

# Beowulf Parallel Processing for Dynamic Load-balancing<sup>12</sup>

Bonnie Holte Bennett, Emmett Davis, Timothy Kunau  
Artificial Intelligence/High Performance and Parallel Computing Laboratory  
Graduate Programs in Software, University of St. Thomas,  
2115 Summit Avenue, OSS301  
St. Paul, MN 55105  
651-962-5509  
[kunau@ahc.umn.edu](mailto:kunau@ahc.umn.edu), [EmmettDa@aol.com](mailto:EmmettDa@aol.com), [bhbennett@stthomas.edu](mailto:bhbennett@stthomas.edu)

William Wren  
Honeywell Technology Center  
3660 Technology Drive, Minneapolis, MN 55418  
612-951-7885

*Abstract*—The Beowulf parallel-processing architecture is a public-domain software technology developed by NASA to link commercial off the shelf (COTS) processors yielding super-computer performance (10 Gflops) very economically (under \$150K). It has been used for applications as diverse as scalable hierarchical particle algorithms for galaxy formation and accretion astrophysics, phase transitions in the early universe and their analogues in the laboratory, and data analysis from the spacecraft in the Solar-Terrestrial Probe Line. Beowulf clusters have been ranked among the top 125 world-class supercomputers. Beowulf runs on a variety of platforms (workstations, Win-tel PCs, Macintosh) and a variety of generations of hardware. It has been successfully demonstrated for heterogeneous parallel processing with hundreds of diverse platforms.

This paper describes our implementation of a Beowulf cluster comprising 18 Sun workstations and the Linux operating system. We describe and demonstrate ParaSort, a distributed, parallel, data-allocation-sorting algorithm with automatic distributed load-balancing, and fault-tolerant performance. The load-balancing feature can provide dynamic, on-board, adaptive optimal distribution of processing tasks across a heterogeneous network of computing devices. Furthermore, it can intelligently allocate processing to off-board resources as appropriate and as they become available. We analyze the strengths and weaknesses of Beowulf and describe future extensions of our work.

## TABLE OF CONTENTS

1. INTRODUCTION
2. APPROACH
3. THE BEOWULF ARCHITECTURE
4. PARASORT ON BEOWULF
5. CONCLUSIONS
6. REFERENCES
7. BIOGRAPHIES

## 1. INTRODUCTION

Dynamic adaptability, a keystone feature for applications implemented on modern networks, is a basis for load balancing across complex networks. A system that is dynamically adaptive can reallocate processing load to keep all components working at maximum efficiency. With the increased dependence on distributed and parallel processing to support general as well as safety-critical applications, we must have applications that perform automatic load balancing. Programs must be able to recognize that current resources are no longer available. Schedulers are employed in the presence of faults to manage resources against program needs using dynamic or fixed priority scheduling for timing correctness of critical application tasks. [1], [2], [3], and [4].

### *Information and Data Fusion Applications*

Information and data fusion applications have been a general thrust of research in our lab. The need for faster, cheaper, more efficient processing for fusion has motivated much of our research. Information fusion is the analysis and correlation of incoming data to provide a comprehensive assessment of the arriving reports. Example applications include military intelligence correlation, meteorological data interpretation, and real-time economic news synthesis. Data fusion is the correlation of individual reports from sensor systems. It operates at a lower level than information fusion. Data fusion is often concerned with processing such as common aperture correlation across similar sensors (for example, SAR and IR radar systems).

The first step to information and data fusion applications is often data sorting. Once the data are properly sorted they can be correlated with other data and higher level reports.

---

<sup>1</sup> 0-7803-5846-5/00/\$10.00 © 2000 IEEE

<sup>2</sup> Updated August 31, 1999

Data sorting is important to many algorithms, but it is particularly important for parallel and distributed algorithms requiring search because non-local data are either ignored or delay the system's performance while they are retrieved.

For example, in a complex data fusion system individual data from a variety of reports may be input to the system. These inputs may be reports from surveillance in border-patrol for drug enforcement or they may be locations and flight paths for aircraft in an air traffic control scenario. The goal of the processing may be to correlate related reports where correlation often depends on spatial or temporal proximity. In either the drug enforcement or the air traffic control scenario the reports are spread across a two- or three-, or four-dimensional spatio-temporal domain. These reports can be more efficiently processed if they are sorted by proximity. Finally, if these reports are to be processed by a distributed system, then proximity sorting is essential since different reports may be sent to different processors for correlation. And, the space and time location of the reports is one way to partition the processing across the nodes. Thus, sorting is important in direct and indirect ways to the processing of many algorithms.

#### *Other Previous Work*

Much of the work in sorting for information fusion deals with linear arrays ([6] and [7]), as opposed to higher-dimensional arrays. The general approach is to take linear sort techniques and use either a row major or a snake-like grid overlaid on a regular grid topology of processors [8] and [9]. The snake-like grid is used at times with a shear-sort or shuffle-sorting program where there is first a row operation and then an alternating column operation. So, either the row or the column connections are ignored in each cycle.

## 2. APPROACH

We have taken a novel approach by designing our algorithms for multi-dimensional structures from the ground up. Subsequently, we have refocused on the design of elemental processes such as load balancing and fault-tolerance. Instead of depending on schedulers, we design process algorithms where global processes are completed using only local knowledge and recovery resources. This lessens the need for schedulers and eases their workload. Also since our ParaSort algorithm load balances before it finishes sorting and detects when the system is sorted, it also detects global termination of load balancing, which “from a practical point of view ... is by no means a trivial problem.” [5]

#### *Local Knowledge and Global Processes*

An efficient network sort algorithm is highly desirable, but difficult. The problem is that it requires local operations with global knowledge. So, consider a group of data (for example, that of names in a phone directory) was to be distributed across a number of processors (for example, 26). Then an efficient technique would be for each processor to take a portion of the unsorted data and send each datum to the processor upon which it eventually belongs (A's to processor 1, B's to processor 2, ... Z's to processor 26).

Thus the global knowledge that all names beginning with the same letter belong on a prespecified processor facilitates local operations in sending off each datum. The problem, however, is that this does not adequately balance the load on the system because there may be many A's (Adams, Anderson, Andersen, Allen, etc.) and very few Q's or X's. So the optimal loading (Aaa-Als on processor 1, Alb-Bix on processor 2, ... Win-Zzz on processor 26) cannot be known until all the data is sorted. So global knowledge (the optimal loading) is unavailable to the local operations (where to send each datum) because it is not determined until all the local operations are finished. ParaSort, however, combines load balancing within sorting. Traditionally, techniques such as hashing have been used to overcome the non-uniform distribution of data. However, parallel hash tables require expensive computational maintenance to upgrade each sort cycle, thus making them less efficient than ParaSort, which requires no external tables.

A significant practical feature of ParaSort is that in our experiments it load balances before it finishes sorting. Since ParaSort detects when the system is sorted, it also detects termination of load balancing. Chengzhong Xu and Francis Lau [5] observe:

From a practical point of view, the detection of the global termination is by no means a trivial problem because there is a lack of consistent knowledge in every processor about the whole workload distribution as load balancing progresses.

#### *Research Objective*

The questions we investigated in our research were: “How can heterogeneous network sorting be optimized?” and “What implications and benefits result?” Our goal was to efficiently load balance and sort data over an arbitrary network. We sought to determine a way to move data around the network quickly. So that, using the telephone directory example, names starting with A and B that were initially assigned to processor 26 would quickly move up to processors 1 or 2 or 3 without having to spend a great deal of time “bubbling up” through the bowels of the network interconnections. Thus, our initial sorting goal was to get

data to processors in the neighborhood where they belonged and then to shuffle the local data into the right positions on the right processor all with the correct load balance.

*Implementation Approach: ParaSort*

ParaSort is our load balancing and sorting algorithm. Our initial approach was to use four-connectedness (as an example of N-connectedness) for load balancing and sorting. In traditional linear sorts data is either high or low for the processor it is on, and is sent up or down the sort chain accordingly. Our approach differs in that we defined data to be very high, high, low, or very low. In order to do this we first defined a sort sequence across an array of processors as depicted in Figure 1.

1	2	3	4
8	7	6	5
9	10	11	12
16	15	14	13

**Figure 1.** The sort sequence is overlaid in a snake-like grid across the array of processors. The lowest valued items in the sort will eventually reside on processor 1 and the highest valued items on processor 16. Node 7's four connected trading partners are in bold: 2, 6, 8, and 10. When Node 7 receives its initial state, it sorts and splits the data into four quarters. The lowest quarter goes to Node 2. The next lowest quarter goes to Node 6, the third quarter to Node 8, and the highest quarter goes to node 10. Thus the extremely high and low data are shipped across the coils of the snake network.

Next we defined the four neighbors. This is easily understood by examining Node 7 in the example of sixteen processors shown in Figure 1.

The trading neighbors Node 2 and Node 10 which are not adjacent on the sort sequence (transcoil neighbors) provide a pathway for very low or very high data to pass across the coils of the snake network into another neighborhood of nodes. This provides an express pathway for extremely ill sorted data to move quickly across the network. The concept of four connectedness is easy to understand with an interior node like Node 7, but other remaining nodes in this example are edge nodes, and their implementation differs slightly.

Simply put, we use a torus for full connectivity. So nodes along the "north" edge of the array which have no north neighbors are connected (conceptually) to nodes along the "south" edge and vice versa (trans edge neighbors). Similarly, a node along the "east" edge are given nodes along the "west" edge as east neighbors and so forth. The odd cycle column of Table 1 summarizes all the nodes of a

sixteen-node network. Thus, the use of the torus for four-connectedness provides full connectivity. The result is a modified shear-sort where both row and column

**Table 1.** Trading partner list. Determining which data is kept at a node depends on how that node falls among the sort order of its neighbors. For example, node 1 falls below all of its neighbors and thus receives the lowest quarter.

Node	Odd Cycle	Even Cycle
1	1 2 4 8 16	1 16 4 8 2
2	1 2 3 7 15	1 2 15 7 3
3	2 3 4 6 14	2 3 14 6 4
4	1 3 4 5 13	3 1 4 13 5
5	4 5 6 8 12	4 5 12 8 6
6	3 5 6 7 11	5 3 6 11 7
7	2 6 7 8 10	6 2 7 10 8
8	1 5 7 8 9	7 5 1 8 9
9	8 9 10 12 16	8 9 16 12 10
10	7 9 10 11 15	9 7 10 15 11
11	6 10 11 12 14	10 6 11 14 12
12	5 9 11 12 13	11 9 5 12 13
13	4 12 13 14 16	12 4 13 16 14
14	3 11 13 14 15	13 11 3 14 15
15	2 10 14 15 16	14 10 2 15 16
16	1 9 13 15 16	15 9 13 1 16 2

connections are used with each round of sorting. Furthermore, ill-sorted data is quickly moved across the network via torus connections. The "express pathway" is a conceptual map of the sorting network. Ideally, the operating system supports express pathways, such as in an Intel Paragon system where we first implemented our algorithm. Where this environmental support is missing, the cost of these non-adjacent operations is higher. In those environments where networks have edges, ParaSort has three strategies. The first is to still implement the conceptual torus at the higher transmission cost. The second is to reconfigure itself to the reality of some nodes having only two or three physical neighbors. A third strategy is particularly useful in heterogeneous environments, where we employ a genetic algorithm to determine the optimal network by minimizing transmission costs.

ParaSort's distributed approach can provide an efficient control mechanism for a wide variety of algorithms. It also provides "reconfiguration-on-fault" fault tolerance when a node or network error occurs. ParaSort automatically reconfigures to account for the failed node(s), and the distributed data is not lost. However, efficient operation requires that major sort axis nodes should reside on near neighbor network physical processors. This minimizes communication costs for efficient operation. And, for a

heterogeneous topology, or a homogeneous topology made irregular by failed nodes, automatically achieving this near neighbor configuration for the sort nodes is challenging.

### *ParaSort Cycles*

Each cycle of ParaSort has two steps: first - sort, partition and send and second - receive and merge. The sort-partition-and-send step occurs at system initialization or whenever the second step is complete and new data arrives from a neighbor. In this step the data are sorted in order on each node and partitioned into quarters, copies of which are then sent off to its four neighbors. The receive-and-merge step occurs when new data arrive. Data are received from neighboring nodes and integrated into the node's current data set. Then the first step begins again.

In an earlier serial simulation written in Object Pascal, ParaSort sorted 12,288 records on 1,024 nodes in 140 cycles. The data consisted of simulated signal data from various sensors: time, location (two UTM coordinates) and signal frequency. We implemented two parallel implementations in C on an Intel Paragon parallel processor for information fusion applications. These simulations validated our approach and provided insights for scaling up the implementation. Recall that information fusion is the analysis and correlation of incoming data to provide a comprehensive assessment of the arriving reports (e.g., for military intelligence correlation, meteorological data interpretation, and real-time economic news synthesis). Thus, it is clear that the first step to information and (lower-level) data fusion applications is often data sorting. Once the data are properly sorted they can be correlated with other data and higher level reports. Details of the ParaSort algorithm (alternately called, "HeteroSort") can be found in [10] and [11].

## 3. THE BEOWULF ARCHITECTURE

### *Background and History*

The Beowulf Project was started at NASA's Center of Excellence in Space Data and Information Sciences ([CESDIS](#)) in the summer of 1994 with the assembly of a 16 node cluster developed for the Earth and space sciences project ([ESS](#)) at the Goddard Space Flight Center ([GSFC](#)). The project quickly spread to other NASA sites, other R&D labs and to universities around the world. The project's scope and the number of Beowulf installations have grown over the years; in fact, they appear to continue to grow at increasing rate.

The original motivation for Beowulf was the need for high performance supercomputing (for example, for data analysis from the spacecraft in the Solar-Terrestrial Probe Line)

where no supercomputer was available and capital funds were limited.

Beowulf clusters are composed of commercial (or commodity) off the shelf (COTS) processors for example, standard personal computers or small scientific workstations, that are linked together to achieve supercomputer performance. Notable Beowulf clusters includes Los Alamos National Lab's Avalon<sup>3</sup> and Oak Ridge National Lab's Stone SouperComputer<sup>4</sup>. Avalon was built at Los Alamos National Laboratory/CNLS. Avalon consists of 140 533MHz DEC AlphaPCs with 256 MB SDRAM, 3Gbyte EIDE drive, connected via 100BaseT Ethernet. The nodes run Linux OS. At press time, the cluster is ranked 160 on the list of top 500 supercomputer sites<sup>5</sup> in the world. Scientific research conducted using Avalaon includes scalable hierarchical particle algorithms for galaxy formation and accretion astrophysics, and phase transitions in the early universe and their analogues in the laboratory. While Avalon demonstrates the power of a Beowulf cluster, Stone SouperComputer demonstrates its flexibility and breadth of accommodation. Oak Ridge's Stone SouperComputer is composed of 126 "found" surplus personal computers. Most (but not all) are Intel 486DX-2/66s or Pentiums. Stone SouperComputer is a research fascination both because of its uniqueness, and the remarkable fact that it works efficiently. Beowulf accommodates a variety of platforms.

The Beowulf software environment is implemented as an add-on to commercially available, royalty-free Linux distributions. It includes several programming environments and development libraries as individually installable packages. PVM, MPI, and BSP are all available. SYS-V--style IPC and p-threads are also supported. A considerable amount of work has gone into improving the network subsystem of the kernel and implementing fast network device support.

In the Beowulf scheme, every node is responsible for running its own copy of the kernel and nodes are generally sovereign and autonomous at the kernel level. However, in the interests of presenting a more uniform system image to both users and applications, the developers have extended the Linux kernel to allow a loose ensemble of nodes to participate in a number of global namespaces. A guiding principle of these extensions is to have little increase in the kernel size or complexity and, most importantly, negligible impact on the individual processor performance.

### *Our Beowulf Cluster*

---

<sup>3</sup> <http://cnls.lanl.gov/avalon/>

<sup>4</sup> <http://stonesoup.esd.ornl.gov/>

<sup>5</sup> <http://www.top500.org>

The Beowulf cluster implemented in the Artificial Intelligence / High Performance and Parallel Computing Lab at the University of St. Thomas comprise 16 SPARC station 5 systems (plus two more, one for cluster control and one for communication) with 32MB RAM and 540MB local disk. The cluster is connected via a Cisco 1200 Ethernet switch 10BaseT, in an enclosed cluster. The current cluster's real-time load is available on the internet at <http://aibc.gps.stthomas.edu/cgi-bin/bc-status.pl>. A current description of the cluster is available at <http://aibc.gps.stthomas.edu>. The operating system for the cluster is LINUX RedHat/5.2.

#### 4. PARASORT ON BEOWULF

The ParaSort implementation on the Beowulf uses the controlling SPARC station to transmit the ParaSort program and the data to be sorted to each of the 16 nodes in the processing cluster. Each node knows its own address and the address of all other nodes so it communicates directly with them. Each node runs ParaSort independently to determine its location in the conceptual torus and to determine which other nodes are its trading partners based on the neighbors' number in the grid and their rank in the sort order. Each node sorts locally, divides the data and sends it to the trading neighbors. Specialized software provides a real-time link to the load levels of the nodes in the cluster demonstrating the processing real-time during the sort process.

##### *Future Work*

We have done some preliminary research into asynchronous cycles for the ParaSort algorithm (detailed in [10]). These are useful near the end of the sorting process when most of the data are in the immediate neighborhood of their proper destination. During this phase, it is useful to suspend trading of the "extremely high" and "extremely low" data across the high-speed transmission routes. So, since each node is only trading productively with its neighbors immediately higher and lower than itself, it restricts its trades to them.

*Best Trades*—At this point, we employ the concept of the "best trade." In the "best trade," the sending node keeps a copy of a quarter of data to be sent. When data arrives from a neighbor it is combined with the copy of the data sent to that neighbor. The new combined list is split in the center and either the top or the bottom half (whichever is appropriate) is kept for the next round of trading. So, for example, node 2 and node 7 may be trading. The appropriate data from node 2 to send to node 8 may be the set (2,5,10). The data from node 8 to node 2 may be (3,4,12). After the data has been sent node 7 combines what it sent with what it received to obtain (2,3,4,5,10,12). Node 7 will keep the top half (5,10,12). Node 2 will have done the

same thing keeping (2,3,4). This is an example of a partial trade in which only some of the data in a given quarter is exchanged. Other alternatives include full trades [as when 2 sends (6,7,8) and 7 sends (1,3,5)] or barren trades [as when 2 sends (1,2,7) and 7 sends (15,16,19)]. A barren best trade occurs when the trading nodes reject all the data their partners have sent. In all these events the "best trade" policy allows each node to transact the optimal transfer with its trading partner. This process eliminates the unfortunate situation where data is simply swapped in oscillating fashion between neighbors, that is, thrashing.

*Transport Optimization*—Another fertile research area proceeds from the observation that the first locations to complete the sort (i.e., have all the data due them in sorted order) are the extremes or ends of the sort sequence. In our example, nodes 1 and 16 would reach this state first. This occurs because extremely high or extremely low data is passed across "express pathways" (transedge) to these nodes very rapidly.

The end nodes 1 and 16 are receiving extreme data from four other nodes as well as keeping their own extreme data. Thus, they receive the data due them (the extremely high or low data in the system) very quickly. When they have received all their data due them they are finished since data on each node is always maintained in sorted order. At this point it makes sense to retire these completed nodes from the sort. This has three effects. First, these nodes are now available for other tasks. For example, these nodes now become available for system monitoring tasks like evaluating the performance of the overall system. Second, minimum and maximum data are located early in the process, if this is useful for other processing. Third, the fact that these nodes are retired means that their immediately adjacent neighbors (node 2, the highest node to 1, and node 15 the next lowest node to 16) now become extremes. These new extremes are now the next nodes to finish, and retire. Thus the system exhibits a completion pattern from the ends inward. This early completion situation applies to the first and last rows also.

*Heterogeneous Architectures*—We have preliminary results indicating that the ParaSort algorithm will work well in a non-homogeneous environment, a network of non-equal nodes and/or asymmetric connections. We intend to explore this lead on this network (implementing asymmetric links) and other networks (we are currently planning a second Beowulf cluster based on Intel based PCs which can alternately be connected with our existing one for heterogeneity). In addition, other architectures (hypercube, for example, as oppose to torus) might prove beneficial.

## 5. CONCLUSIONS

We have demonstrated local control using local knowledge for efficient global processing in ParaSort. The implementation has been shown to provide both fault-tolerance and dynamic load balancing. This is the first implementation of this powerful algorithm on a Beowulf architecture. The Beowulf cluster provides a natural implementation environment for demonstration of these results and for future research.

ParaSort works well within a “uniform communication costs” environment. The Beowulf cluster is wired to connect all nodes through a switch box. That is all nodes are directly linked to all other nodes almost directly or at least at a similar cost /distance. ParaSort’s torus accounts for a variety of topologies including ones where nodes are wired directly only to their four neighbors. In this model, transmission costs are higher when communicating through physical neighbors to other nodes. Various implementations (for example, Intel’s Paragon) compensate for this with wormholes to speed packets through intervening nodes. Thus, ParaSort imposes its snake like grid with torus edge connections and each node works with just four neighbors.

For the Beowulf cluster  $n-1$  neighbors or a number of neighbors between four and  $n-1$  might prove beneficial. Future tests will be needed to determine the optimal. Future tests will be needed to determine the optimal of neighbor nodes.

A variety of research questions present themselves for future work. For example, the topology might be able to be redesigned (for example, from a torus sheet to a hypercube). The issue of what is a row and column in such an environment becomes interesting. In addition, we intend to add the Intel PC cluster parallel to the SPARC cluster. A future challenge will be for a ParaSort to treat the two clusters as part of one heterogeneous system.

The Beowulf cluster is not designed for large amounts of I/O. Yet ParaSort moves data as an essential method for sorting. It is generally optimal to move data in as few cycles as possible in any system to “get the work done” ASAP. But in a Beowulf cluster environment it is also important to move data among few nodes as possible.

The “best trade” feature of ParaSort plays an important role in this area. Usually it is useful to avoid oscillation of data back and forth between two trading partners. In a Beowulf cluster, the best trade with the memory of the last trade, can provide local resources to suspend, based on either heuristics or other rules, transferring data that has not been fruitfully traded recently.

## 6. REFERENCES

- [1] Jay Strosnider, Department of Electrical and Computing Engineering, Carnegie Mellon University, Fault-Tolerant Real Time Computing Project.
- [2] Katcher, Daniel I., Jay K. Strosnider, and Elizabeth A. Hinzelman-Fortino. "Dynamic versus Fixed Priority Scheduling: A Case Study"  
<http://usa.ece.cmu.edu/Jteam/papers/abstracts/tse93.abs.html>
- [3] Ramos-Theul, Sandra and Jay K. Strosnider. "Scheduling Fault Tolerant Operations for Time-Critical Applications."  
<http://usa.ece.cmu.edu/Jteam/papers/abstracts/dcca94.abs.html>.
- [4] Theul, Sandra. "Enhancing the Fault Tolerance of Real-Time Systems Through Time Redundancy."  
<http://usa.ece.cmu.edu/Jteam/papers/abstracts/thuel.abs.html>.
- [5] Chengzhong Xu and Francis C. M. Lau, *Load Balancing in parallel Computers: Theory and Practice*, Boston: Kluwer Academic Publishers, 1997, p. 122.
- [6] Lin, Yen-Chun. "On Balancing Sorting on a Linear Array." *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no 5, pp. 566-571, May 1993.
- [7] Thompson, C.D., and H.T. Kung. "Sorting on a Mesh-connected Parallel Computer." *Communications of the ACM*, vol. 20, no 40, pp. 263-271, April 1977.
- [8] Scherson, Isaac D., and Sandeep Sen. "Parallel Sorting in Two-Dimensional VSLI Models of Computation." *IEEE Transactions of Computers*, vol. 38, no 2, pp. 238-249, February 1989.
- [9] Gu, Qian Ping, and Jun Gu. "Algorithms and Average Time Bounds of Sorting on a Mesh-Connected Computer." *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no 3, pp. 308-315, March 1994.
- [10] Davis, E., Wren, W., and Bennett, B.H., "Reconfigurable Parallel Sorting and Load Balancing: ParaSort", *Proceedings of the 4<sup>th</sup> International Workshop on Embedded HPC Systems and Applications (EHPC'99)*, 16 April 1999.
- [11] Davis, E., W. Wren, and B.H. Bennett, "Fault-Tolerant High-performance Parallel Sorting for Load Balancing Across Heterogeneous Networks," *2nd Annual Workshop on Embedded High Performance Computing Systems and Applications*, Geneva, Switzerland, 5 April 1997.

## 7. BIOGRAPHIES

**Bonnie Holte Bennett, Ph. D.**, is an Associate Professor in the Graduate Programs in Software at the University of St. Thomas in St. Paul, Minnesota. She is the founder and director of the Artificial Intelligence/High Performance and Parallel Computing Lab. She is also a principal with Knowledge Partners of Minnesota, Inc. ([www.kpmi.com](http://www.kpmi.com)). She consults extensively with Fortune 200 companies in the areas of knowledge-based systems and knowledge management. Previously, she spent 14 years at the Honeywell Technology Center in Minneapolis, Minnesota. She holds Ph.D. and M.S. degrees in Computer Science from the University of Minnesota, and a Bachelors degree in Computer Science and English from the College of St. Thomas.



**Emmett Davis**, is manager of the Artificial Intelligence/High Performance and Parallel Computing Lab at the University of St. Thomas in St. Paul, Minnesota. He holds an M.S. in Software Engineering from the University of St. Thomas, M.A.s in Library Science and History/Geography from the University of Minnesota, and a B.A. in History from the State University of New York at Albany. He is an Information Technology Supervisor for an Application Development Unit for Hennepin County.

**Timothy M. Kunau**, is on academic staff at the University of Minnesota where he designs information systems for genomics and computational biology. Tim has worked in high performance computing research for a decade. Prior to that he was with Cray Research/Silicon Graphics. He holds an M.S. in Software Engineering from the University of St. Thomas, and Bachelors degrees in Computer Science and Philosophy from Luther College in Decorah, Iowa.



**William Wren**, is a Senior Staff Research Scientist at the Honeywell Technology Center in Minneapolis, Minnesota. He has over thirty years experience designing and implementing high-performance computing systems. He holds an BSEE from the University of Nebraska, and did graduate research work at Arizona State University.

