are described below. Recall from our earlier discussion that the adaptor and the host communicate using DMA buffers, a report ring, and descriptors. Figure 1 shows how the modified operating system organizes these in the kernel, user address spaces, and in the adaptor.

A portion of kernel memory is allocated and kept pinned for DMA transfers to and from the network; separate regions of this buffer memory are mapped into the address spaces of each application, allowing it complete access to the memory. There is no security breach in allowing an application direct access to its portion of DMA memory because the driver never needs to interpret the data in this memory during sends or receives—all the necessary information is contained in the descriptors.

When the adaptor completes DMA, it writes a report into the report ring and interrupts the processor, which then executes the device driver code. The adaptor implements a single report ring that is securely shared among multiple applications as follows. The driver partitions VCs into two groups, one for non-PVM uses such as the TCP/IP protocol stack and another for PVM. When the driver executes on packet arrival, it does one of two things depending on the VC. If the report is for a PVM virtual circuit, the sole action of the driver is to copy the report to a memory region that is mapped into the receiver's address space. As the adaptor has already performed the DMA to the receiver's address space using the appropriate receive descriptors, the driver need not copy the data in the packet. Since reports are only sixteen bytes long, this mechanism has low overhead and does not significantly impact the performance seen by the PVM application. A report for a non-PVM virtual circuit invokes higher layers of the software protocol stack.

Descriptor memory is common to all virtual circuits and thus to potentially many address spaces. Allowing untrusted user code (for example, the PVM library) unrestricted access to the shared descriptor region would be a security loophole. User code must therefore access this region through the kernel. To this end we added a few system calls to the operating system. These calls control (1) the allocation and initialization of descriptors and DMA regions, (2) the initiation of outbound transfers, and (3) the replenishment of receive DMA buffers.

The first set of calls, usually used once at initialization, causes the adaptor driver to allocate descriptor memory and map DMA buffers into user space. The descriptor memory is organized into an appropriate number of transmit and receive packet descriptors, as determined by the application. After initialization, applications do not have direct access to the descriptors, but can refer to a descriptor as an offset from an address that only the kernel can access.

The second call is used each time a packet is to be sent out. The kernel supports a specialized, low-overhead system call to initiate transmission. An application uses this call to specify the transmit descriptor and the region of DMA it wants to use. The driver verifies the arguments in the system call, sets up the descriptor, and prods the adaptor to initiate a data transfer.

The final call is invoked when an application is finished with a DMA receive buffer. Recall that host DMA buffers are used up on each receive, and must be replenished. This call returns a DMA buffer back to the network adaptor for use in receiving new packets. End-to-end flow control ensures that packets do not arrive when there are no receive buffers available.

To summarize, the operating system supports the following functions.

- It maps portions of the kernel's pinned DMA buffers into user space. The adaptor does DMA to and from these buffers to transfer data between the network and the user address space.

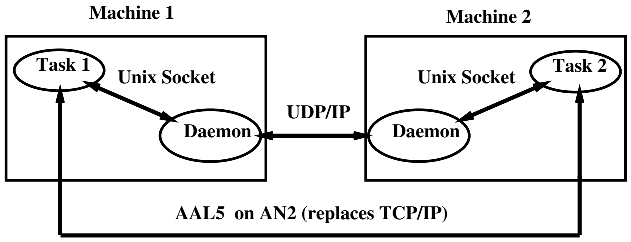


Figure 2. Communication Structure of PVM Applications

- It sets up a copy of the report ring in user space, and copies reports from the adaptor's report ring.
- It reserves VCs for applications, initializes descriptors, and allows applications restricted access to these descriptors through fast system calls.

3. Improving the performance of the PVM library

To support PVM efficiently on our cluster, we made changes to the implementation of the PVM library routines (version 3.3.6), in addition to changes to the operating system. We did not change the programmer-visible interface to PVM library routines or the PVM daemon.

Figure 3 shows the canonical structure of two PVM applications communicating using our PVM library. In our current version, we did not give high priority to speeding up the path between the PVM daemons. Our primary goal was to replace the connection between the application processes so that they use the facilities of AN2 directly, instead of TCP/IP over AN2. In a local area workstation cluster, we believe the error guarantees provided by AN2 enable us to avoid using heavyweight TCP connections.

3.1. Eliminating data copies

A common problem in implementing communication libraries is that there is unnecessary data copying on sends and receives. Often, the interface to the library and the interface to the network implemented by the operating system are mismatched, making it particularly challenging to achieve zero copies [23].

In contrast, our experience with the PVM library and the adaptor has been very different. One of the features of the programmer-visible PVM interface is that the message buffers used by the programmer are never explicitly referred to by memory addresses. This is different from traditional interfaces (e.g., a Unix socket receive), where the programmer explicitly specifies the buffer.

The PVM programmer allocates, manipulates, and deallocates a buffer using an integer identifier. The extra level of indirection provided by this identifier has the benefit that the library has complete freedom in choosing the actual memory addresses to store the data. For instance, when the adaptor performs DMA into a particular receive memory buffer, the library does not have to copy it into a programmer-specified buffer. Instead, the programmer is given a new buffer identifier that stands for the DMA buffer in the library. Of course, a particular application may need to copy data out of a PVM buffer to another location, but that copying occurs only for applications that require it. The PVM library provides unmarshaling routines that the application can use for reading typed data from a PVM buffer and possibly copying the data to another location. The PVM interface meshes nicely with the scatter-gather DMA facilities of the adaptor and enables us to eliminate all unnecessary data copying at the library level.

3.2. Reducing system call overheads

Security needs dictate that the PVM library has to cross into kernel space to access the descriptors on sends and receives. Thus, system call overhead is a potential source of increased latency. We minimized the latency of the send by building a specialized kernel entry point. The special entry point avoids much of the overhead of a standard system call by saving as little state as possible and dispatching quickly to the device driver routine.

Recall that on receive, the adaptor transfers the data to user space via DMA and the driver writes a copy of the report in user memory. Thus, an application does not require a system call to read the data; it can simply check for a new report in its ring at user level. In addition to this low overhead mechanism, the driver also allows applications to block on the adaptor waiting for data using the standard *select* system call. When the adaptor next receives data, the application is unblocked. In our implementation, the PVM receive routine checks the report ring a few times and then blocks if there is no data. The kernel's time-slicing mechanism prevents a polling PVM application from unfairly monopolizing the processor.

Periodically, an application has to replenish DMA buffers so that the adaptor can transfer incoming data to the user. As described above, we provide a lightweight system call to indicate that a previously used buffer is now free and can be used by the network adaptor to receive new messages. Our PVM library uses this system call when a process explicitly frees a PVM message buffer and when a process receives a new message, which implicitly frees the current PVM receive buffer.

3.3. Message arrival notification

In parallel applications with complex data structures and irregular communication patterns, much of the communication involves request-reply messages for data. Request messages can arrive at a processor at any time and should be serviced as soon as possible in order to keep the latency low. In general, a process can detect asynchronous message arrival by polling or through interrupts. The PVM programmer interface provides functions for message polling, but no mechanism for interrupting the computation on message arrival. With standard PVM, polling with a `pvm_probe` is expensive (977 μs), because each `pvm_probe` requires a *select* system call. This high overhead can impact the parallel performance of applications if they poll frequently. Nonetheless, it is important to poll for messages reasonably often so that request messages are served without a long delay.

As described above, the adaptor allows applications to check for incoming messages without a system call, and we reimplemented the `pvm_probe` routine to use this facility instead. However, it must still use a non-blocking version of the *select* system call to check for messages from the PVM daemon, which communicates with the application via a standard TCP socket. In order to keep polling overheads very low, we check for messages from the adaptor on every poll, but not for messages from the daemon. Instead, we check for messages from the daemon only once every 200 `pvm_probe` operations. This number can be configured by an application via an environment variable. This approach amortizes the cost of a *select* and works well since the majority of the messages for a PVM process arrive via the adaptor. Because of this amortization, our modified routine has an *average* poll time of 1.3 μs and applications can therefore poll frequently to check for incoming messages.

3.4. Performance measurements of the new library

Figure 3 summarizes the performance of exchanging messages between two PVM processes using `pvm_send` and `pvm_recv` on 225 DEC 3000/700 processors, rated at 163 SPECint92 and 230 SPECfp92. The machines are connected by a 155 Mbits/s switch-based AN2 network. Excluding ATM header overheads, the maximum achievable per-link bandwidth is about 16.8 Mbytes/s (about 134 Mbits/s). Elapsed times are measured using a cycle counter in the Alpha processor. In all cases, the switch is unloaded and the hosts are running multiuser but are idle except for the test program and standard daemons.

Figure 3(a) shows the bandwidth achieved in sending messages via our modified PVM. For comparison, we show the performance of the original unmodified PVM3 running in an identical environment. The throughput of the modified library is 15.8 Mbytes/s, about 94% of what the link is capable of supporting. In contrast, the original implementation gives us about 10 Mbytes/s or about 60% of link bandwidth. As packet size increases, reduced copying overheads in our implementation contribute to better performance.

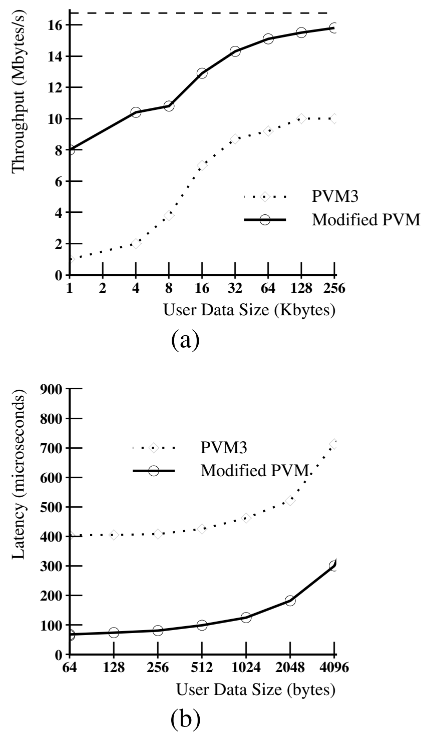


Figure 3. Throughput and One-Way Latency of PVM on DEC 3000/700 Processors

Figure 3(b) compares the latency of our implementation with the original on the same hardware. The latency shown is for half a roundtrip using `pvm_send` and `pvm_recv` to exchange a message. For a null message, this is about 65 μs for our implementation. The costs of initializing a message buffer and packing and unpacking data into it are not included in this measurement. (If they were, it would add about 15 μs to the one-way latency of a null message.)

Of the total one-way latency for a null packet, the kernel and user-level software account for about 17 μs. The adaptor hardware and the switch account for the remaining 48 μs. The sending and receiving adaptors each add 17.5 μs and the switch adds 13 μs. Some of the hardware latency can be attributed to the interface chips that the adaptor and switch use for SONET framing. These chips introduce a delay of 5 cell times on a one-way trip, which translates to about 14 μs with 155 Mbits/s links.

The message latency between PVM applications is dependent on processor speed. In the 16-node experiments described below, we use the 233-MHz AlphaStations rated at 157 SPECint92 and 184 SPECfp92. (That is, the DEC 3000/700 machines and the AlphaStations have comparable integer performance, but the former has significantly better floating-point performance.) The one-way latency in this case is around 88 μs for a null message.

4. Applications

In this section we describe our experience with running four parallel applications: Water simulation, Grobner basis computation, Barnes-Hut simulation, and block Cholesky factorization. The Water simulation is a coarse-grained application, while the others are more fine-grained and irregular. All except the Grobner basis computation are part of the SPLASH-2 application suite [28]. All applications are written in a shared-memory style using the SAM runtime system [18].

SAM is a portable shared object runtime system for distributed memory machines, including workstation clusters. SAM supports a global name space in which each processor can access any shared object regardless of its current location, and SAM caches the shared objects in software in order to exploit the data locality of parallel applications. In addition, SAM incorporates techniques for (1) prefetching data, (2) eagerly pushing data to remote destinations, and (3) making chaotic accesses to data to improve application performance. SAM relies on PVM for its communication needs on workstation clusters.

We report on the performance of these applications in two configurations: an eight-node cluster of DEC 3000/700 machines connected by a single AN2 switch and a cluster of sixteen DEC AlphaStations connected by another identical AN2 switch. (We have also run the finer-grained application on a mixed cluster of 24 nodes containing both types of machines and describe their performance briefly.) For comparison, we show the performance of our applications on an IBM SP2, a dedicated multicomputer.

The SP2 has a custom interconnect and 66-MHz POWER2 processors rated at 122 SPECint92 and 260 SPECfp92. On the SP2, the SAM library uses MPL, the machine's native message-passing library, rather than PVM. The measured bandwidth on the SP2 interconnect is about 30 Mbytes/s and the one-way latency for a null message is about 45 μs. Thus, the SP2 has about twice the bandwidth of our implementation and about 70% of our best-case latency.

We report application speedups with respect to the efficient sequential version of each application on the same kind of node. Table 1 shows the absolute execution times of the sequential version of the applications running on each kind of node. The numbers in parenthesis in the columns for the AlphaStation and the SP2 are the ratios of the execution times of each application on these nodes compared to the corresponding times on a DEC 3000/700.

Application	Execution Time in Seconds		
	3000	AlphaStation	SP2
Water	118.0	144.5 (1.2)	162.3 (1.4)
Grobner	8.3	9.9 (1.2)	8.6 (1.0)
Barnes-Hut	86.6	100.4 (1.2)	129.5 (1.5)
Cholesky (sparse)	2.4	2.9 (1.2)	1.8 (0.8)
Cholesky (dense)	4.5	6.7 (1.5)	3.4 (0.8)

Table 1. Absolute and Relative Sequential Execution Times

4.1. Performance summary

Subsequent sections analyze application performance in detail. To summarize the results, for the coarse-grained application, i.e., Water, performance is independent of the PVM implementation; both implementations yield good performance that compares well with the SP2.

Performance of the finer-grained applications is very dependent on latency and throughput. In all cases, application execution times with our new PVM library are significantly smaller compared to the standard version. Also, on the eight-node cluster, the execution time for Water and Barnes-Hut with our library is faster than on an eight-node SP2. On the same cluster, our execution time for Grobner basis is about 1.5 times that of the SP2. For block Cholesky factorization, our time is about 1.2 times what the SP2 achieves. On the slower 16-node cluster, our run times for the finer-grained applications range from about 1.3 to 1.8 times the execution times on a 16-node SP2.

4.2. Coarse-grained application

Water is an example of a coarse-grained application. It is a molecular dynamics application that evaluates the forces and potentials of a system of water molecules. The computation is performed over a number of time steps. Each time step requires solving Newtonian equations of motion for water molecules in a cubic region.

Figure 4 shows the speedup curves for simulations of 1728 molecules. For comparison, we show the speedups for the original and modified PVM implementation as well as for the SP2. One characteristic of Water is that the main processor must send out and collect a large amount of data from the other processors on each time step. The available network bandwidth therefore affects the performance of the application, and Water has slightly better performance using our version of PVM. The available bandwidth on the SP2 is about twice that on the workstation cluster and we observe better speedup on the SP2, especially for larger numbers of processors. However, because of faster uniprocessor performance, the execution time on our eight-node cluster is less than the time on an eight-node SP2.

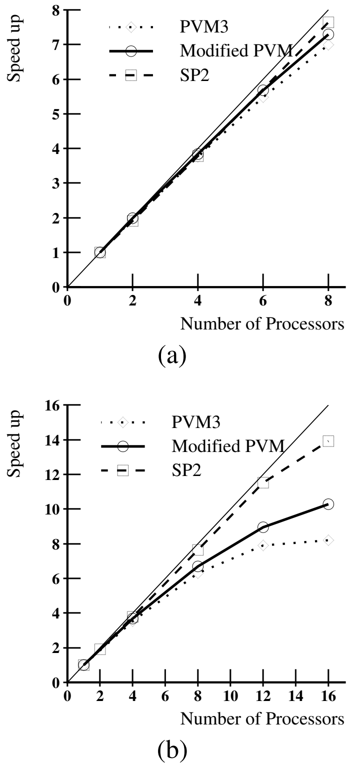


Figure 4. Speedup of Water for 1728 molecules on (a) DEC 3000/700 and (b) AlphaStation 400

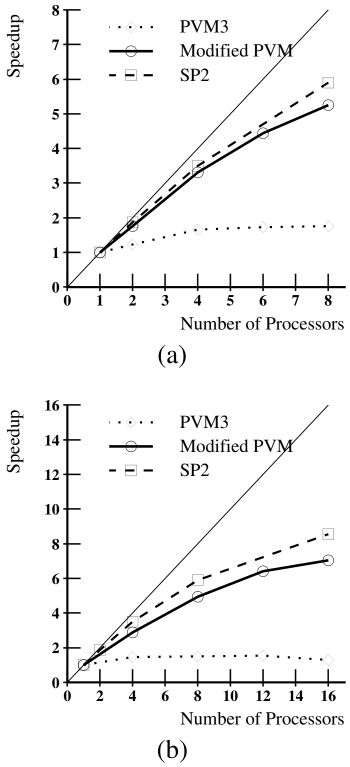


Figure 5. Speedup of Grobner Basis on Input Set Lazard on (a) DEC 3000/700 and (b) AlphaStation 400

4.3. Medium-grained applications

The Grobner basis computation, Barnes-Hut simulation, and block Cholesky factorization are three medium-grained applications we ran on our cluster. Of these, the first two applications have similar performance characteristics.

The Grobner basis application computes the Grobner basis [8] of a set of polynomials. Our parallel version is derived from a sequential version due to Vidal [25]. The application proceeds by repeatedly processing pairs of polynomials and potentially adding new polynomials to the basis set, until all possible pairs have been tested. Processors frequently access the polynomials already in the basis as they test new polynomials, so the SAM caching functionality is crucial for good performance. Latency is the most important factor in determining performance for this application. Low latency decreases the processor contention time for the common basis set and allows all processors to see modifications to the basis as soon as possible.

Figure 5 shows the speedup curves for the Grobner basis application on input set Lazard. Since our implementation of PVM has low latency, we observe good speedup and good overall performance up to eight processors. However, eventually the performance tails off because increasing the number of processors increases contention for the basis, causing processors to do significantly more work than the original serial code. The SP2 has even lower latency than our modified PVM and achieves slightly better speedups than the workstation clusters. We have run this application on 24 nodes containing a mix of the two machine types. On our 24-machine cluster, the absolute execution time is about 1.9 times that on a 24-node SP2.

The Barnes-Hut application [5] implements a fast algorithm for simulating the evolution of bodies in a 3-d gravitational field. At each time step, the algorithm calculates the forces between the bodies and uses the force on each body to determine its new position and velocity. The algorithm builds a data structure called an *oct-tree* such that the force calculation can be done in time $O(n \log n)$ rather than $O(n^2)$. The force computation is partitioned in such a way that there is extensive locality of access to the oct-tree by each processor. Each processor accesses remote nodes frequently, but SAM automatically exploits the locality by caching recently used nodes. To increase the communication granularity somewhat, the application has been modified to group together nearby nodes of the tree into a single object.

Figure 6 shows the speedup curves of the Barnes-Hut

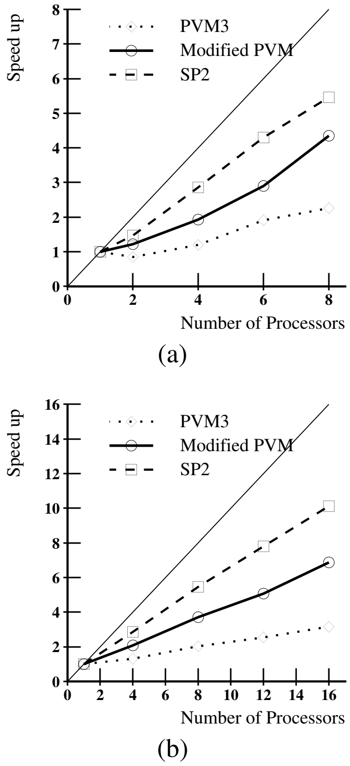


Figure 6. Speedup of Barnes-Hut on 25000 Bodies on (a) DEC 3000/700 and (b) AlphaStation 400

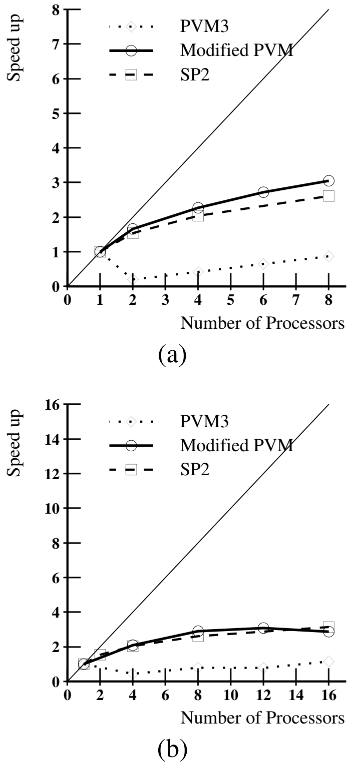


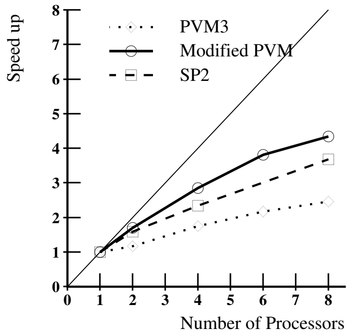
Figure 7. Speedup of Block Cholesky Factorization on BCSSTK15 on (a) DEC 3000/700 and (b) AlphaStation 400

application when simulating 25,000 bodies. Because of faster uniprocessor performance, the overall running time on the eight DEC 3000/700 machines is better than on an eight-node SP2. Even though the Barnes-Hut application has good locality, performance is greatly influenced by the request-reply latency when retrieving nodes of the tree that are not currently cached. Given the latency characteristics of the implementations, application speedup of our PVM library is, predictably, intermediate between the SP2 and the standard library. On our 24-node mixed cluster, for our implementation of PVM, we observe a speedup of 6.7 relative to the faster DEC 3000/700 workstations; the speedup for the original PVM is 3.1. The absolute execution time on our 24-machine cluster is 1.4 times that on a 24-node SP2. We note that the overall performance of Barnes-Hut on the Alpha workstations for the modified PVM is relatively impressive for such a dynamic and fine-grained application running on workstation clusters.

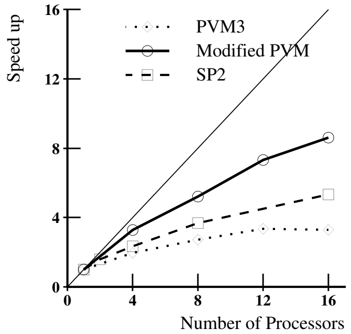
The last application we report on is the block Cholesky application [17], which performs a Cholesky factorization of a symmetric matrix in parallel. It statically decomposes the matrix into blocks and assigns each processor the work of updating a fixed set of blocks. Updates typically involve a destination block that depends on two source blocks. The application achieves speedup by updating blocks in parallel while maintaining the dependency requirements. Our implementation takes advantage of a facility in SAM to push data to nodes that might need them. A completed block of the matrix is pushed to the set of nodes that will need to access it. This optimization is important, because it can reduce the time spent by nodes on the critical path waiting for a particular source block to arrive.

Figures 7 and 8 show the speedup curves of the application on the Harwell Boeing sparse matrix BCSSTK15 and a dense matrix D1000, respectively. In all cases, speedup is shown relative to an efficient, sequential, column-oriented factorization algorithm. We were unable to use the "push" optimization described above on the SP2 for eight-processor runs, because of some apparent deadlock problems with the SP2 message-passing library, but we were able to use it for the 2-, 4-, and 16-processor runs. The sparse matrix has dimension 3948×3948 matrix with 117,816 double precision non-zero elements and the dense matrix is 1000×1000.

We note that the speedups on the sparse matrix are low and taper off at about 12 processors. The performance on 16 processors is slightly lower than on 12 processors. We believe this is because the increased bandwidth requirement creates multiple processor bottlenecks, each of which affects



(a)



(b)

Figure 8. Speedup of Block Cholesky Factorization on D1000 on (a) DEC 3000/700 and (b) AlphaStation 400

multiple nodes. In general, the low speedups are due in large part to the limited parallelism available in factoring a sparse matrix. Speedups are also affected by the available network bandwidth. High network bandwidth reduces the amount of time spent sending blocks of the matrix and reduces the critical path time that limits overall performance. Our implementation of PVM has much better bandwidth than the original and hence, significantly better performance. The SP2 has about twice our bandwidth and thus, better overall performance. The ratio of communication to computation is lower for the dense matrix, so the unmodified PVM does comparatively better on the dense matrix.

In factoring the sparse matrix, with its limited parallelism, the peak performance we have observed is 214 Mflops on the eight DEC 3000/700 workstations. For the dense matrix, we observe 324 Mflops on the same cluster, and on a 24-node cluster with a mix of the two machines, we observe 429 Mflops. For comparison, an eight-node SP2 achieves 252 Mflops on the sparse matrix and 364 Mflops on the dense matrix, and a 24-node SP2 has a peak performance of 587 Mflops in factoring the dense matrix.

5. Conclusions

We have described the hardware and operating system support that allows us to achieve good communication performance on a workstation cluster. To summarize, credit-based flow control in the AN2 reduces the probability of cells lost due to congestion, our AN2 adaptor transfers messages via DMA directly from the network into user-level receive buffers, and our operating system mechanisms provide light-weight interfaces to send and receive messages.

We were able to use these features to tune an implementation of PVM, a popular message-passing library for parallel programs. In the past, high-performance implementations of message-passing libraries on networks like Ethernet or FDDI have been difficult to achieve. This is often an artifact of the constraints imposed by the networks rather than a function of the library per se. For example, Ethernets and FDDI rings provide no reliability guarantees and scientific applications typically do not have mechanisms to deal with packet loss. As a result, message-passing implementations have usually relied on heavyweight transports like TCP/IP for reliability. Also, many older network adaptors and operating system interfaces cannot eliminate overheads caused by unnecessary data copying. Our experience has been different. Given the small set of capabilities described above, we were able to get good performance from commodity hardware.

Dedicated multicomputers, such as the SP2, are expected to scale to hundreds of processors. We do not yet have the experience to know whether our workstation cluster will scale beyond 24–32 nodes. However, for medium- and coarse-grained applications, of the type described in the paper, our cluster has good performance that is within a factor of two of the SP2, even in the worst case. High-performance, general-purpose computers and LANs, like the Alpha workstation and ATM products, have a per-node price advantage of more than a factor of two over special-purpose machines like the SP2. Thus, our experience suggests that, in terms of cost-performance, our cluster compares quite favorably with such dedicated multicomputers for medium- and coarse-grained applications.

Acknowledgments

We would like to thank Hal Murray and Tom Rodeheffer for helping us understand the AN2 hardware. Herb Yeary and Hal Murray helped us with configuring the building-wide AN2 network. Kai Li, Hal Murray, Roger Needham, Mike Schroeder, and Chuck Thacker commented on earlier drafts of this paper. We would also like to acknowledge the anonymous referees for their many useful comments. Cynthia Hibbard provided invaluable editing help. We would also like to thank our colleagues at SRC for donating their machines for our experiments. Our use of the SP2 at the Maui High Performance Computing Center for the performance comparison was supported by the Phillips Laboratory, Air Force Material Command, USAF, under cooperative agreement number F29601-93-2-0001.

References

[1] T. E. Anderson, D. E. Culler, and D. A. Patterson. A case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, Feb. 1995.

[2] T. E. Anderson, S. S. Owicki, J. B. Saxe, and C. P. Thacker. High-speed switch scheduling for local-area networks. *ACM Trans. Comput. Syst.*, 11(4):319–352, Nov. 1993.

[3] E. A. Arnould, F. J. Bitz, E. C. Cooper, H. Kung, R. D. Sansom, and P. A. Steenkiste. The design of Nectar: A network backplane for heterogeneous multicomputers. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 205–216, April 1989.

[4] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3), Mar. 1992.

[5] J. E. Barnes and P. Hut. A hierarchical O(NlogN) force-calculation algorithm. *Nature*, 324(6096):446–449, Dec. 1986.

[6] M. A. Blumrich, K. Li, R. Alpert, Z. Dubnicki, and E. W. Felten. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 142–153, May 1994.

[7] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15:29–36, Feb. 1995.

[8] B. Buchberger. Grobner basis: An algorithmic method in polynomial ideal theory. In N. K. Bose, editor, *Multidimensional Systems Theory*, chapter 6, pages 184–232. D. Reidel Publishing Company, 1985.

[9] J. B. Carter, J. K. Bennet, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, Oct. 1991.

[10] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 147–158, Dec. 1989.

[11] D. R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, Mar. 1988.

[12] P. Druschel, L. L. Peterson, and B. S. Davie. Experiences with a high-speed network adaptor: A software perspective. In *Proceedings of the 1994 SIGCOMM Symposium on Communications Architectures, Protocols and Applications*, pages 2–13, Aug. 1994.

[13] M. D. Hill, J. R. Larus, and D. A. Wood. Tempest: A substrate for portable parallel programs. In *COMPCON '95*, Mar. 1995.

[14] H. Kanakia and D. R. Cheriton. The VMP network adapter board (NAB): High-performance network communications for multiprocessors. In *Proceedings of the 1988 SIGCOMM Symposium on Communications Architectures and Protocols*, pages 175–187, Aug. 1988.

[15] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–132, January 1994.

[16] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, Nov. 1989.

[17] E. Rothberg and A. Gupta. An efficient block-oriented approach to parallel sparse Cholesky factorization. In *Proceedings of Supercomputing '93*, pages 503–512, Nov. 1993.

[18] D. J. Scales and M. S. Lam. The design and evaluation of a shared object system for distributed memory machines. In *Proceedings of the First Symposium on Operating System Design and Implementation*, pages 101–114, Nov. 1994.

[19] M. D. Schroeder, A. D. Birrell, M. Burrows, H. Murray, R. M. Needham, T. L. Rodeheffer, E. H. Satterthwaite, and C. P. Thacker. Autonet: A high-speed, self-configuring local area network using point-to-point links. *IEEE Journal on Selected Areas in Communications*, 9(8):1318–1335, Oct. 1991.

[20] A. Z. Spector. Performing remote operations efficiently on a local computer network. *Commun. ACM*, 25(4):246–260, April 1982.

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

[22] C. P. Thacker and M. D. Schroeder. AN2 switch overview. In preparation.

[23] C. A. Thekkath and H. M. Levy. Limits to low-latency communication on high-speed networks. *ACM Trans. Comput. Syst.*, 11(2):179–203, May 1993.

[24] C. A. Thekkath, H. M. Levy, and E. D. Lazowska. Efficient support for multicomputing on ATM networks. Technical Report 93-04-03, Department of Computer Science and Engineering, University of Washington, Apr. 1993.

[25] J.-P. Vidal. The computation of Grobner bases on a shared-memory multiprocessor. Technical Report CM-CS-90-163, Carnegie Mellon University, 1990.

[26] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 40–53, Dec. 1995.

[27] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266, May 1992.

[28] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.