Reconfigurable Parallel Sorting and Load Balancing: HeteroSort

Emmett Davis[1], Dr. Bonnie Holte Bennett[1], Bill Wren [2], Dr. Linda Davis[1],

[1] University of St. Thomas, Graduate Programs in Software, Mail # OSS 301, 2115 Summit Avenue, Saint Paul, MN
EmmettDa@aol.com, bhbennett@stthomas.edu
[2] Honeywell Technology Center, 3660 Technology Drive, Minneapolis, MN 55418
wrenpc@worldnet.att.net

Abstract. HeteroSort load balances and sorts within static or dynamic networks. Upon failure of a node or path, HeteroSort uses a genetic algorithm to minimize the distribution path by optimally mapping the network to physical near neighbor nodes. We include a proof that a final state of HeteroSort (barren best trades of the Wren cycle) is always a detection of termination of sorting. By capturing global system knowledge in overlapping microregions of nodes, HeteroSort is useful in data dependent applications such as data information fusion on distributed processors.

1 Introduction

Dynamic adaptability is a keystone feature for applications implemented on modern networks. Dynamic adaptability is a basis for fault tolerance. A system, which is dynamically adaptive, strives to withstand the assault of hardware glitches, electrical spikes, and component destruction. The research described in this paper set out to develop a high-speed load balancing algorithm, which would balance loads by sorting data across the network of nodes and resulted in developing a reconfigurable system for parallel sorting with dynamic adaptability.

1.1 Dynamic Adaptability

With the increased dependence on distributed and parallel processing to support general as well as safety-critical applications, we must have applications that are fault tolerant. 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]

We have taken a different approach and refocused on the design of elemental processes such as load balancing and sorting. 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.

1.2 Information and Data Fusion Applications

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. 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. Furthermore, this is of growing importance now.

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

[1] Examples of these fault tolerant efforts can be found in the work of Jay Strosnider and his colleagues at Department of Electrical and Computing Engineering, Carnegie Mellon University in the Fault- Tolerant Real Time Computing Project. 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.
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.
Theul, Sandra. "Enhancing the Fault Tolerance of Real-Time Systems Through Time Redundancy." http://usa.ece.cmu.edu/Jteam/papers/abstracts/thuel.abs.html.

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. Furthermore, 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.

1.3 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).

A significant practical feature of HeteroSort is that in our experiments it load balances before it finishes sorting. Since HeteroSort detects when the system is sorted, it also detects termination of load balancing. Chengzhong Xu and Francis Lau in Load Balancing in Parallel Computers: Theory and Practice (Boston: Kluwer Academic Publishers, 1997) state:

> 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.[2]

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. HeteroSort combines load balancing within sorting processes.

[2] Xu, Chengzhong and Francis C.M. Lau. *Load Balancing in Parallel Computers: Theory and Practice*. (Boston: Kluwer Academic Publishers, 1997), p. 122.

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 HeteroSort, which requires no external tables.

1.4 Related Work

Much of the work in this area deals with linear arrays.[3] 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.[4] 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.

1.5 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.

[3] 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.
Thompson, C.D., and H.T. Kung. "Sorting on a Mesh-connected Parallel Computer." *Communication of the ACM*, vol 20, no 40, pp. 263-271, April 1977.

[4] 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.
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.

2 Approach

HeteroSort 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

Fig. 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 neighbors for Node 7 are 2, 6, 8, and 10. When Node 7 receives its initial data, it sorts it and splits it into four quarters. The lowest quarter goes to Node 2, the next lowest quarter goes to Node 6, the third quarter goes to Node 8, and the highest quarter goes to Node 10. Thus, the extremely high and low data are shipped on "express pathways" across the coils of the snake network.

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.

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

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 (transedge 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 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 systems 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, HeteroSort has three strategies. The first is to still implement the conceptual torus at the higher transmission cost. The second is to re-configure 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.

2.1 Optimization of HeteroSort

HeteroSort'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. HeteroSort 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 difficult.

Figure 2 indicates a homogeneous mesh made irregular by two failed nodes. The numbers in the boxes (nodes) indicate the node's position in the sort order.

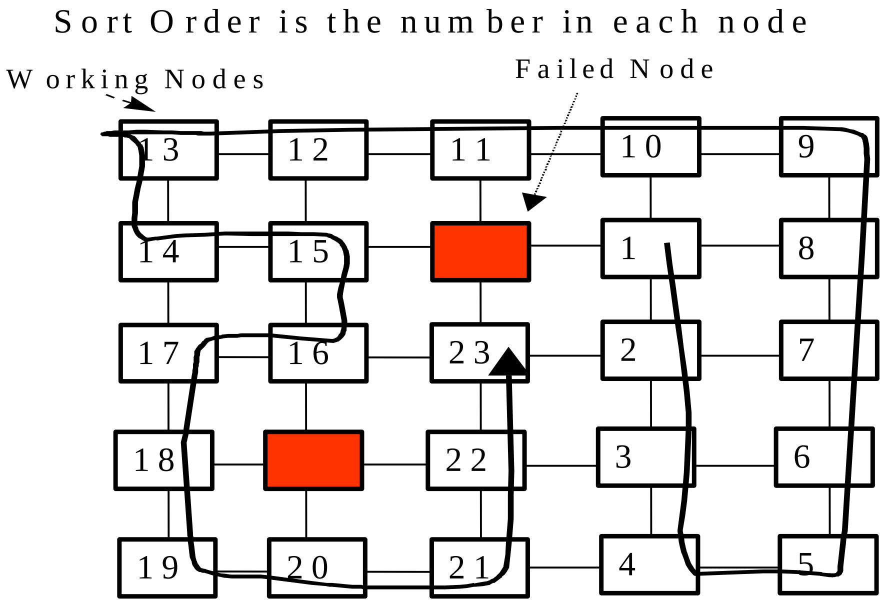

Fig. 2. Sort order is the number of each node. A homogeneous mesh of 25 nodes made irregular by two failed nodes requires a new sort order for efficient performance. The numbers in the boxes (nodes) indicate the node's position in the new sort order. The lowest valued items in the sort will eventually reside on processor 1 and the highest valued items on processor 23. This new order optimizes near neighbor relations.

We assume that a message cannot be sent across a failed node. To provide for online reconfiguration of the node sort order, we have developed an adaptive online sort order optimizer named the Scaleable Adaptive Load-balancing (SAL) Online Optimizer (SOO). Shown in Figure 3 is a result of running SOO. SOO is performed by using a genetic algorithm which minimizes the total path length of the HeteroSort major sort axis, indicated on the figure by the line from one to 23. Note that other possible minimum path sort orders exist. Also note that for some topologies or failure patterns, strict near neighborness is not achievable. For these cases SOO defines the minimum path that includes store-and-forwards or traversals across other nodes. SOO can optimize given any combination of failed nodes and busses.

3 The SOO Genetic Optimization Methodology

In order for the SOO to be most useful in a wide variety of networks, SOO must be able to efficiently adapt the HeteroSort algorithm to irregular topologies. Even if the target network is working perfectly, nodal resources may periodically be available and then become unavailable.

SOO is able to adapt the HeteroSort algorithm architecture to a wide variety of topological changes because SOO is intended to operate online on the target architecture itself. Furthermore, we explicitly designed SOO to be executed in scaleable parallel fashion in order to provide high performance.

The SOO genetic algorithm (GA) is very similar to the GA used for the Traveling Salesperson Problem. In general, GA requires a means for initially creating possible solutions, calculating the fitness of each trial solution, and a means for creating new trial solutions (reproduction). To provide for computing the fitness functions, SOO uses a utility named the Dynamic System Optimizing Simulator (DySOS). DySOS is resident on each node(s) of the network that calculates the new sort order after a network configuration change.

It is assumed that the network HeteroSort is operating on is instrumented so that when the state of the network changes these changes may be input to SOO so that reconfiguration of the HeteroSort may be calculated. The steps employed by SOO in reconfiguring its network are as follows:

1. Topology / Node information is received from the network via instrumentation.

2. DySOS calculates new routing tables from the topology/node information. These tables represent the shortest distance between any two nodes in the network.

3. The new routing tables are loaded into the DySOS fitness function calculator.

4. SOO uses genetic algorithms (GA) to compute the new HeteroSort sort file order.

If the GA is computed in parallel, each node running SOO/DySOS has its own separate population of trial solutions, and computes its own fitness functions independently of the other node(s). After each generation of calculation, each node sends its best solution (most fit) to a command tree which coordinates all SOO nodes. This command node selects the best single solution from all the nodes, and then broadcasts this solution to all the nodes. This best solution is then used to in the next generation of trial solutions. This technique is named Elitism, and enables the many SOO nodes to effectively cooperate in calculating the best solution for the SAL HeteroSort.

A disadvantage of GA is running time. A complex GA solution for a large network can be lengthy, even when performed in parallel. To help obviate this shortcoming, the SOO GA saves its solutions, so that if the changes made to the network are incremental, by using the previous solution space as a start, the next solution can be made more quickly, than if the SOO were started from scratch each optimization period. However, the advantages GA brings to SOO far outweigh the performance shortcomings. These advantages are:

1. Robustness - The SOO and its GA always yields a solution that works for any topology.
2. Arbitrary networks with nonlinear properties can be modeled.
3. The GA is scaleable to high performance.

SOO and DySOS are currently implemented in Visual Basic running on PCs.

4 HeteroSort Operation

Each cycle of HeteroSort 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 is sorted in order on each node and partitioned into quarters, copies of which are then sent off to the neighbors in the method described above.   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.

4.1 Best Trade

The integration process includes the concept of the “best trade.” In the “best trade,” a copy of a quarter of data to be sent is kept by the sending node. 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

4.2 Transport Optimization

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 next 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 is expected to exhibit a completion pattern from the ends inward. This early completion situation applies to the first and last rows also.

Furthermore, after a number of cycles there is a time when we can switch to a smaller number of trading partners where only the immediately adjacent neighbors (next highest and next lowest) are the trading partners. This is done because most of the data are on their eventual destination nodes or immediate vicinity. Thus, we can maximize linear trading.

4.3 Dual Configuration Cycle Sorting

HeteroSort sorts data by having each node use a combination of directional transfers, both left and right on the linear sort (translinear) and up and down across the coils of the snake network (transcoil). In the torus configuration, the edges of the net itself are bridged (transedge). The use of transcoil and transedge data movement improves the efficiency of the transportation of data when large amounts of the data is highly unsorted. These transcoil and transedge bridges, however, introduce side effects.

HeteroSort uses alternating odd and even cycles to compensate for these side effects. Using inequalities, we can provide insight into the side effects and how the odd and even cycles compensate for the side effects. When HeteroSort using only a single configuration cycle with transcoil and/or transedge data movements and best trading ceases to have productive trades, are the records totally ordered, that is, sorted? In other words, when best trades are no longer productive has the system stalled before full sorting or has full sorting always occurred?

This is important for the implementation of the algorithm, where we depend on local knowledge to govern the system. Barren best trades could indicate that sorting is completed, if we could depend on barren trades indicating only full sorts and not either full or stalled sorts. Experimentally, when the HeteroSort consists of only a single trading cycle (not alternating odd and even cycles) and best trades, the records can stall before they are completely sorted.

4.4 Logical Basis for Dual Cycles

Definition: Let u and v be finite sets of real numbers. The set u is said to be less than or equal to the set v, written $u \le v$, if every number in u is less than or equal to every number in v. It is easy to see that $\le$ has the transitive property, i.e. if u, v, w are three finite sets of real numbers satisfying $u \le v$ and $v \le w$ then $u \le w$. To see this let a be any element of u, b be any element of v, and c be any element of w. Since $u \le v$ and $v \le w$, we have $a \le b$ and $b \le c$. Then, of course, $a \le c$.

Comment: The symbol $\le$ is being used in two different ways: to compare sets of numbers and to compare numbers in the usual manner.

Let k be a positive integer. Define Node k to a collection of four finite sets a_k, b_k, c_k, and d_k of real numbers where k is a positive integer. Let N be a positive integer.

Let Node 1, …, Node N be a collection of N nodes. Refer to the collection of all of the sets a_1 , b_1 , c_1 , d_1 , … a_N , b_N , c_N , d_N as data, and the individual sets as parcels.

Definition: The data on nodes Node 1, … , Node N is said to be sorted if and only if the following three conditions are satisfied

1. Each one of the sets a_k , b_k , c_k , d_k , $1 \le k \le N$, is in increasing order.
2. $a_k \le b_k \le c_k \le d_k$ for all k, $1 \le k \le N$.
3. $u \le v$ whenever
   - $u = a_i$, b_i , c_i , or d_i
   - $v = a_j$, b_j , c_j , or d_j
   - $i < j$, $1 \le i \le N$, $1 \le j \le N$.

Now assume that there are sixteen nodes Node 1, …, Node 16 arranged as in Figure 1. Assume further that the parcels will be traded and compared as described in the HeteroSort and Best Trade Sections. A more detailed description of the trading and comparing is given below.

The trading partners' chart for the odd and even cycles is given in Table 1. The construction of the odd cycle column was described in the Approach Section. During the odd cycle, the trading partners are arranged from lowest to highest according to the subscript of the node. The trading partners list for the even cycle is obtained from the odd cycle list by exchanging the order of the highest and lowest partners lower than the node, and exchanging the highest and lowest partners above the node. Three representative examples are given below.

Node 1 order for odd cycle: Node 2, Node 4, Node 8, Node 16. Lower neighbors: none. Higher neighbors: Node 2, Node 4, Node 8, Node 16. To obtain order for the even cycle: exchange Node 2 and Node 16. Order for the even cycle: Node 16, Node 4, Node 8, Node 2.

Node 5 order for odd cycle: Node 4, Node 6, Node 8, Node 12. Lower neighbors: Node 4. Higher neighbors: Node 6, Node 8, Node 12. To obtain order for the even cycle: exchange Node 6 and Node 12. Node 4 is the only lower node. There is no exchange. Order for the even cycle: Node 4, Node 12, Node 8, Node 6.

Node 7 order for odd cycle: Node 2, Node 6, Node 8, Node 10. Lower neighbors: Node 2 and Node 6. Higher neighbors: Node 8 and Node 10. To obtain order for the even cycle: exchange Node 2 and Node 6. Exchange Node 8 and Node 10. Order for the even cycle: Node 6, Node 2, Node 10, Node 8.

The Trading Partners chart (Table 1) can be used to decide which parcels are sent to which neighbors during the odd and even cycles. As an example, consider what happens on Node 5.

During the odd cycle:

Node 5 sends a_5 to Node 4 and receives c_4 from Node 4.
Node 5 sends b_5 to Node 6 and receives b_6 from Node 6.
Node 5 sends c_5 to Node 8 and receives b_8 from Node 8.
Node 5 sends d_5 to Node 12 and receives a_{12} from Node 12.

During the even cycle:

Node 5 sends a_5 to Node 4 and receives d_4 from Node 4.
Node 5 sends b_5 to Node 12 and receives c_{12} from Node 12.
Node 5 sends c_5 to Node 8 and receives b_8 from Node 8.
Node 5 sends d_5 to Node 6 and receives a_6 from Node 6.

The best trades on Node k, as described in the section Best Trades, are barren on the odd or even cycle if there are no changes in the parcels a_k, b_k, c_k, and d_k after the best trades have been performed. The question this section seeks to answer is, can barren best trades be used as a criterion to stop the sort? Specifically:

What is the order of the data when the trades are barren during the odd cycle?
What is the order of the data when the trades are barren during the even cycle?

The Barren Best Trades chart (Table 3) is constructed to answer this question. The Barren Best Trades chart lists the relationships which exist among the groups of data on the nodes when the best trades are barren on the odd cycle, and when the best trades are barren on the even cycle. First of all, the data on each node is ordered, i.e. $a_k \le b_k \le c_k \le d_k$. This is true for both cycles.

Barren Best Trade Examples for Odd Cycle, Using the Trading Partners List.
Node 1: Node 1 and Node 2 trade parcels a_1 and a_2 . Since the best trade is barren, all of the records in group a_1 must be smaller than those in a_2 . This can be expressed with the notation $a_1 \le a_2$. Node 1 and Node 4 trade parcels b_1 and a_4. Since the best trade is barren, $b_1 \le a_4$. Node 1 and Node 8 trade parcels c_1 and a_8. Since the best trade is barren, $c_1 \le a_8$. Node 1 and Node 16 trade parcels d_1 and a_{16} . Since the best trade is barren, $d_1 \le a_{16}$.

Node 2: Node 2 and Node 1 trade parcels a_2 and a_1 . Since the best trades are barren, $a_1 \le a_2$. This relationship has already been listed under Node 1 and is not repeated. Redundant relationships are not repeated on the Barren Best Trades Chart. Node 2 and Node 3 trade parcels b_2 and a_3 . Since the best trade is barren, $b_2 \le a_3$. Node 2 and Node 7 trade parcels c_2 and a_7 . Since the best trade is barren, $c_2 \le a_7$. Node 2 and Node 15 trade parcels d_2 and a_{15} . Since the best trade is barren, $d_2 \le a_{15}$.

Node 5: Node 5 and Node 4 trade parcels a_5 and c_4. This relationship has already been listed under Node 4 and is not repeated. Node 5 and Node 6 trade parcels b_5 and b_6. Since the best trade is barren, $b_5 \le b_6$. Node 5 and Node 8 trade parcels c_5

and b_8. Since the best trade is barren, $c_5 \leq b_8$. Node 5 and Node 12 trade parcels d_5 and a_{12}. Since the best trade is barren, $d_5 \leq a_{12}$.

This process is repeated for the remainder of the sixteen nodes.

Barren Best Trade Examples for Even Cycle, Using the Trading Partners List.
Node 1: Node 1 and Node 16 trade parcels a_1 and d_{16}. Since the best trade is barren, $a_1 \leq d_{16}$. Node 1 and Node 4 trade parcels b_1 and b_4. Since the best trade is barren, $b_1 \leq b_4$. Node 1 and Node 8 trade parcels c_1 and c_8. Since the best trade is barren, $c_1 \leq c_8$. Node 1 and Node 2 trade parcels d_1 and a_2. Since the best trade is barren, $d_1 \leq a_2$.

Node 5: Node 5 and Node 4 trade parcels a_5 and d_4. This relationship has already been listed under Node 4 and is not repeated. Node 5 and Node 12 trade parcels b_5 and c_{12}. Since the best trade is barren, $b_5 \leq c_{12}$. Node 5 and Node 8 trade parcels c_5 and b_8. Since the best trade is barren, $c_5 \leq b_8$. Node 5 and Node 6 trade parcels d_5 and a_6. Since the best trade is barren, $d_5 \leq a_6$.

This process is repeated for the remaining fourteen nodes.

<u>If the best trades are barren on the even cycle, the Wren cycle, the file is sorted.</u>

Proof: If the best trades are all barren on the even cycle, then the Barren Best Trade Chart (Table 3) reveals that the following relationships hold.

$$
\begin{array}{lll}
a_1 \leq b_1 \leq c_1 \leq d_1 & d_1 \leq a_2 & \\
a_2 \leq b_2 \leq c_2 \leq d_2 & d_2 \leq a_3 & \\
a_3 \leq b_3 \leq c_3 \leq d_3 & d_3 \leq a_4 & \ldots \\
 & & \\
a_{15} \leq b_{15} \leq c_{15} \leq d_{15} & d_{15} \leq a_{16} & \\
a_{16} \leq b_{16} \leq c_{16} \leq d_{16} & &
\end{array}
$$

The transitive property of $\leq$ now implies that the data are sorted. Note that not all of the inequalities on the barren Best Trades chart are needed.

Barren best trades on the odd cycle do not guarantee that a sorted order has been achieved. This has been demonstrated experimentally on data sets. Examination of the Barren Best Trades chart for the odd cycle reveals that there are not enough inequalities present to get a comparison of d_1 and a_2 or d_2 and a_3, et cetera through use of transitivity.

The results in this section can be extended to an arbitrary N x N network.

Table 2. Barren Best Trades Chart. This table contains the relations needed to prove that both the odd and even cycles constitute a total order and lead to a full sort. The inequalities immediately following the node numbers hold for both even and odd cycles.

Node	Odd Cycle	Even Cycle
1	$a_1 \le b_1 \le c_1 \le d_1$	
	$a_1 \le a_2$	$a_1 \le d_{16}$
	$b_1 \le a_4$	$b_1 \le b_4$
	$c_1 \le a_8$	$c_1 \le c_8$
	$d_1 \le a_{16}$	$d_1 \le a_2$
2	$a_2 \le b_2 \le c_2 \le d_2$	
	$b_2 \le a_3$	$b_2 \le c_{15}$
	$c_2 \le a_7$	$c_2 \le b_7$
	$d_2 \le a_{15}$	$d_2 \le a_3$
3	$a_3 \le b_3 \le c_3 \le d_3$	
	$b_3 \le a_4$	$b_3 \le c_{14}$
	$c_3 \le a_6$	$c_3 \le b_6$
	$d_3 \le a_{14}$	$d_3 \le a_4$
4	$a_4 \le b_4 \le c_4 \le d_4$	
	$c_4 \le a_5$	$c_4 \le b_{13}$
	$d_4 \le a_{13}$	$d_4 \le a_5$
5	$a_5 \le b_5 \le c_5 \le d_5$	
	$b_5 \le b_6$	$b_5 \le c_{12}$
	$c_5 \le b_8$	$c_5 \le b_8$
	$d_5 \le a_{12}$	$d_5 \le a_6$
6	$a_6 \le b_6 \le c_6 \le d_6$	
	$c_6 \le b_7$	$c_6 \le b_{11}$
	$d_6 \le a_{11}$	$d_6 \le a_7$
7	$a_7 \le b_7 \le c_7 \le d_7$	
	$c_7 \le c_8$	$c_7 \le b_{10}$
	$d_7 \le a_{10}$	$d_7 \le a_8$
8	$a_8 \le b_8 \le c_8 \le d_8$	
	$d_8 \le a_9$	$d_8 \le a_9$
9	$a_9 \le b_9 \le c_9 \le d_9$	
	$b_9 \le b_{10}$	$b_9 \le b_{16}$
	$c_9 \le b_{12}$	$c_9 \le b_{12}$
	$d_9 \le b_{16}$	$d_9 \le a_{10}$
10	$a_{10} \le b_{10} \le c_{10} \le d_{10}$	
	$c_{10} \le b_{11}$	$c_{10} \le b_{15}$
	$d_{10} \le b_{15}$	$d_{10} \le a_{11}$
11	$a_{11} \le b_{11} \le c_{11} \le d_{11}$	
	$c_{11} \le c_{12}$	$c_{11} \le b_{14}$
	$d_{11} \le b_{14}$	$d_{11} \le a_{12}$
12	$a_{12} \le b_{12} \le c_{12} \le d_{12}$	
	$d_{12} \le b_{13}$	$d_{12} \le a_{13}$
13	$a_{13} \le b_{13} \le c_{13} \le d_{13}$	
	$c_{13} \le c_{14}$	$c_{13} \le c_{16}$
	$d_{13} \le c_{16}$	$d_{13} \le a_{14}$
14	$a_{14} \le b_{14} \le c_{14} \le d_{14}$	
	$d_{14} \le c_{15}$	$d_{14} \le a_{15}$
15	$a_{15} \le b_{15} \le c_{15} \le d_{15}$	
	$d_{15} \le d_{16}$	$d_{15} \le a_{16}$
16	$a_{16} \le b_{16} \le c_{16} \le d_{16}$	

6 Results: Graphs

We implemented HeteroSort on two computer environments: a serial simulation and a parallel processor. These implementations worked as predicted. In a serial simulation, HeteroSort 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. The initial state is where the lowest node has the highest valued records, that is, the data is in the reverse order.

We used a snake pattern of nodes with a torus topology. The sort was simulated on an Intel Pentium Pro, Windows 95 machine using Borland International's Delphi, that is, essentially Pascal with Paradox databases. Figures 4 through 9 show Maple V plots of the results of the initial state, the final 140th cycle, and the 2nd through the Xth cycle's results.

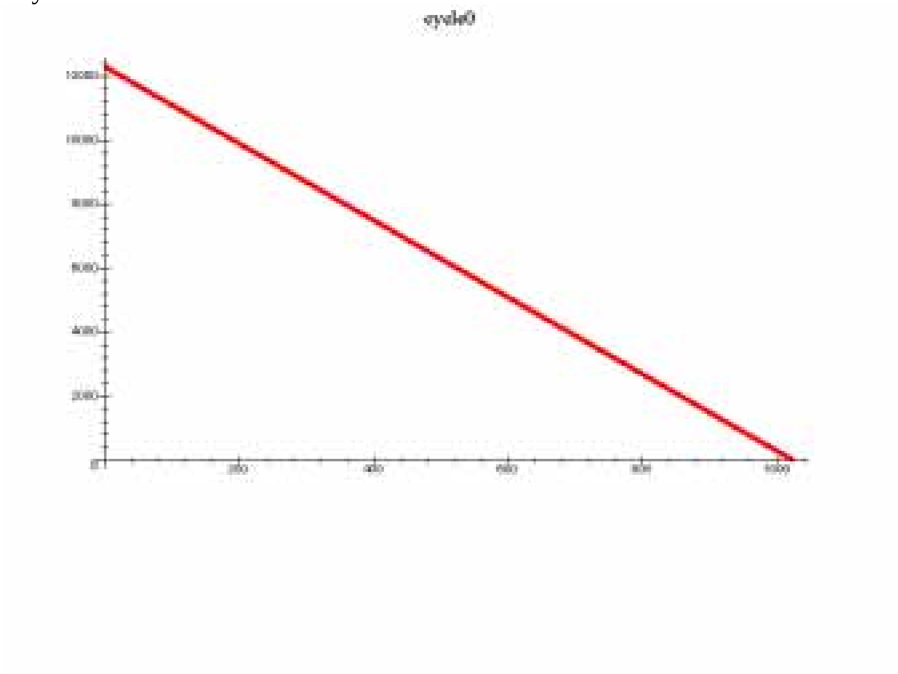

Fig. 3. 12,288 records across 1,024 nodes at the initial 0th cycle of the sort. The x axis is the number of the node and the y axis is the value of data, in this instance, the first UTM location coordinate.

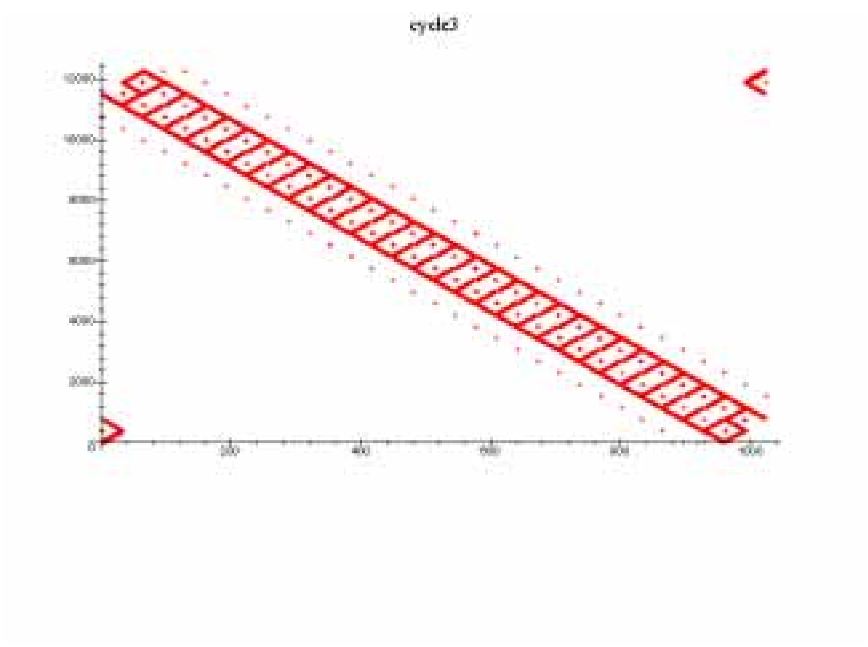

Fig. 4. 12,288 records across 1,024 nodes at the 3^d cycle of the sort.

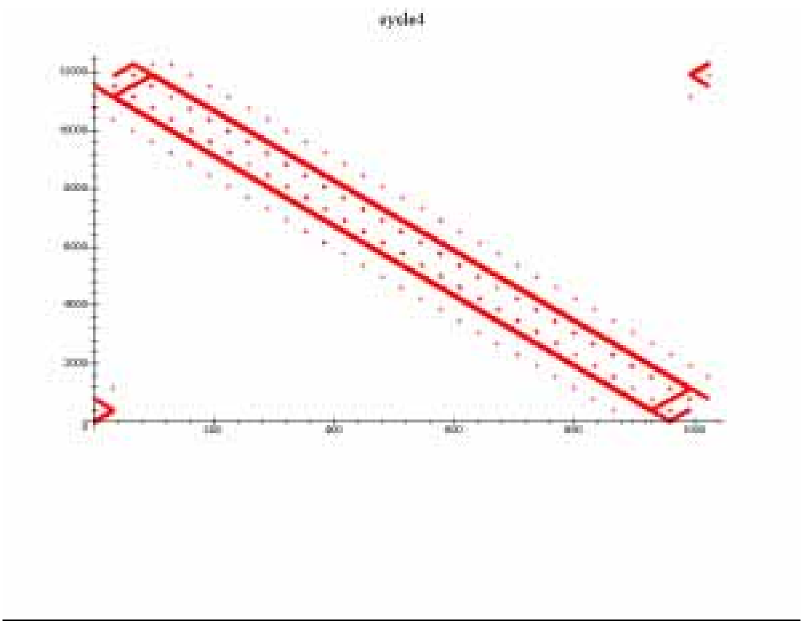

Fig. 5. 12,288 records across 1,024 nodes at the 4th cycle of the sort.

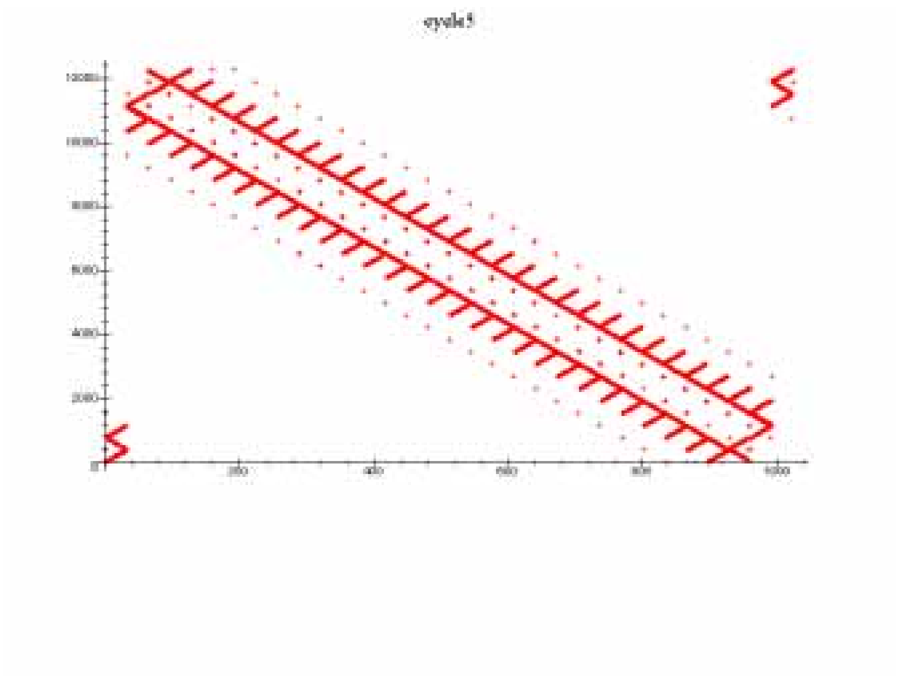

Fig. 6. 12,288 records across 1,024 nodes at the 5th cycle of the sort.

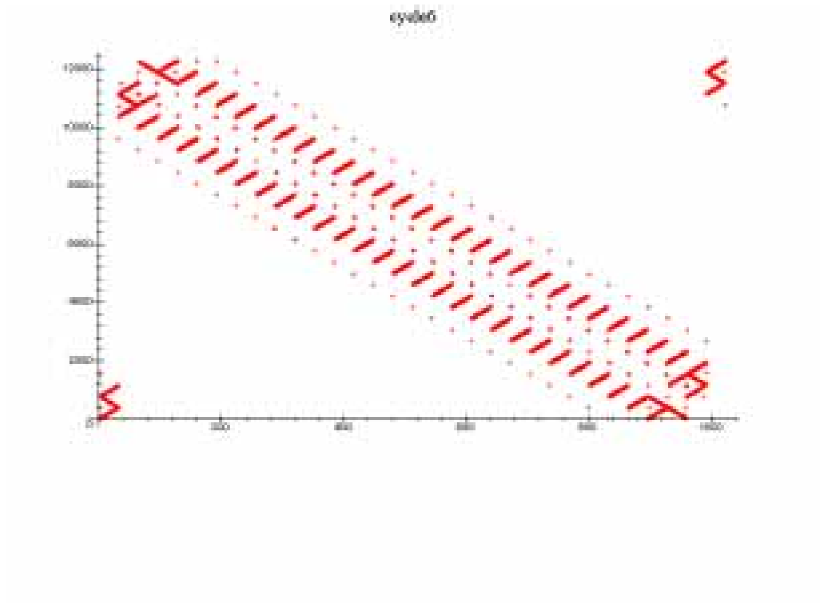

Fig. 7. 12,288 records across 1,024 nodes at the 6th cycle of the sort.

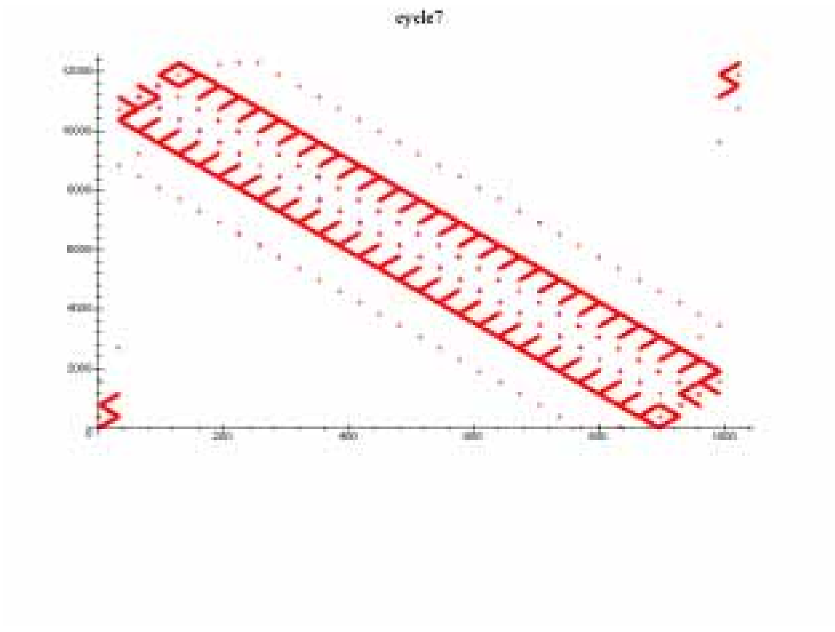

Fig. 8. 12,288 records across 1,024 nodes at the 7th cycle of the sort.

Note the four phase pattern to the borders of the graphs' rectangle. They phases reflect the results of the odd and even cycles.

The hashing pattern reflects the impact of the snake pattern. In this case the 1,024 nodes are ordered in 32 rows with a snake formation. The formation, as noted in Figure 1, reverses the sort order for every other row.

The first and last rows are sorted early in the process and are not represented as other rows are in the graphics.

Also note the individual points above and below the rectangle. These are indications of inefficiencies in the process as these data require additional sorting cycles. We are currently studying ways to optimize the sorting processes and sort these outliers earlier.

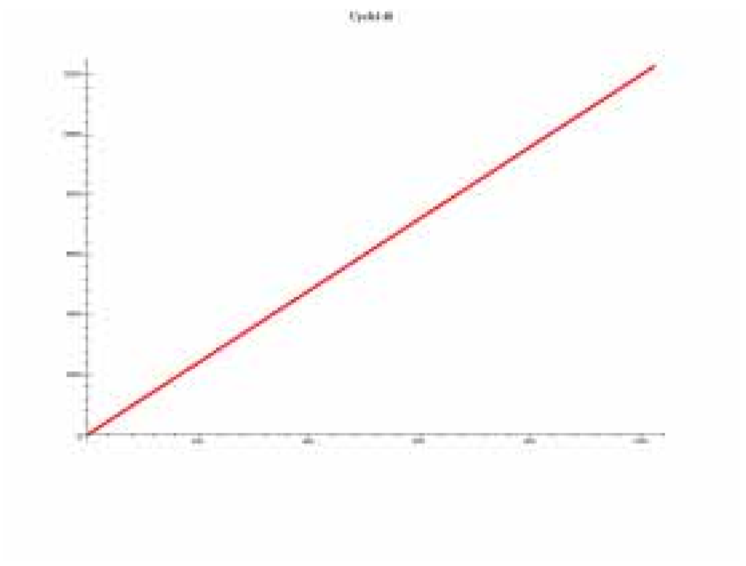

Fig. 9. 12,288 records across 1,024 nodes at the final 140th cycle of the sort.

7 Implementations

The initial serial simulations of nine and sixteen node networks of this algorithm are written in C on a personal computer. These simulations validated our approach and provided insights for scaling up the implementation. They performed as we predicted with the diameter of the network and depth of the connection graphs.

We implemented two parallel implementations in C on an Intel Paragon parallel processor. One implementation has nine nodes, the other sixteen nodes. Again, these implementations validated our predictions. The current simulations are written in Delphi, Borland's Object Pascal. The networks vary from nine to 1,024 nodes.

7.1 Fault Tolerance

The most important aspect of our algorithm is that it does not depend on a regular network topology (as, for example, a traditional shear sort does) because the torus can be superimposed on any physical architecture.

This yields fault tolerance because our system can dynamically reconfigure itself, and easily accommodates "holes" in the connection. All that is required is for HeteroSort to change the partitioning schema in the data, and to stop sending data to a node when it is removed.

Three other aspects of fault tolerance result from this algorithm. First, since only local knowledge is used in the sort, the system is fault tolerant because it does not require global knowledge. Thus, individual nodes continue to operate regardless of the performance (or even existence) of other non-neighbor nodes. Second, since each node keeps a backup copy of the data it sends off to its neighbors, if a node is eliminated during operation of the load balancing and sorting, its neighbors can make up for the loss of data. Third, the natural load balancing of the data during operation of the sorts adds a degree of fault tolerance. With data evenly distributed across nodes, then the loss of a node means the minimal loss of data to the system. The intent is to build minimum weight spanning trees and to use them in improving sort efficiency.

7.2 Future Directions

We currently have the concept of near (adjacent) neighbors and far neighbors (which exist with the implementation of the torus structure). This has implications for implementations on heterogeneous and distributed networks. Specifically, the far neighbors are metaphors for nodes on another processor in a distributed system. So, one component of the sort, partition, and send task could be that the data is partitioned not into equal subsets, but into subsets of a size proportional to the speed of the link to that node.

Furthermore, in heterogeneous architectures, the subset size could also be related to the speed of the corresponding neighbor node. Thus, future enhancements will include an applications kernel that will be resident on each node of the heterogeneous network. Upon startup, each kernel will negotiate with its near neighbor kernels to adjust the size of the exchange list (to be load balanced and sorted). The negotiated value will be a function of each node’s own capacity in memory, processing, and its number of neighbors. Upon a fault, the kernels will re-negotiate the exchange files with the surviving near neighbors.

The primary implementations of HeteroSort will be in distributed workstation or parallel processing environments where workstations drop in and out of the shared work pool. In addition, high performance, real-time applications which require clustering, tracking, data fusion and other “assignment intensive” problems with non-uniformly populated data spaces will benefit from our algorithm.

Acknowledgments

This research was partially supported by a grant from the Defense Nuclear Agency 93DNA-3. We gratefully acknowledge this support.