Parallel Computing on the Berkeley NOW

David E. Culler, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Brent Chun,
Steven Lumetta, Alan Mainwaring, Richard Martin, Chad Yoshikawa, Frederick Wong

Computer Science Division
University of California, Berkeley

Abstract: The UC Berkeley Network of Workstations (NOW) project demonstrates a new approach to large-scale system design enabled by technology advances that provide inexpensive, low latency, high bandwidth, scalable interconnection networks. This paper provides an overview of the hardware and software architecture of NOW and reports on the performance obtained at each layer of the system: Active Messages, MPI message passing, and benchmark parallel applications.

1 Introduction

In the early 1990's it was often said that the "Killer Micro" had attacked the supercomputer market, much as it had the minicomputer and mainframe markets earlier. This attack came in the form of massively parallel processors (MPPs) which repackaged the single-chip microprocessor, cache, DRAM, and system chip-set of workstations and PCs in a dense configuration to construct very large parallel computing systems. However, another technological revolution was brewing in these MPP systems – the single-chip switch – which enabled building inexpensive, low latency, high bandwidth, scalable interconnection networks. As with other important technologies, this "killer switch" has taken on a role far beyond its initial conception. Emerging from the esoteric confines of MPP backplanes, it has become available in a form that is readily deployed with commodity workstations and PCs. This switch is the basis for *system area networks*, which have performance and scalability of the MPP interconnects and the flexibility of a local area network, but operate on a somewhat restricted physical scale.

The Berkeley NOW project seeks to demonstrate that it is viable to build large parallel computing systems that are fast, inexpensive, and highly available, by simply snapping these switches together with the latest commodity components. Such cost-effective, incrementally scalable systems provide a basis for traditional parallel computing, but also for novel applications, such as internet services[Brew96].

This paper provides an overview of the Berkeley NOW as a parallel computing system. Section 2 gives a description of the NOW hardware configuration and its layered software architecture. In the following sections, the layers are described from the bottom-up. Section 3 describes the Active Message layer and compares its performance to what has been achieved on MPPs. Section 4 shows the performance achieved through MPI, built on top of Active Messages. Section 5 illustrates the application performance of NOW using the NAS Parallel Benchmarks in MPI. Section 6 provides a more detailed discussion of the world's leading disk-to-disk sort, which brings out a very important property of this class of system: the ability to concurrently perform I/O to disks on every node.

2 Berkeley NOW System

The hardware configuration of the Berkeley NOW system consists of one hundred and five Sun Ultra 170 workstations, connected by a large Myricom network[Bode95], and packaged into 19-inch racks. Each workstation contains a 167 MHz Ultra1 microprocessor with 512 KB level-2 cache, 128 MB of memory, two 2.3 GB disks, ethernet, and a Myricom "Lanai" network interface card (NIC) on the SBus. The NIC has a 37.5 MHz embedded processor and three DMA engines, which compete for bandwidth to 256 KB of embedded SRAM. The node architecture is shown in Figure 1.

The network uses multiple stages of Myricom switches, each with eight 160 MB/s bidirectional ports, in a variant of a fat-tree topology.

2.1 Packaging

We encountered a number of interesting engineering issues in assembling a cluster of this size that are not so apparent in smaller clusters, such as our earlier 32-node prototype. This rack-and-stack style of packaging is extremely scalable, both in the number of nodes and the ability to upgrade nodes over time. However, structured cable management is critical. In tightly packaged systems the interconnect is hidden in the center of the

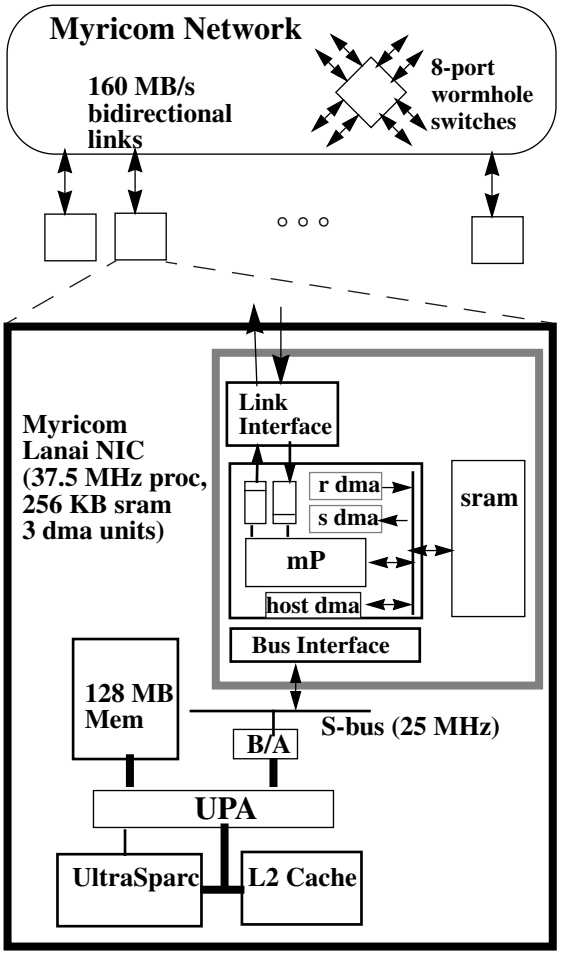

Figure 1. NOW Node Configuration

machine. When multiple systems are placed in a machine room, all the interconnect is hidden under the floor in an indecipherable mess. However, in clusters, the interconnect is a clearly exposed part of the design. (a bit like the service conduits in deconstructionist buildings). Having the interconnect exposed is valuable for working on the system, but it must stay orderly and well structured, or it becomes both unsightly and difficult to manage.

The Berkeley NOW has four distinct interconnection networks. First, the Myrinet provides high-speed communication within the cluster. We discuss this in detail below. Second, switched-Ethernet into an ATM backbone provides scalable external access to the cluster. The need for an external network that scales with the size of the cluster was not apparent when we began the project, but the traffic between the cluster and other servers, especially file servers, is an important design consideration. Third, a terminal concentrator provides direct console access to all the nodes via the serial port. This is needed only in situations when the node cannot be rebooted through the network, or during system development and debugging. Fourth, conventional AC lines provide a power distribution network. As clusters transition to the commercial mainstream, one engineering element will be to consolidate these layers of interconnect into a clean modular design. Figure 2 shows a picture of the NOW system.

Figure 2. NOW System

2.2 Network topology

The Myrinet switches that form the high-speed interconnect use source routing and can be configured in arbitrary topologies. The NOW automatic mapping software can handle arbitrary interconnect[Mai*97]; however, we wire the machine as a variant of a Fat-tree to create a system with more uniform bandwidth between nodes, thereby minimizing the impact of process placement. The topology is constrained by the use of 8-port (bidirectional) switches and wiring density concerns. Initially we planned to run cables from all the nodes to a central rack of switches; however, the cable cross-sectional area near the switches became unmanageable as a result of bulky, heavily-shielded copper network cables. Using fiber-optic cables that are now available, the cable density may be reduced enough to centrally locate the switches.

In building an indirect network out of fixed-degree switches, the number of upward links depends on the number of downward links. We elected to attach five hosts to each first level switch, which eliminates 40% of the cable mass. As shown in Figure 3, groups of seven of these switches are treated as a 35-node subcluster with the 21 upward links spread over four level-two switches. Three of these subclusters are wired together to comprise the NOW. We have found that as a rule of thumb, adding 10% extra nodes and extra ports greatly simplifies system administration, allowing for node failures, software or hardware upgrades, and system expansion.

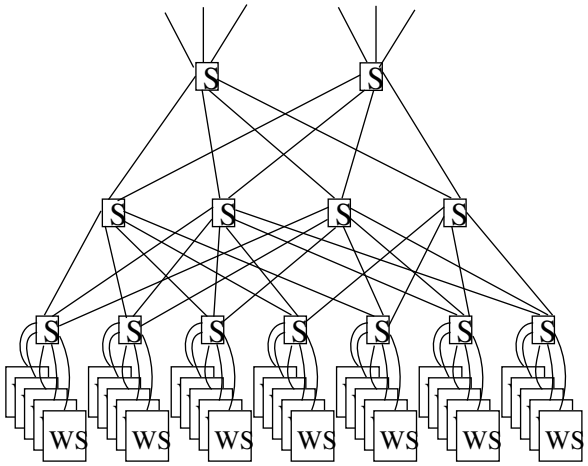

Figure 3. NOW Myrinet Network Topology

2.3 Software Architecture

The system software for NOW employs a layered architecture, as illustrated in Figure 4. Each node runs a complete, independent Solaris Unix with the associated process management, memory management, file system, thread support, scheduler, and device drivers. We extend Solaris at each of these interfaces to support global operations over the NOW.

Process Management: A global OS layer, called GLUnix, provides NOW-wide process management as a layer on top of Solaris (via sockets, daemons, and signals). Using either a global shell, `glush`, or the `glurun` command, sequential processes can be started anywhere on the NOW or parallel processes can be started on multiple nodes. Local `pids` are elevated to a global `pids`, and the familiar process control operations, such as ctrl-C or ctrl-Z, work on global processes. The Unix process information and control utilities, such as `ps` and `kill`, are globalized as well.

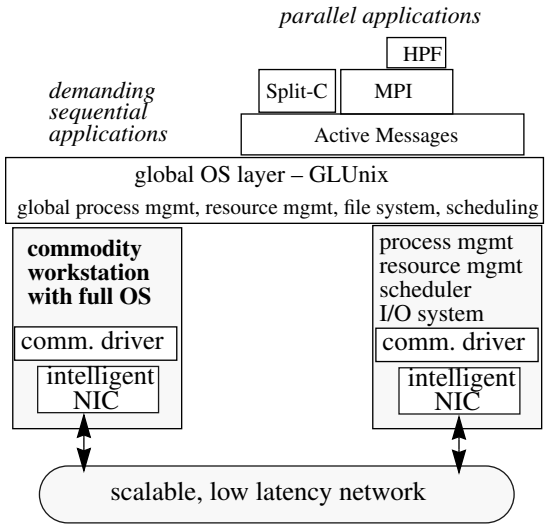

Figure 4. NOW software architecture

File System: A prototype file system, xFS, extends Solaris at the vnode interface to provide a global, high performance file system[And*95b]. Files are striped over nodes in a RAID-like fashion so that each node can read file data at the bandwidth of its interface into the network. The aggregate bandwidth available to nodes is that of all the disks. xFS uses a log-structured approach, much like Zebra[HaOu95], to minimize the cost of parity calculations. A single node accumulates enough of the log so that it can write a block to each disk in a stripe group. Before writing the blocks, it calculates a parity block locally and then writes it along with the data blocks.

An update-based file cache-coherence strategy is used, and the caches are managed cooperatively to increase the population of blocks covered by the collection of nodal caches. If a block about to be discarded is the last copy in the system, then it is cast off to a random remote node. Nodes take mercy on this block until it has aged to the point where it appears pointless to keep it in memory. This policy has the attractive property that actively used nodes behave like traditional clients while idle nodes behave like servers, so the cooperative file cache adapts dynamically to system usage.

Virtual Memory: Two prototype global virtual memory systems have been developed to allow sequential processes to page to the memory of remote idle nodes, since communication within the NOW has higher bandwidth, and much lower latency than access to local disks. One of these uses a custom Solaris segment driver to implement an external user-level pager which

exchanges pages with remote page daemons. The other provides similar operation on specially mapped regions using only signals.

3 Active Messages

Active Messages are the basic communication primitives in NOW. This work continues our investigation of implementation trade-offs for fast communication layers[vE92*,Gol*96,Kri*96] and on NOW we have sought to generalize the approach and take full advantage of the complete OS on every node. The segment driver and device driver interface is used to provide applications with direct, protected user-level access to the network. Active Messages map to simple operations on queues and buffers that are shared between the user process and the communication firmware, which is executed on a dedicated processor embedded in the network interface card.

We have built two Active Message layers. The first, Generic Active Messages (gam) is oriented toward the traditional single-parallel-program-at-a-time style of parallel machines, and provides exactly the same API across a wide range of platforms[Cul*95].This serves as a valuable basis for comparison.

The newer AM layer[Main95], AM-II, provides a much more general purpose communication environment, which allows many simultaneous parallel programs, as well as client/server and system use. It is closely integrated with POSIX threads. The AM implementation is extremely versatile. It provides error detection and retry a the NIC-to-NIC level and allows the network to be reconfigured in a running system. A privileged mapper daemon explores the physical interconnection, derives deadlock-free routes, and distributes routes periodically[Mai*97]. AM-II provides a clean return-to-sender error model to support highly available applications.

The Active Messages communication model is essentially a simplified remote procedure call that can be implemented efficiently on a wide range of hardware. Three classes of messages are supported. Short messages pass eight 32-bit arguments to a handler on a destination node, which executes with the message data as arguments. Medium messages treat one of the arguments as a pointer to a 128 byte to 8 KB data buffer and invoke the handler with a pointer to a temporary data buffer at the destination. Bulk messages perform a memory-to-memory copy before invoking the handler. A request handler issues replies to the source node.

We have developed a microbenchmarking tool to characterize empirically the performance of Active Messages in terms of the LogP model[Cul*93, Cul*95]. Figure 5 compares the gam short message LogP parameters on NOW with the best implementations on a range of parallel machines. The bars on the left show the one-way message time broken down into three components: send overhead (o_s), receive overhead (o_r), and the remaining latency (L). The bars on the right shows the time per message (g = *1/MessageRate)* for a sequence of messages. NOW obtains competitive or superior communication performance to the more tightly integrated, albeit older, designs.

The overhead on NOW is dominated by the time to write and read data across the I/O bus. The Paragon has a dedicated message processor and network interface on the memory bus; however, there is considerable overhead in the processor-to-processor transfer due to the cache coherence protocol and the latency is large because the message processors must write the data to the NI and read it from the NI. The actual time on the wire is quite small. The Meiko has a dedicated message processor on the memory bus with a direct connection to the network, but the overhead is dominated by the exchange instruction that queues a message descriptor for the message processor and the latency is dominated by the slow message processor accessing the data from host memory. Medium and bulk messages achieve 38 MB/s on NOW, limited primarily by the SBus.

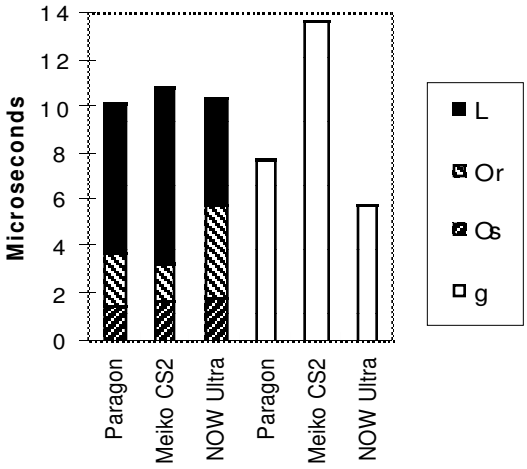

Figure 5. Active Messages LogP Performance

Traditional communication APIs and programming models are built upon the Active Message layer. We have built a version of the MPI message passing stan-

dard for parallel programs in this fashion, as well as a version of the Berkeley Sockets API, called Fast Sockets[Rod*97]. A shared address space parallel C, called Split-C[Cul*93], compiles directly to Active Messages, whereas HPF[PGI] compiles down to the MPI layer.

4 MPI

Our implementation of MPI is based on the MPICH reference implementation, but realizes the abstract device interface (ADI) through Active Message operations. This approach achieves good performance and yet is portable across Active Message platforms. The MPI communicator and related information occupy a full short message. Thus, a zero-byte control message is implemented as a single small-message request-response, with the handler performing the match operation against a receive table. The one-way time for an echo test is 15 µs. MPI messages of less than 8 KB use an adaptive protocol implemented with medium Active Messages. Each node maintains a temporary input buffer for each sender and senders keep track of whether their buffers are available on the destination nodes. If the buffer is available, the send issues the data without handshaking. Buffer availability is conveyed back to the source through the response, if the match succeeds, or via a request issued by the later matching receive. Large messages perform a handshake to do the tag match and convey the destination address to the source. A bulk operation moves the message data directly into the user buffer.

Figure 6 shows the bandwidth obtained as a function of message size using Dongarra's echo test on NOW and on recent MPP platforms[DoDu95]. The NOW version has lower start-up cost than the other distributed memory platforms and has intermediate peak bandwidth. The T3D/pvm version does well for small messages, but has trouble with cache effects. Newer MPI implementations on the T3D should perform better than the T3D/pvm in the figure, but data is not available in the Dongarra report.

5 NAS Parallel Benchmarks

An application-level comparison of NOW with recent parallel machines on traditional scientific codes can be obtained with the NAS MPI-based parallel benchmarks in the NPB2 suite[NPB]. We report briefly on two applications. The LU benchmark solves a finite difference discretization of the 3-D compressible Navier-Stokes equations. A 2-D partitioning of the 3-D data grid onto a power-of-two number of processors is obtained by halv-

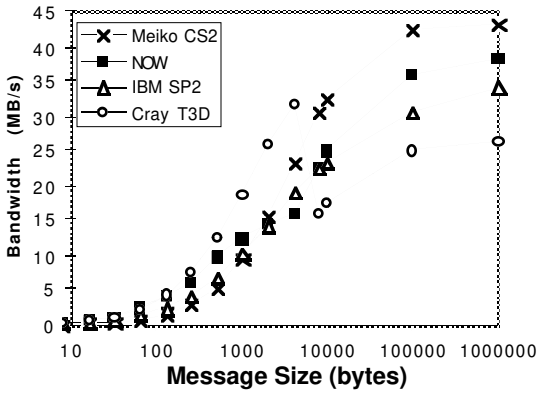

Figure 6. MPI bandwidth

ing the grid repeatedly in the first two dimensions, alternating between *x* and *y*, resulting in vertical pencil-like grid partitions. The ordering of point based operations constituting the SSOR procedure proceeds on diagonals which progressively sweep from one corner on a given z plane to the opposite corner of the same *z* plane, thereupon proceeding to the next *z* plane. This constitutes a diagonal pipelining method and is called a "wavefront" method by its authors [Bar*93]. The software pipeline spends relatively little time filling and emptying and is perfectly load-balanced. Communication of partition boundary data occurs after completion of computation on all diagonals that contact an adjacent partition.

The BT algorithm solves three sets of uncoupled systems of equations, first in the *x*, then in the *y*, and finally in the *z* direction. These systems are block tridiagonal with 5x5 blocks and are solved using a multi-partition scheme[Bru88]. The multi-partition approach provides good load-balance and uses coarse-grained communication. Each processor is responsible for several disjoint sub-blocks of points ("cells") in the grid. The cells are arranged such that for each direction in the line-solve phase, the cells belonging to a certain processor are evenly distributed along the direction of solution. This allows each processor to perform useful work throughout a line-solve, instead of being forced to wait for the partial solution to a line from another processor before beginning work. Additionally, the information from a cell is not sent to the next processor until all sections of linear equation systems handled in this cell have been solved. Therefore the granularity of communications is kept large and fewer messages are sent. The BT code requires a square number of processors.

Figure 7 shows the speedups obtained on sparse LU with the Class A input (200 iterations on a 64x64x64 grid) for the IBM SP-2 (wide nodes), Cray T3D, and NOW. The speedup is normalized to the performance on four processors, shown in Table 1, because this is the smallest number of nodes for which the problem can be run on the T3D. NOW achieves the scalability of the T3D with much high per-node performance. It scales better than the SP-2, although it has only two-thirds the node performance. These results are consistent with the ratio of the processor performance to the performance of small message transfers.

TABLE 1. NPB baseline MFLOPS

	T3D	SP-2	NOW
LU @ 4 proc	74	236	152
BT @ 25 proc	378	1375	714

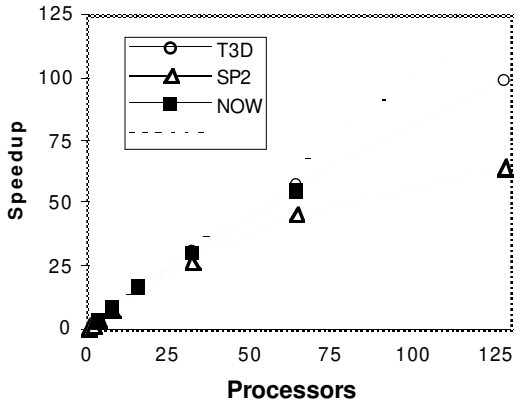

Figure 7. Speedup on NPB Sparse LU

Figure 8 shows the scalability of the IBM SP2, Cray T3D, and NOW on the Class A problem of the BT benchmark. Here the speedup is normalized to the performance on 25 processors, the smallest T3D configuration for which performance data is available. The scalability comparison is similar to that with LU, with the SP-2 lagging somewhat more, but having higher per node performance.

6 Disk-to-Disk Sort

The real capability of NOW with a complete operating system per node and I/O capability spread across the nodes is best illustrated through I/O-intensive applications, such as commercial workloads and scientific

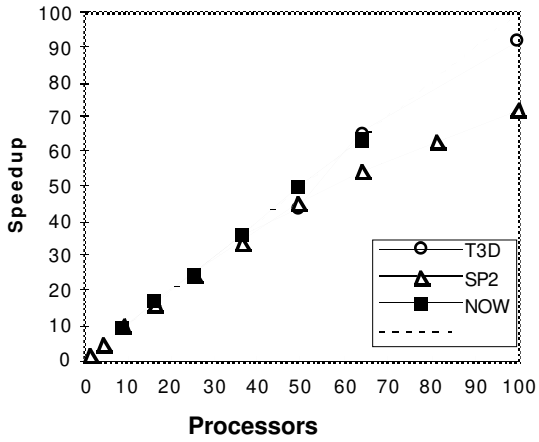

Figure 8. Speedup on NPB BT

application on very large data sets. To evaluate this aspect of the system, we have developed a high-performance disk-to-disk sort. Sorting is an important application to many fields, especially the database community; it stresses both the I/O subsystem and operating system, and has a set of well-known benchmarks allowing comparison with other systems. We describe our experience with the Datamation benchmark, proposed in 1985 by a group of database experts[Anon85].

By utilizing the full operating system capability, local disks, and fast communication, NOW-sort has broken all previous records on the Datamation benchmark, which measures the elapsed time to sort one million 100-byte records with 10-byte keys from disk to disk, and Minute-sort, which measures the amount of record data sorted in under a minute[Nyb*94].

We briefly describe our parallel algorithm for NOW-Sort, which is given in complete detail in [Arp*97]. One of the goals of NOW-Sort is to dynamically adapt to a variety of cluster configurations: namely, differing numbers of disks, amounts of memory, and number of workstations. A primary variation is the size of the data set as compared to the size of available memory. A one-pass version is used when all records fit into main memory; otherwise a two-pass version is used. Due to space limitations, we only describe the one-pass implementation here.

6.1 Algorithm Description

In the Datamation sort benchmark the keys are from a uniform distribution and begin evenly distributed across all workstations. At the highest level, the single-pass

parallel sort of N records on P processors is a generalized bucket sort and contains four steps:

1. **Read:** Each processor reads $\frac{N}{P}$ 100-byte records from its local disks into memory.
2. **Communicate:** Keys are examined and the records are sent to the appropriate destination processor. The destination processor copies a portion of the key to a local bucket, keeping a pointer to the full key and record; thus the keys are partially sorted according to their most-significant bits.
3. **Sort:** Each processor sorts its keys in memory using a partial-radix sort on each bucket.
4. **Write:** Each processor gathers and writes its sorted records to local disks.

At the end of the algorithm, the data is sorted across the disks of the workstations, with the lowest-valued keys on processor 0 and the highest-valued keys on processor $P-1$. The number of records per node will only be approximately equal, and depends upon the actual distribution of key values.

6.2 Local disk performance

A key advantage of a NOW is that the performance of each node can be studied and optimized in isolation before considering the interactions between nodes. For NOW-Sort, we needed to understand how best to utilize the disks, the memory, and the operating system substrate of each node.

To fully utilize the aggregate bandwidth of multiple disks per machine, we implemented a user-level library for file striping on top of each local Solaris file system. We have two disk configurations to consider: two 5400 rpm disks on a fast-narrow SCSI bus and an additional two 7200 rpm disks on a fast-wide SCSI. Table 2 shows the performance of the striped file system for several configurations.

In the first two rows, we see that the two 5400 rpm disks saturate the fast-narrow SCSI bus, which has a peak bandwidth of 10 MB/s. We measure 8.3 MB/s from two disks capable of a total of 11 MB/s. The full NOW cluster has two disks per node, providing a potential file I/O bandwidth of 830 MB/s on 100 nodes.

A subset of the nodes have additional external disks. The second two rows indicate that the (external) fast-wide SCSI bus adequately handles the two faster disks. Finally, the last rows shows we achieve 20.5 MB/s, or 96% of the peak aggregate bandwidth from the four disks. To harness this bandwidth in the sort, we need to adjust the blocking factor on the two kinds of disks to balance the transfer times.

TABLE 2. Bandwidths of disk configurations

Seagate Disks	SCSI Bus	Read (MB/s)	Write (MB/s)
1 5400 rpm Hawk	narrow	5.5	5.2
2 5400 rpm Hawk		8.3	8.0
1 7200 rpm Barracuda	wide	6.5	6.2
2 7200 rpm Barracuda		13.0	12.1
2 of each	both	20.5	19.1
2 of each (peak)		21.3	20.1

6.3 OS Interfaces for Buffer Management

Given a near peak bandwidth local file system, the sorting performance of each node depends critically on how effectively memory is used. With a general purpose OS, there is no simple way for the application to determine how much memory is actually available to it. Depending upon the system interface used to read data from disk, the application may or may not be able to effectively control its memory usage.

We compare two approaches for reading records from disk: `read` and `mmap` with `madvise`. For demonstration purposes, we use a simple implementation of the sorting algorithm using one UltraSparc with one disk and 64 MB of DRAM. It quicksorts all of the keys in memory. The upper graph of Figure 9 shows that when the application uses the `read` call to read records into memory from disk, the total execution time increases severely when more than 20 MB of records are sorted, even though 64 MB of physical memory are in the machine. The operating system uses roughly 20 MB and the file system performs its own buffering, which effectively doubles the application footprint. This performance degradation occurs because the system starts paging out the sorting data.

To avoid double-buffering while leveraging the convenience of the file system, we use memory mapped files by opening the file, calling `mmap` to bind the file into a memory segment of the address space, and accessing the memory region as desired. The auxiliary system call, `madvise`, informs the operation system of the intended access pattern. For example, one call to `madvise` notifies the kernel that region will be accessed sequentially, thus allowing the OS to fetch ahead of the current page

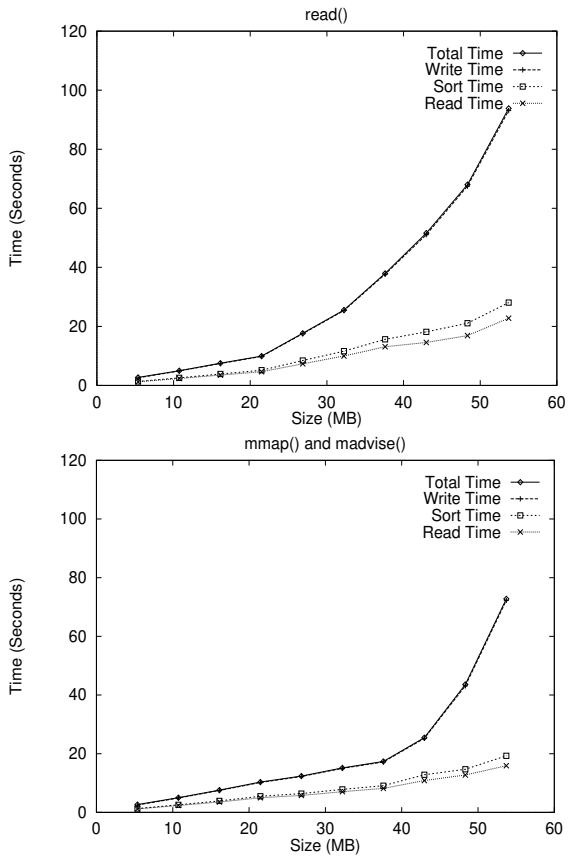

Figure 9. Sensitivity to OS Interface

and to throw away pages that have already been accessed. The lower graph of Figure 9 shows that with `mmap` and `madvise` the sorting program has linear performance up to roughly 40 MB, when it has used all the memory that the OS makes available. For larger data sets, the two pass algorithm is used, which places greater demand on the file system and requires multiple threads to maintain the disk bandwidth[Arp*97].

6.4 Using Active Message communication

The optimizations for striping data across local disks and using operating system interfaces for memory management apply when running on a single-node or multiple nodes. The impact of parallelization is isolated to the communication phase.

After each node has memory-mapped its local input files, it calculates the processor which should contain this key in the final sorted order. Using the assumption that the keys are from a uniform distribution, we determine the destination processor with a simple bucket function (i.e., the top $\log P$ bits of each key) and copy the record from the input file to a 4 KB *send-buffer* allocated for each destination processor. When a send-buffer is filled, it is sent to its destination processor.

Upon arrival of a message, an Active Message handler is executed. The handler moves the full record into main memory and copies a portion of the key into one of $B = 2^b$ buckets based on the high-order b bits of the key. The number of buckets is calculated at run-time such that the average number of keys per bucket fits into the second-level cache.

The read and communication phases are easily overlapped due to the interfaces provided by `mmap` and Active Messages. Copying keys into send-buffers is completely hidden under the disk transfer time. Obtaining this same performance with the `read` system call would require more programming complexity; because the cost of issuing each `read` is high, threads must be used prefetch data in large chunks.

Measurements on a cluster with two disks per workstation show that the communication time is mostly hidden by the read time. However, with four disks per workstation, very little communication is overlapped with reading, and the algorithm is actually slower than with only two disks. This penalty occurs because the UltraSPARC I/O bus, the SBus, saturates long before its theoretical peak of 80 MB/s. Since almost all records are sent to another processor, the SBus must transfer three times the I/O rate: once for reading, once for sending, and once for receiving.

6.5 Sorting and Writing

The sort and write phase on each node are straightforward. After synchronizing across processors to ensure that all records have been sent and received, each node performs a partial-radix sort on the set of keys in each bucket, very similar to the approach described in [Agar96]. The partial-radix sort orders keys using the top 22-bits after the stripping off the $\log P + b$ bits used to determine the destination processor and bucket. At this point, with high-probability, most keys are correctly sorted, and a simple bubble-sort cleans-up the misordered keys. A pointer is kept to the full 100-byte record so that only keys and pointers are swapped. The sorted records are then gathered and written to local disk using the `write` interface.

6.6 Performance Measurements

Our performance on the Datamation benchmark is shown in Figure 10 for two NOW configurations. For

each configuration, the lower curve is the sort time and the upper curve gives the additional GLUnix start-up time. The previous record-holder on this benchmark, indicated by the horizontal line, was a 12 processor SGI PowerChallenge with 96 disks and 2.25 GB of main memory[Swe96]. In NOW-Sort, each processor sorts an equal portion of the one million records, so as more processors are added, each node sorts fewer records. With the resulting small problem sizes, remote process start-up is a significant portion of our total sorting time. In fact, on 32 processors, the total time is almost equally divided between process start-up and our application. However, this is probably more a function of the lack of maturity of our own cluster OS, than the fundamental costs of a distributed operating system, and is currently the focus of optimization.

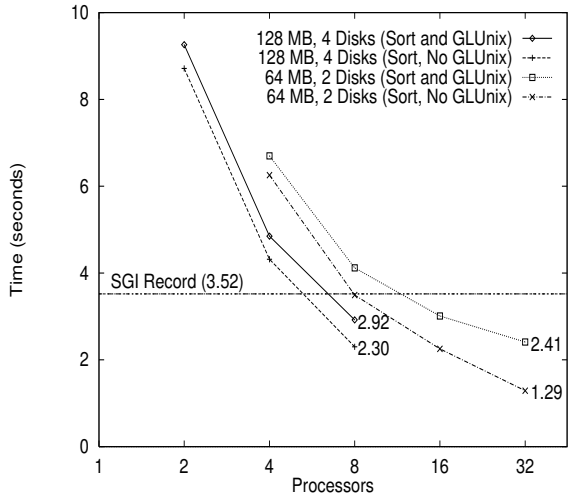

Figure 10. Datamation Sort Time

Interestingly, process start-up can also be time-consuming on SMPs: both [Nyb*94] and [Swe*96] indicate that clearing the address-space is a significant, if not dominant, cost for one-pass external sorts, most likely because the SMP OS does not parallelize the process. On NOWs, this aspect of process creation is not a problem, since each local address space is initialized in parallel.

NOW-Sort is almost perfectly scalable as the number of processors is increased and the number of records per processors is held constant. Figure 11 shows the time for the parallel one-pass sort, for the two-disk, 64 MB clusters and the four-disk 128 MB cluster. The slight increase in time with more processors is mostly due to the increase in overhead of GLUnix and initializing the Active Message layer.

Recognizing that the Datamation benchmark has become more a test of startup and shutdown than I/O performance, the authors of AlphaSort introduced MinuteSort in 1994[Nyb*94]. The key and record specifications are identical to that of Datamation, but the performance metric is now the amount of data that can be sorted in one minute of elapsed time.

NOW-Sort currently holds the record for this benchmark as well, sorting 8.41 GB in one-pass on 95 machines, 190 disks, and 12.16 GB of memory. For details on how we solved this problem when there was insufficient memory to hold all keys in main memory, see [Arp*97].

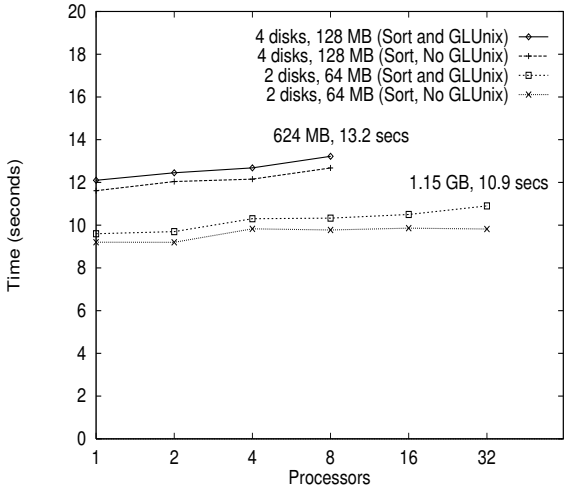

Figure 11. Scalability of NOW-Sort

7 Conclusions and Future Directions

Leveraging the advent of the "killer network switch", the Berkeley NOW project has shown that large-scale clustered systems can be built and utilized for a range of parallel applications. Assembling the more than 100 Sun UltraSparc workstations in our current configuration raised a number of unique engineering issues, such as cable management, boot and diagnostic support, and system administration, but reasonable solutions exist for each.

The infrastructure necessary for running parallel applications has been thoroughly exercised in our system. We have shown that not only can we obtain the low-latency, high-bandwidth communication traditionally seen in only tightly-integrated MPP systems, but also that this communication performance directly corresponds to speedups for scientific parallel applications. Through disk-to-disk sorting, we have demonstrated many of the strengths of the NOW approach: the presence of a com-

plete OS and file-system on each node for memory-management and high-performance I/O, and the ability to isolate performance issues.

The experiments that we have described in this paper have focused on single applications running in isolation. A number of most interesting current challenges involve using the Berkeley NOW as a truly time-shared resource: particularly, integrating communication and scheduling. The second-generation of Active Messages, AM-II, was specifically designed for such an environment: by virtualizing communication endpoints, multiple processes on a single node fairly and efficiently share network resources. Preliminary measurements indicate that while the round-trip times of our prototype with a single communicating process are somewhat higher than the more restricted gam implementation (45 µs versus 20 µs), performance degrades gracefully under heavy load and the system is resilient of network failures.

Some of the future work with AM-II involves determining which communication endpoints should be resident when the physical resources on the network interface are over-committed. A step in this direction is to understand the communication working sets of various applications. Investigating how information from the process scheduler can be incorporated into AM-II is also of interest.

Coordinating the scheduling of communicating processes across the nodes in the cluster is currently an unsolved problem. Traditional time-shared MPPs used explicit coscheduling, or gang scheduling, to achieve this coordination; however, explicit coscheduling has a number of disadvantages that are accentuated in the NOW environment: it behaves poorly with mixed workloads of interactive and parallel jobs and significantly increases the cost of context-switches. Our simulation results indicate a non-intrusive, implicit approach works better: communication and synchronization events occurring naturally within the parallel application transmit sufficient information to implicitly coschedule applications[Dus*96]. By dynamically adapting the amount of time a parallel process waits for communication events to complete, bulk-synchronous applications achieve performance within 10% of an ideal implementation of coscheduling.

Integrating Active Messages with implicit coscheduling in the cluster is likely to raise many more issues. One question that remains is how well implicitly-scheduled parallel and interactive jobs can share the same workstation. We are also beginning to investigate how to fairly allocate the pool of resources among competing users [ArCu97].

8 Acknowledgments

We would like to thank the rest of the members of the NOW project for their critical contributions to this work, including Tom Anderson, Dave Patterson, Eric Anderson, Satoshi Asami, Douglas Ghormley, Kim Keeton, Cedric Krumbein, Jeanna Neefe Matthews, Steve Rodrigues, Nisha Talagala, Amin Vahdat, Randy Wang, Eric Fraser, and Ken Lutz. We also thank Bill Saphir and Horst Simon of NERSC for their help with the NPB.

This work was supported in part by the Defense Advanced Research Projects Agency (N00600-93-C-2481, F30602-95-C-0014), the National Science Foundation (CDA 9401156), Sun Microsystems, California MICRO, Hewlett Packard, Intel, Microsoft, and Mitsubishi.

9 References

[Agar96] R Agarwal, “A Super Scalar Sort Algorithm for RISC Processors”, *Proceedings of the 1996 ACM SIGMOD Conference*, June 1996, p. 240-246.

[And*95a] T. Anderson, D. Culler, D. Patterson, and the NOW team, “A Case for NOW (Networks of Workstations),” *IEEE Micro*, Feb. 1995.

[And*95b] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, R. Wang, “Serverless Network File Systems”, *15th Symposium on Operating Systems Principles,* 1995.

[Anon85] Anonymous et. al, “A Measure of Transaction Processing Power”, *Datamation*, 31(7):112-118, 1985, Also in *Readings in Database Systems*, M.H. Stonebraker ed., Morgan Kaufmann, San Mateo, 1989.

[Arp*97] A. Arpaci-Dusseau, R. Arpaci-Dusseau, D. Culler, J. Hellerstein, and D. Patterson, “High-Performance Sorting on Networks of Workstations,” *Proceedings of the 1997 ACM SIGMOD Conference*, (to appear).

[ArCu97] A. Arpaci-Dusseau, D. Culler, “Extending Proportional-Share Scheduling to a Network of Workstations”, *International Conference on Parallel and Distributed Processing Techniques and Applications*, 1997 (to appear).

[Bar*93] E. Barszcz, R. Fatoohi, V. Venkatakrishnan, and S. Weeratunga, ``Solution of Regular,

Sparse Triangular Linear Systems on Vector and Distributed-Memory Multiprocessors", Technical Report NAS RNR-93-007, NASA Ames Research Center, Moffett Field, CA, 94035-1000, April 1993.

[Bod*95] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Sietz, J. Seizovic, and W. Su. "Myrinet -- A Gigabit-per-Second Local-Area Network," *IEEE Micro*, Feb. 1995, p. 29-36.

[Brew96] E. Brewer, "The Inktomi Experience," Invited Talk at the *Second Workshop on Networks of Workstations,* Cambridge, MA, Oct. 1996.

[Bru88] Bruno, J; Cappello, P.R.: Implementing the Beam and Warming Method on the Hypercube. Proceedings of 3rd Conference on Hypercube Concurrent Computers and Applications, Pasadena, CA, Jan 19-20, 1988.

[Cul*93a] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, T. von Eicken, "LogP: Towards a Realistic Model of Parallel Computation," Proc. *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* San Diego, CA May1993.

[Cul*93b] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken and K. Yelick, "Parallel Programming in Split-C," Proceedings of *Supercomputing 93*, November 1993.

[Cul*96] D. Culler, L.T. Liu, R. P. Martin, C. O. Yoshikawa, "Assessing Fast Network Interfaces", *IEEE Micro (Special Issues on Hot Interconnects)* vol 16, no. 1, Feb 1996.

[DoDu95] J. Dongarra, and T. Dunnigan, "Message Passing Performance of Various Computers", Univ. Tenn Tech Report CS-95-299, May 1995.

[Dus*96] A. Dusseau, R. Arpaci, D. Culler, "Effective Distributed Scheduling of Parallel Workloads", *SIGMETRICS '96 Conference on Measurement and Modeling*,1996.

[Gol*96] S. Goldstein, K. Schauser, and D. Culler, "Lazy Threads: Implementing a Fast Parallel Call", *Journal of Parallel and Distributed Computing,* 1996.

[HaOu95] J. Hartman and J. Ousterhout, "The Zebra Striped Network File System", *ACM Transactions on Computer Systems,* 13(3) Aug. 1995, 279-310.

[Kri*96] A. Krishnamurthy, R. Wang, C. Scheiman, K. Schauser, D. Culler and K. Yelick, "Evaluation of Architectural Support for Global Address-based Communication in Large-scale Parallel Machines", *Seventh International Symposium on Architectural Support for Programming Languages and Operating Systems,* Oct, 1996.

[Liu*96] L. Liu, D. Culler, and C. Yoshikawa, "Benchmarking Message Passing Performance using MPI", *1996 International Conference on Parallel Processing*, Aug. 1996.

[Mai95] A. Mainwaring, et al, "Active Messages Specification", Available at http://now.cs.berkeley.edu/.

[Mai*97] A. Mainwaring, B. Chun, S. Schleimer, D. Wilkerson, "System Area Network Mapping", *Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, June, 1997.

[Mar*97] R. Martin, A. Vahdat, D. Culler, and T. Anderson, "Effect of Communication Latency, Overhead, and Bandwidth on a Cluster Architecture," *Int'l Symp. on Computer Architecture, June 1997* (to appear)

[MPI93] MPI Forum, "MPI: A Message Passing Interface", *Supercomputing 93*, 1993.

[NPB] NAS Parallel Benchmarks, `http://science.nas.nasa.gov/Software/NPB/`.

[Nyb*94] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet, "AlphaSort: A RISC Machine Sort}", *Proceedings of 1994 ACM SIGMOD Conference,* May1994.

[PGI] The Portland Group, Inc., `http://www.pgroup.com/lit.papers.html`

[Rod*97] S. Rodrigues, T. Anderson, and D. Culler. "High-Performance Local-Area Communication Using Fast Sockets", *Proceedings of the USENIX 1997 Annual Technical Conference,* Jan. 1997.

[Swe*96] A. Sweeny, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS File System, *Proceedings of the USENIX 1996 Annual Technical Conference*, Jan. 1996.

[vEi*92] T. von Eicken, D. E. Culler, S. Goldstein and K. Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation," Proc. *19th Annual International Symposium on Computer Architecture*, May 1992, pp. 256-67.

[Wood96] P. R. Woodward, "Perspectives on Supercomputing: Three Decades of Change", *IEEE Computer,* Oct. 1996, pp. 99-111.