Cluster Computing Architectures and Algorithms for Passive Sonar Arrays

by A. George, W. Rosen[†], J. Markwell, L. Hopwood, and R. Fogarty

High-performance Computing and Simulation (HCS) Research Laboratory
Department of Electrical and Computer Engineering
University of Florida

Abstract

This paper summarizes ongoing research and development efforts involving the design, modeling, simulation, experimentation, and analysis of distributed, parallel computing algorithms and architectures for autonomous sonar arrays. Several fine-grain interconnection network models have been developed including unidirectional linear array, bidirectional linear array, and unidirectional register-insertion ring. Basic time- and frequency-domain algorithms for parallel beamforming have been designed and implemented, and promising performance results have been achieved on experimental Ethernet-, ATM-, and SCI-based cluster testbeds in the HCS Research Lab.

1. Introduction

Quiet submarine threats and high clutter in the littoral undersea environment demand that higher-gain acoustic sensors be deployed for undersea surveillance. This trend requires the implementation of high-element-count sonar arrays and leads to a corresponding increase in data rate and the associated signal processing. The U.S. Navy is developing a series of low-cost, disposable, battery-powered, and rapidly-deployable sonar arrays for undersea surveillance. The algorithms being mapped to these and other sonar arrays are computationally-intensive, particularly when environmentally adaptive for detection and classification. As a result, the processing and dependability requirements placed on the data collector/processor, the monolithic embedded computer required to collect and process sensor data in a real-time fashion, are becoming prohibitive in terms of cost, electrical power, size, and weight.

Parallel processing techniques together with advanced networking and distributed computing technologies and architectures can be used to transform the telemetry nodes of these autonomous sonar arrays into processing nodes of a large distributed, parallel processing system for sonar signal processing—a distributed, parallel sonar array (DPSA). This approach holds the potential to eliminate the need for a centralized data collector/processor, reduce the aggregate battery drain, and increase overall system performance, dependability, and versatility.

The technical issues involved with the design and development of parallel and distributed computing architectures and algorithms for fault-tolerant sonar arrays include unsolved problems in four major areas. These areas are: topology, architecture, and protocol; decomposition, partitioning, and mapping of beamforming algorithms; hardware fault tolerance for node and network components and subsystems; and cluster-based simulator tools. Techniques and mechanisms under development for this effort include algorithms and performance models for the decomposition and mapping of both time- and frequency-domain beamforming algorithms to linear array and ring multicomputer architectures for sonar arrays, an integrated, cluster-based, fine-grain system simulator, a small-scale hardware prototype, and a prototype software system capable of running on both the cluster-based simulator and the hardware prototype.

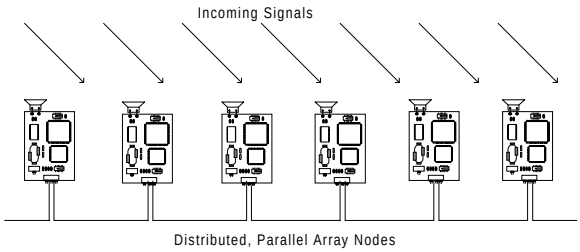

Figure 1 : Distributed, Parallel Sonar Array

The distributed, parallel sonar array differs from current sonar arrays in that rather than relying on a front-end processor to do the digital signal processing, the array nodes themselves distribute and together perform the processing.

There are three thrusts in this ongoing project that must be merged to achieve an efficient and effective DPSA system. These thrusts include network and node architecture modeling, simulation, and analysis; parallel time-domain beamforming; and parallel

[†] Dr. Rosen is with the Department of Electrical and Computer Engineering at Drexel University in Philadelphia, PA.

frequency-domain beamforming. The network research is aimed at modeling topologies to be embedded in a linear sonar array (shown in Figure 1), such as unidirectional array, ring, and bidirectional array. The parallel beamforming algorithms are designed for distributed sonar signal processing over such a coarse- or medium-grain network.

This paper highlights ongoing research results in the development of parallel and sequential algorithms for use on the DPSA. A number of fine-grain network models have been developed as candidates for the DPSA network architecture, and an overview of these networks is provided in Section 2. A series of fundamental time-domain beamforming algorithms have been designed and implemented both sequentially and via several methods of parallelization. An overview of these algorithms, implementations, and their performance results are provided in Section 3. Similarly, a series of fundamental frequency-domain beamforming algorithms have been designed, both sequential and parallel, and an overview of these and their performance is included in Section 4. In Section 5 the basic structure by which rapid virtual prototyping of this DPSA system will be developed is briefly presented. When complete, this simulator will integrate fine-grain models of network architectures, node architectures, and parallel software systems in a manner that provides high-fidelity analysis of performance and power issues associated with this system. Finally, in Section 6, a number of conclusions and topics of future research are presented.

2. Candidate Network Architectures

Three network architectures have selected for consideration on the DPSA system: a unidirectional linear array (the baseline for this system), a bidirectional linear array, and a unidirectional register-insertion ring. A fine-grain network model has been developed and verified for each. The criteria for the final selection of the network architecture will include functionality, efficiency, power requirements, fault-tolerance, and circuit complexity.

2.1 Baseline Unidirectional Array

A unidirectional point-to-point network is an array in which each node, excluding the first and last, has one incoming connection and one outgoing connection. The stream is unidirectional and thus limits the protocol in many ways. The first limitation is that for a network of this structure to work, the traffic must be absolutely deterministic. The topology also limits the communication ability, making it impossible to construct an acknowledgement-based protocol.

The fine-grain model for the RDSA baseline was designed, developed, and verified using the Block-Oriented Network Simulator (BONeS), a fine-grain network modeling and simulation tool available from the Alta Group [ALTA94]. The furthest node upstream becomes the master node and creates a large packet of "empty boxcars" so that each node has room to squeeze its particular data into the “freight train” as it passes. This is the basic protocol structure used in the U.S. Navy Rapidly-Deployable Sonar Array (RDSA) baseline architecture. With this protocol, any node could potentially become master for fault-tolerant measures in the event an upstream node or link failed. As master, the node is in charge of setting the network speed and creating the freight trains. The medium access control (MAC) for this protocol is analogous to a freight train picking up passengers as it passes each station. This is shown below as Figure 2. The feasibility of the unidirectional approach for this application is limited, since it clearly cannot support robust algorithms which require acknowledged services, retransmission, or upstream transmission.

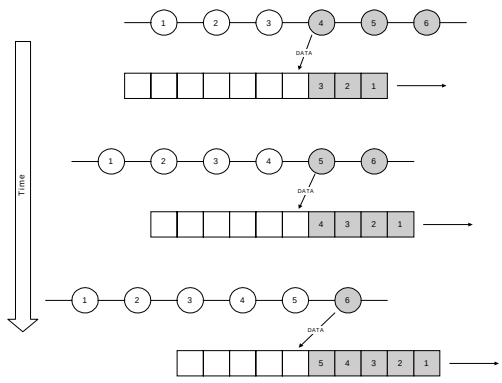

Figure 2 : Unidirectional Linear Array

The protocol essentially reserves space for each upstream node to place its data into a “boxcar” of the passing train. Thus the data is emptied onto the train as it passes.

2.2 Insertion Ring

Register insertion is named after its most distinctive trait, the insertion buffer. As shown in Figure 3, three buffers typify the MAC protocol -- the insertion, input, and output buffers. The output multiplexer switches the output path between the insertion buffer and the output buffer. The input switch routes packets to the insertion buffer or the input buffer if the destination field matches that of

the node. The theoretical maximum instantaneous throughput of the insertion ring is equal to the network rate, *R*, times the number of nodes, *N*, of the network. This peak represents the purely determinate case in which each node sends data to its immediate neighbor downstream. The traffic which is intended for the network may be designed in such a manner as to take advantage of the network's theoretical peak throughput of *N*R*.

The basic theory of an insertion ring is that every node has equal probability of writing to the network, and perhaps its greatest weakness is its poor fault-tolerant nature. Any break in the ring or a bad node destroys the state of the system. However, it may be argued that it is no less fault-tolerant than a unidirectional array since a broken ring could theoretically be reinitialized as a unidirectional array or otherwise reconfigured in the same fashion. But, the overhead to ensure this degree of fault-tolerance would be significant, since not only would the network protocol need to be adapted in this scenario but also the application would have to be able to support low connectivity or limited communication.

The register-insertion ring was found to have unacceptable blocking when random traffic was injected in the application layer. However, this is not a problem if the traffic is deterministic, an assumption which simplifies the model. Because of this fact, it is presumed that each node will offer approximately an equal amount of traffic. This result was also used in the design of the bidirectional array.

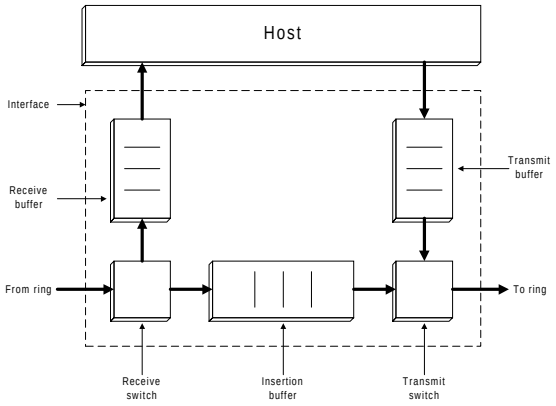

Figure 3 : Block Diagram of Insertion Ring Node [HAMM88]

The MAC layer of the insertion ring is composed of three buffers: the output, input, and insertion.

2.3 Bidirectional Linear Array

The bidirectional linear array also uses the insertion buffer scheme. The bidirectional array could be connected by the use of two discrete paths, or to save cabling costs the signals could be multiplexed and share the same link. Either way, a node cannot avoid the use of two I/O ports, one for traffic in each direction. Each layer is design in a symmetric fashion with respect to each direction up until the network layer.

The MAC was constructed of two insertion buffers and two output buffers. The protocol can be used as a dual-concentric insertion ring or a bidirectional array. The MAC was built by mirroring all of the buffers in the register insertion MAC except the input buffer. Only one input buffer is required and may be doubled in size if necessary to compensate for the doubling of the other buffers. As mentioned with register-insertion rings, each node has random contention in placing traffic on the network. Stochastic traffic patterns need some management protocol when the system is nearing saturation, however, the traffic intended for this network is deterministic so no management protocol is needed for fairness in the MAC layer.

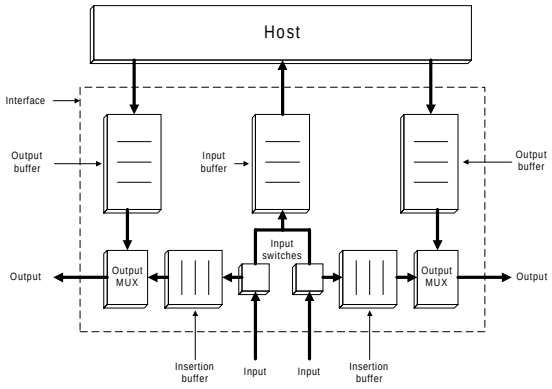

Figure 4 : Bidirectional Array Node

The MAC layer of the bidirectional array is essentially modeled as two insertion ring nodes. The two input streams share a common input buffer.

As shown in Figure 4, the protocol requires two sets (or a symmetric pair) of many functions which illustrates one of the bidirectional array's greatest faults -- an increase in hardware complexity, cost, and power consumption. However, the bidirectional array has the distinction of being the most fault-tolerant. If a part of the network goes down, the rest of the network can reinitialize as a fully-functional array. Therefore, it is a much better alternative to the ring and much better than the unidirectional array because of additional advantages for support of algorithmic complexity and flexibility.

The details associated with the fine-grain modeling, simulation, and analysis of these three network architectures are outside the scope of this paper. However, each of the systems has thus far proven able to support the traffic offered to it, although the bidirectional array slightly outperformed the register insertion ring when fed with traffic patterns sampled from parallel beamforming algorithms. The register-insertion ring and bidirectional array are clearly capable of handling robust beamform algorithms of many orders more complex than those used in these simulations.

Performance aside, the bidirectional array, although more costly in terms of price and power, thus far promises to be the best alternative for a parallel and distributed sonar array. Results indicate that it clearly handles traffic better than the other topologies in addition to having a natural advantage for fault tolerance.

3. Time-Domain Delay-and-Sum Beamforming with Interpolation

The first sub-category in beamforming algorithms researched was the class of basic time-domain solutions. These algorithms are characterized by using techniques such as delay-and-sum, filter-and-sum and autocorrelation are used to pass band filter data arriving in a particular direction [JOHN93, NIEL91].

3.1 Delay-and-Sum Decomposition

This section presents the methods used in the preliminary parallel decomposition of the time-domain beamformer. Specific consideration is given to the topology for which the algorithm is targeted, such as unidirectional array, ring, or bidirectional array.

3.1.1 Sequential Delay-and-Sum with Interpolation Algorithm

The beam power pattern of the sonar array is found for several steering directions using the delay-and-sum with interpolation (DSI) algorithm. This method is computationally equivalent to detecting sources across the search region for several steered beams. A flowchart for the sequential beam pattern algorithm is given in Figure 5. Each block is labeled with a number for future references to the diagram. The first step in the sequential DSI is to initialize the array and incoming signal specifications, and calculate the number of possible steering directions based on the improved resolution offered by the interpolation factor. The coefficients for the lowpass filter required for interpolation are then calculated.

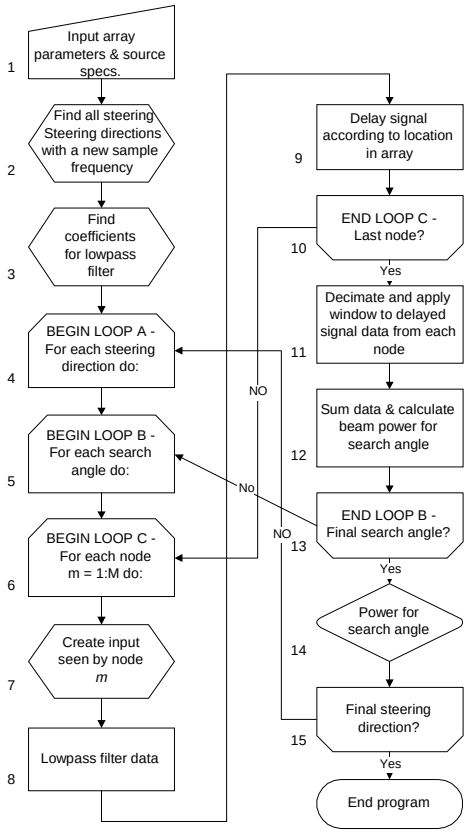

Figure 5: Delay-and-Sum with Interpolation Sequential Algorithm

This flow chart shows the execution path for the delay and sum with interpolation sequential beamforming algorithm.

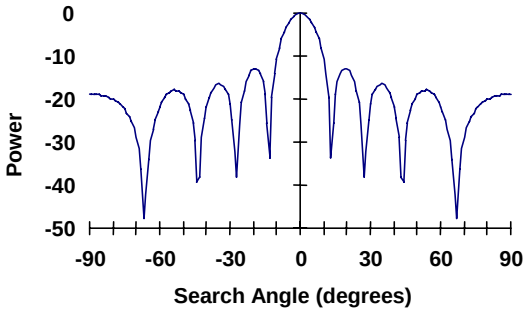

Figure 6: Expected Beam Pattern Output

This is the expected beam pattern output of a simple linear array steered to 0 degrees.

The next step is to choose a particular steering direction and calculate the appropriate delay for each node. To measure the beam pattern for the array for a particular steering direction, a source is placed at a number of different angles and the output power of

the array is calculated for each input source. A beam pattern is calculated for many different steering directions. Figure 6 shows a typical output from the sequential algorithm for an array steered to zero degrees.

In the flowchart of Figure 5, the outermost loop is the number of steering directions, the middle loop is for each incoming signal, and the innermost loop is to calculate the data at each node for the incoming signal when seen from a particular steering direction.

After filtering the data, the signal at each node is delayed an amount relative to its position on the array. A window function is applied and the data is summed and squared to form the beam power for that particular input signal. The process for a single beam pattern continues until all specified incoming angles are exhausted. Then the pattern begins again for a new steering direction and a new beam pattern.

3.1.2 Parallel Delay-and-Sum with Interpolation Algorithms

Perhaps the most intuitive partitioning method of the DSI algorithm from the flowchart is functional decomposition. In Figure 5, blocks 1, 2, 3, 7, 8, 9, 11, 12, and 14 are considered the tasks. A functional breakdown of this algorithm imitates a pipeline structure, where each process is responsible for a single task. The data is streamed through the tasks until the algorithm is complete. Separating this algorithm into 9 tasks is a fine- to medium-grained functional approach.

By contrast, because each node in the sonar array must collect its own data, this application naturally involves a high degree of data parallelism which naturally implies a domain decomposition approach. The data is divided among the nodes and each set of data is processed in the same way. At the end, the data is collected into a single entity and processed, and the result is given.

Another interesting approach to partitioning this algorithm is a combination of the functional and domain decomposition strategies. Figure 7 illustrates this concept. While the unrolling of the loops provides for domain decomposition, the interior of the loops where each task is performed by a different process (i.e. blocks 7, 8, and 9) represents a functional decomposition.

The DPSA system is itself a message-passing multicomputer with either a unidirectional, bidirectional, or ring interconnection network. In the partitioned algorithm, the collection of data at the end all flows into a single process. For the unidirectional and ring topologies, this is a problem. The data can only flow in a single direction, thus restricting which process can perform this task. In addition, care must be taken when passing data to the next process so that its input queue does not become overloaded with data. One way to reduce the chance for queue overflow is to perform the loop for each steering direction sequentially instead of in parallel. This subtracts one level of parallelism, but reduces the buildup of data along the array. Another way to reduce data flow is to perform the summing as the data is moved along the array instead of at the very end of the algorithm.

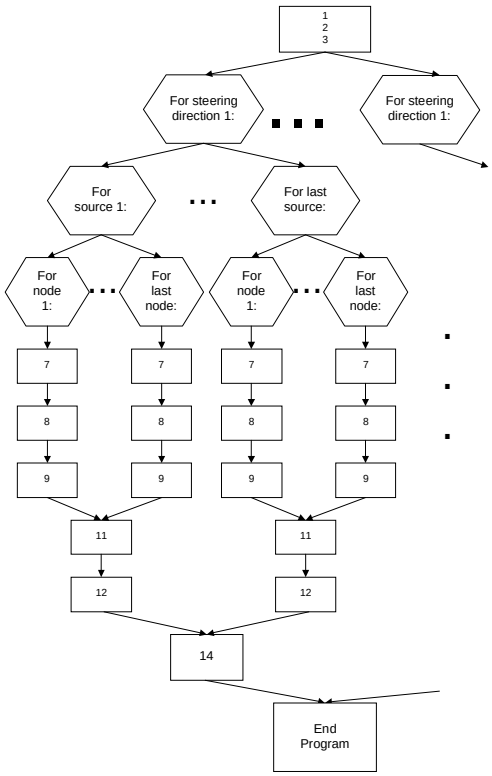

Figure 7 : Combining Domain and Functional Decomposition

Unrolling the loops represents domain decomposition, and the interior of the loops represents functional decomposition.

The unidirectional array is the simplest to implement of the three interconnection networks because the virtual architecture of the array remains static. Only one end process is ever responsible for the final result, and that process is labeled the "boss." If a break occurs in the array, the new end process takes over, thus giving the array fault tolerance. Figure 8 represents the unidirectional algorithm. The first time through, each process is waiting for data from the previous node before it begins any calculations. In this way, the algorithm resembles a pipeline. After each process has received some data, all processes are busy until the next steering direction is encountered, at which time they synchronize. This

is necessary to keep a build up in the data at nodes further down the line.

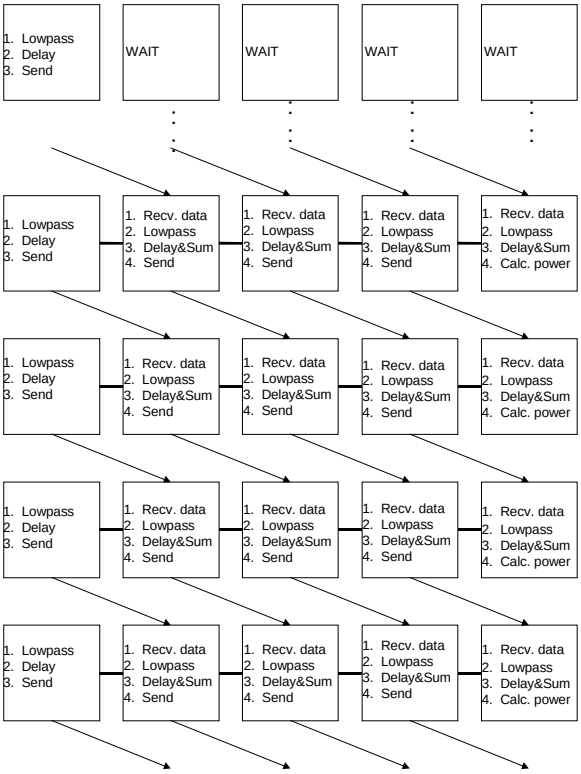

Figure 8 : Unidirectional DSI Algorithm

As can be seen by the uniform direction of the arrows, this flow chart depicts the unidirectional delay and sum with interpolation.

The ring topology is similar to the unidirectional array in that it is also unidirectional. In fact, the ring can be viewed as a modification to the unidirectional array. Unlike the previous algorithm, the final process (i.e. the boss) that calculates the power does not remain the same node. The responsibility of collecting the final data floats around the ring to different processes on different nodes. If the current boss is process 1, where the ring is configured such that the numbers increase clockwise and the direction is clockwise, then the next boss is process 0. In this way, while the boss is calculating the beam power for one angle, the previous node can set up as the next boss and begin to receive data.

The bidirectional array is somewhat different from the ring and unidirectional. Since the bidirectional array can communicate in both directions, we can assign the middle node to be the boss. If there is an even number of nodes, then the node closest to the transmitter (i.e. closest to node 0) becomes the boss. All nodes to the left of the boss send data to the right, and all nodes to the right of the boss send data left. This algorithm takes less time to process the data because data is being sent in two directions at once and only half as far. Thus, more results are available more quickly. Figure 9 illustrates the bidirectional array algorithm.

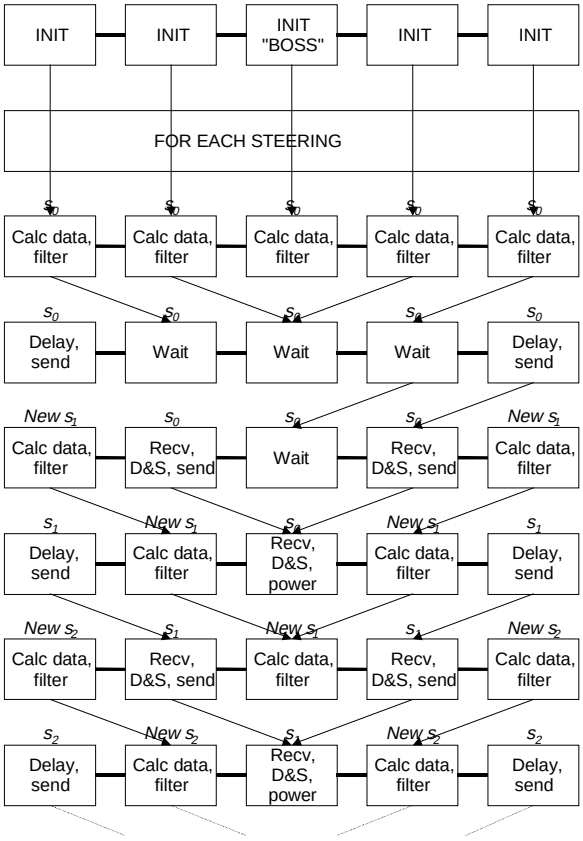

Figure 9 : Bidirectional DSI Algorithm

The s_n variables represent the incoming angles. Downward motion represents increasing time. Each column represents one node. The boss is responsible for calculating the power and beam pattern.

3.2 Delay-and-Sum with Interpolation Implementation

Preliminary results indicate that the domain decomposition technique provides a better parallel partitioning approach in that it allows for better fault tolerance, more naturally fits the algorithm, and requires less communication among the processors. This section implements this time-domain technique to help quantify the performance of the decomposed algorithm over a distributed array, the speedup of parallelizing this conventional beamforming technique, and the performance benefits when executed over different communication networks. In the sonar array, each of these factors must be addressed in order to determine the usefulness of parallel processing on the sonar array. In order to

show the performance improvement of the proposed parallel processing techniques, a baseline system was also simulated which implements a sonar array with an end processor. The following sections detail the simulation of the DSI algorithm over the baseline and in parallel over a unidirectional topology, a ring topology, and a bidirectional topology.

The results provided represent a preliminary investigation into the parallelizability of time-domain beamforming algorithms. The primary focus of the time-domain research was the application of standard parallel programming techniques to a distributed processing sonar array. Conversely, the optimal execution of multithreaded algorithms was the goal of the frequency domain implementations presented in Section 4 but the results of both endeavors are of equal importance. For this reason, the time-domain implementations use a simple single-threaded execution and communication model as thus does not as yet support a complex simulation environment.

3.2.1 General Implementation for the Delay-and-Sum with Interpolation Algorithms

To simulate the sonar array, each array node was implemented as a single process on a workstation. As a result, the number of nodes comprising the sonar array was limited to the number of workstations when using MPI [MPIF93] over native SCI (i.e. MPI/SCI [PARA95]) or to the number of processors using MPI over TCP/IP over Ethernet or ATM (i.e. MPICH [GROP]). By allowing only a single process per processor, the simulation implementation was single-threaded and simpler. All interworkstation communication was handled by the MPI implementation specifically designed for the network in use. All experiments and timing runs were conducted on a cluster testbed of eight 85-MHz, dual-processor, SPARCstation-20 workstations running Solaris 2.5.1 and connected by 10-Mbps Ethernet, 155-Mbps/link Asynchronous Transfer Mode (ATM) [GORA95], and 1-Gbps/link Scalable Coherent Interface (SCI) [SCI93], as shown in Figure 10.

For MPICH on Ethernet and ATM, the algorithm was able to simulate up to fifteen array nodes on these eight dual-processor workstations. The processes were mapped in a round-robin fashion on the workstations. For between three and eight array nodes, each process was mapped directly to a workstation. For more than eight array nodes, each workstation was assigned another process until the number of sonar nodes was exhausted. In this case, the second processor in the workstations is used in parallel but the network interface is shared between the two processes. A maximum of eight nodes was used for the SCI runs. Because only MPI is used for all communications during the time-domain simulations, the timing results can accurately compare the different computer networks used for simulation.

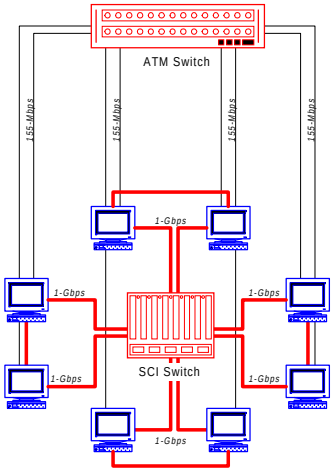

Figure 10 : Experimental Cluster Testbed

Each workstation is a dual-CPU, symmetric multiprocessing SPARCstation-20 machine connected by Ethernet, ATM, and SCI via the Sbus.

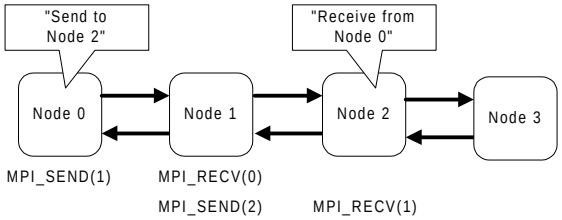

Figure 11 : MPI Simulation of Network Architectures

This figure illustrates how MPI communications are constrained to use only those types of communications that are available in the target network architecture.

Defining and enforcing proper internode communication rules allowed the simulation of multiple network topologies. For example, the array nodes in a unidirectional topology were limited to communicating only with their immediate downstream neighbor. In the ring topology the most downstream node is also capable on sending data to the most upstream node. In the bidirectional array, each sonar array node is capable of directly communicating with its immediate neighbor. Figure 11 shows an example of this constrained communication to simulate a network architecture.

3.2.2 Baseline Specification: Sequential Unidirectional DSI Beamformer

The array configuration used as the baseline for comparisons is a number of nodes linearly connected together that simply collect data and send that data to the next immediate node. The data is sent via the boxcar network as described in Section 2.1. A single end processor eventually gathers all data where it is processed. Each sonar node in the array was executed as a separate process that communicated with other nodes using MPI. Each process was responsible for reading the incoming boxcar, applying a window function to the nodes local data, inserting this data into the boxcar, and sending the boxcar to the next node. Finally, the boxcar packet arrives at the end processor, where the sequential program on a single workstation is executed on the data. This baseline provides the point of comparison for all proposed parallelization of the array calculations.

All speedup numbers calculated for the parallel programs in the following subsections were made by comparison with the "like" interconnect. For example, all ATM parallel programs are compared with the ATM baseline and all SCI parallel programs are compared with the SCI baseline. On the horizontal axis, each group shows the values for a particular program, and bars within a group show how many processors were used to obtain the value in that bar. The variable *L* shown in all of the following charts represents the interpolation factor. Larger values for the interpolation factor mean a larger problem size, and a number of experiments were conducted with *L* as an independent variable, however, only the results for the execution times and speedups for an interpolation value of eight are included in this section.

3.2.3 Parallel Unidirectional Delay-and-Sum with Interpolation Beamformer

In the parallel unidirectional time-domain program (PUT), each node was responsible for generating its own data. Figure 8 illustrated how this algorithm was implemented. Each block across a single row represents a process. The last node on the array, the highest ranked node, calculates its data and creates the vector that will be sent along the network by delaying its data the appropriate amount and sending it on. The next node then receives that data with an *MPI_Recv*, delays and sums its data with the vector being passed along, then sends it to the next node down the line. Eventually this reaches the first node, which then performs the beam power processing to obtain the beam pattern. Then the process begins again. The disadvantage of this technique is that even though some of the other nodes are taking the burden off the end processor, it is still this end processor that performs the most calculations and has the potential for causing a bottleneck. Besides the parallel processing, a major difference between this configuration and the baseline is that should the end processor fail in PUT, the next node can take over its responsibilities such that it now becomes the main node. All nodes have the computational capacity to assume this responsibility. If this happened in the baseline, the array would be rendered useless.

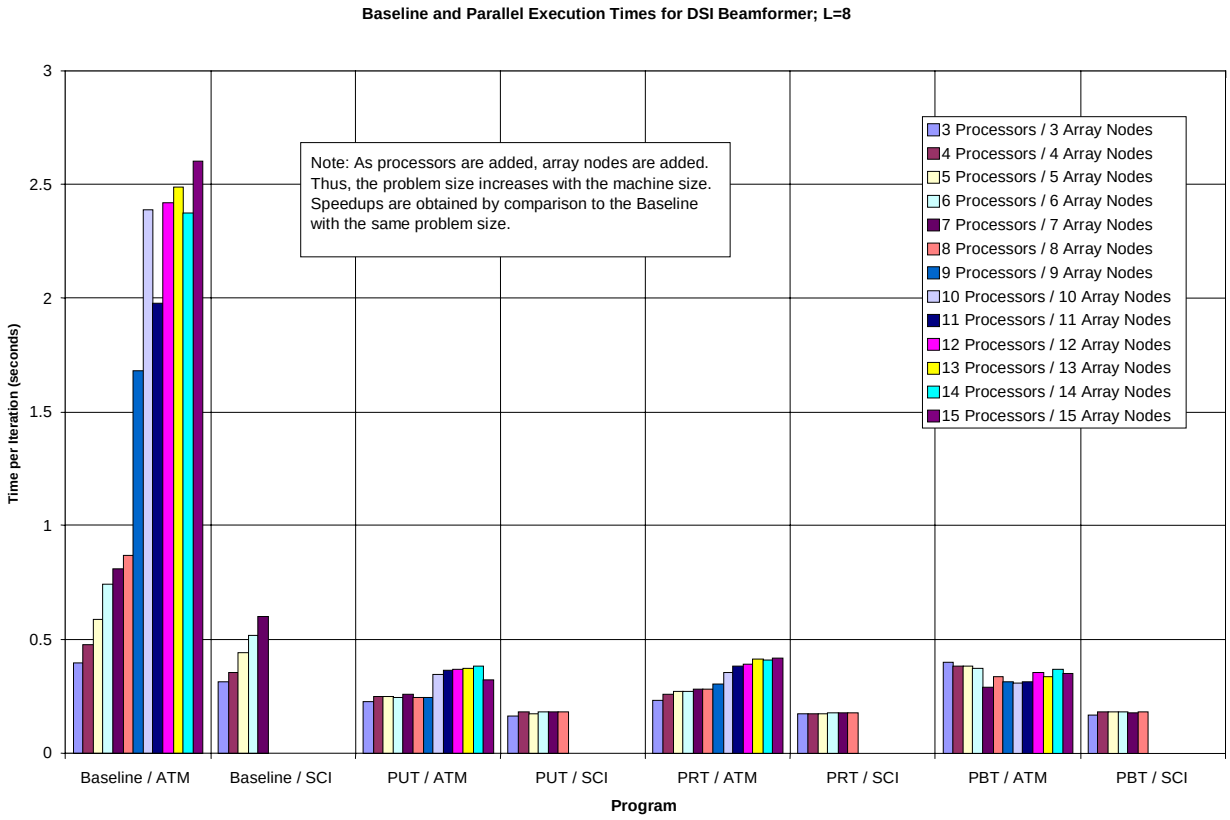

Figure 12 : Execution Times

ATM and SCI execution of all programs (parallel and baseline) for interpolation $L = 8$.

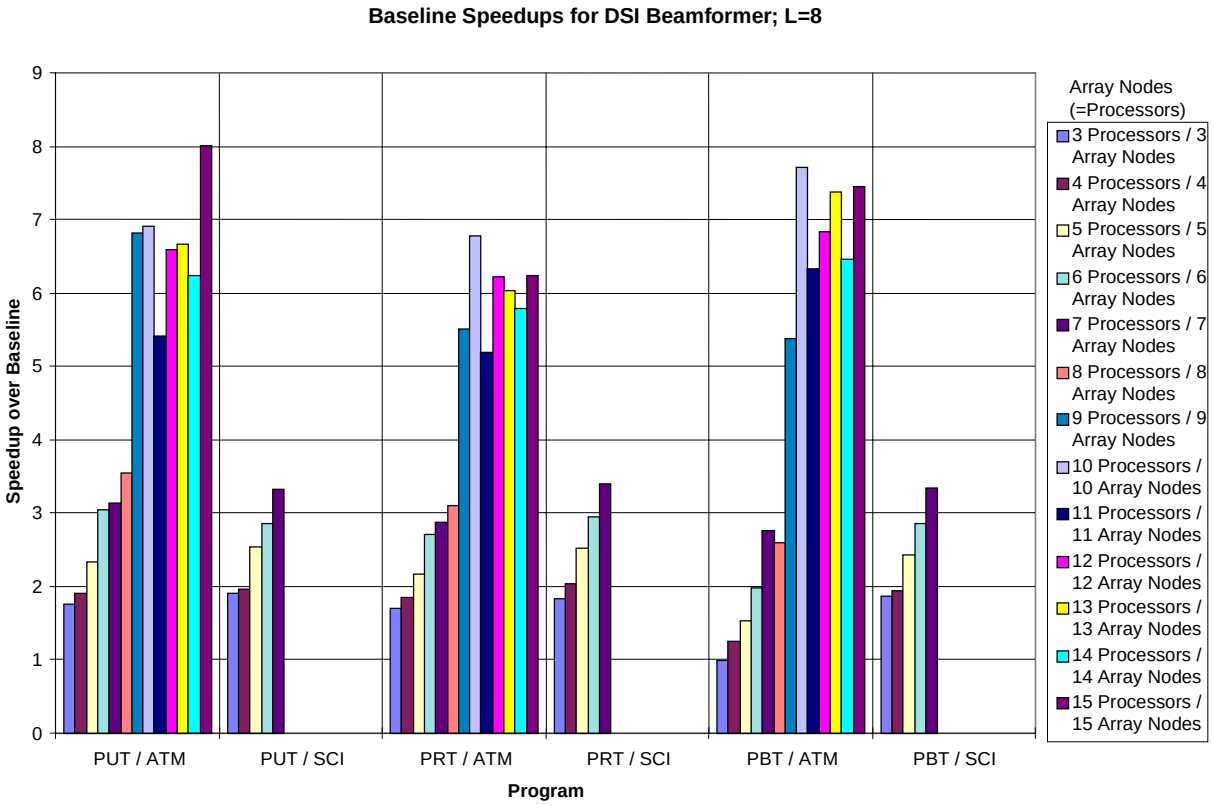

Figure 13 : Speedup vs. Baseline

ATM and SCI speedups of all the parallel programs over the Baseline (with "like" interconnect) for interpolation $L = 8$.

The decomposition methods employed spread the addition of the delayed data samples out over the nodes, thus lessening the amount of communication required by each node. This decrease in communication and decrease in amount of calculations performed by the head processor means that parallel speedup can be attained. The PUT-based parallel results groups (i.e. PUT/ATM and PUT/SCI) of Figure 12 and Figure 13 illustrate these speedups with the ATM-based cluster and the SCI-based cluster respectively, each versus the baseline architecture (i.e. unidirectional linear array with no distributed processing) with the same interconnect.

As expected, the speedup rises as the number of nodes increases. In addition, although not shown in these figures, the higher the interpolation factor, the better the speedup for most cases. This trend is a result of the increase in the number of calculations required for the interpolation and the ability of the parallel algorithm to spread out the increased computations over the various nodes.

3.2.4 Parallel Ring Delay-and-Sum with Interpolation Beamformer

The next configuration, parallel ring time-domain (PRT), is based on a revolving head processor. When comparing the baseline against the ring topology, the speedup once again is not expected to reach that of linear because the communication and topology setup still poses overhead that is inevitable. The PRT-based parallel results groups (i.e. PRT/ATM and PRT/SCI) of Figure 12 and Figure 13 illustrate these speedups with the ATM-based cluster and the SCI-based cluster respectively.

Generally, the speedup increases as the number of nodes is increased. In the ATM runs, most of the execution times for greater than eight nodes had a large variance. This condition could be a result of a number of factors, but mainly the simulation method. By assigning two processes to some of the workstations, not only do the processors share the execution of the processes, but the bandwidth of the network as seen by each process also effectively

decreases because it is now being shared by two array nodes instead of one.

3.2.5 Parallel Bidirectional Delay-and-Sum with Interpolation Beamformer

Finally, the PBT algorithm was implemented as described in Figure 9. Timings and speedup numbers were taken in the same manner as the ring program and are also presented in Figure 12 and Figure 13. These PBT bars show the similar performance of the PBT algorithm and the PRT algorithm. The speedup of PBT peaks at about three for seven processors over both SCI and ATM. The ATM runs maintained similar performance to the PRT as more nodes were added.

As can be seen in the execution time and speedup graphs, all the parallel algorithms scale to some extent. As more processors are added and the problem size increases, the speedup increases. This speedup is not linear with the number of, but the programs do perform considerably better than the baseline. No one algorithm stands out as a preferred method. The programs which took advantage of a fully connected, bounded-degree network (PBT and PRT) are capable of a higher degree of parallelism due to the fact that all nodes can be fully utilized because no node is trapped with a one-way communication.

3.2.6 Parallel Speedup Over Sequential

The previous discussion concentrated on the results and speedups obtained versus the baseline. This comparison is the kind of most importance for real-time beamforming applications. However, it is also important to make comparisons between these parallel programs and the purely sequential program, since this speedup relates more to centralized, off-line beamforming, where data is collected but processed at a later time by a conventional computer.

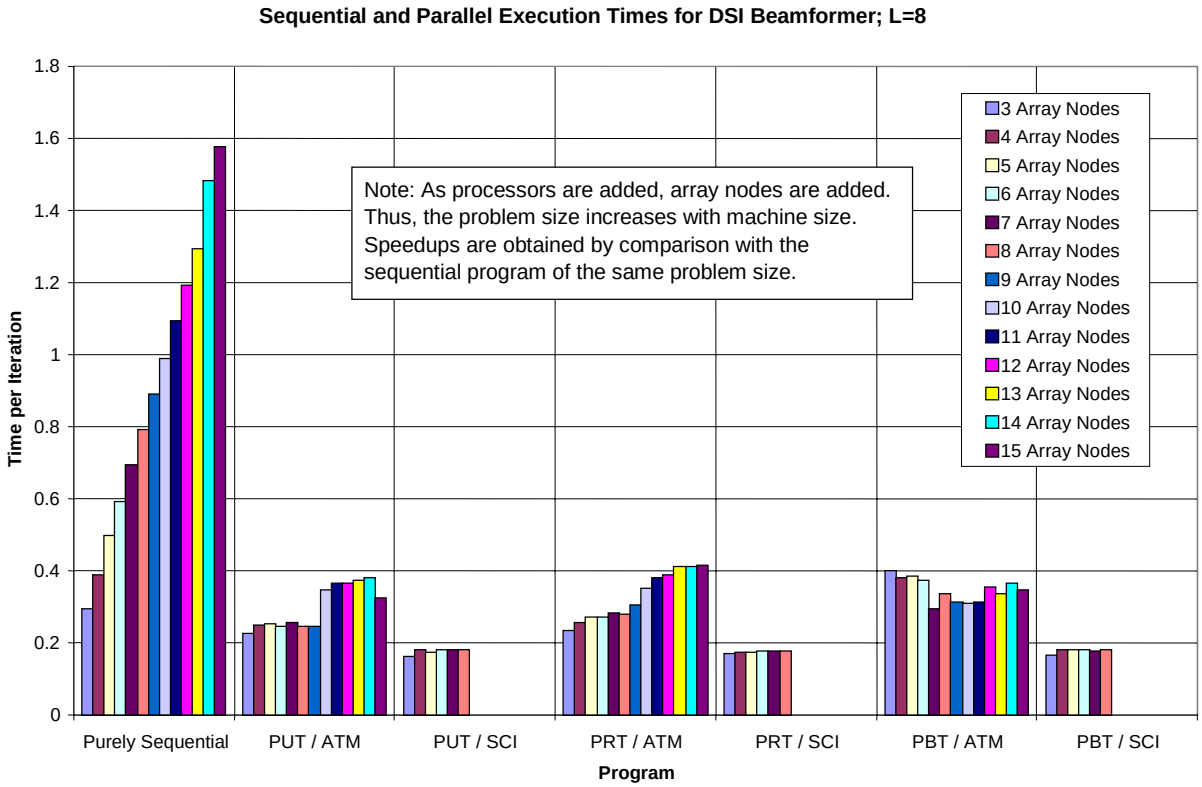

Figure 14 : Execution Times

Execution times of all programs (parallel and sequential) for interpolation $L = 8$. Each bar group represents a program. Each bar within a group indicates the number of array nodes, and hence the number of processors used.

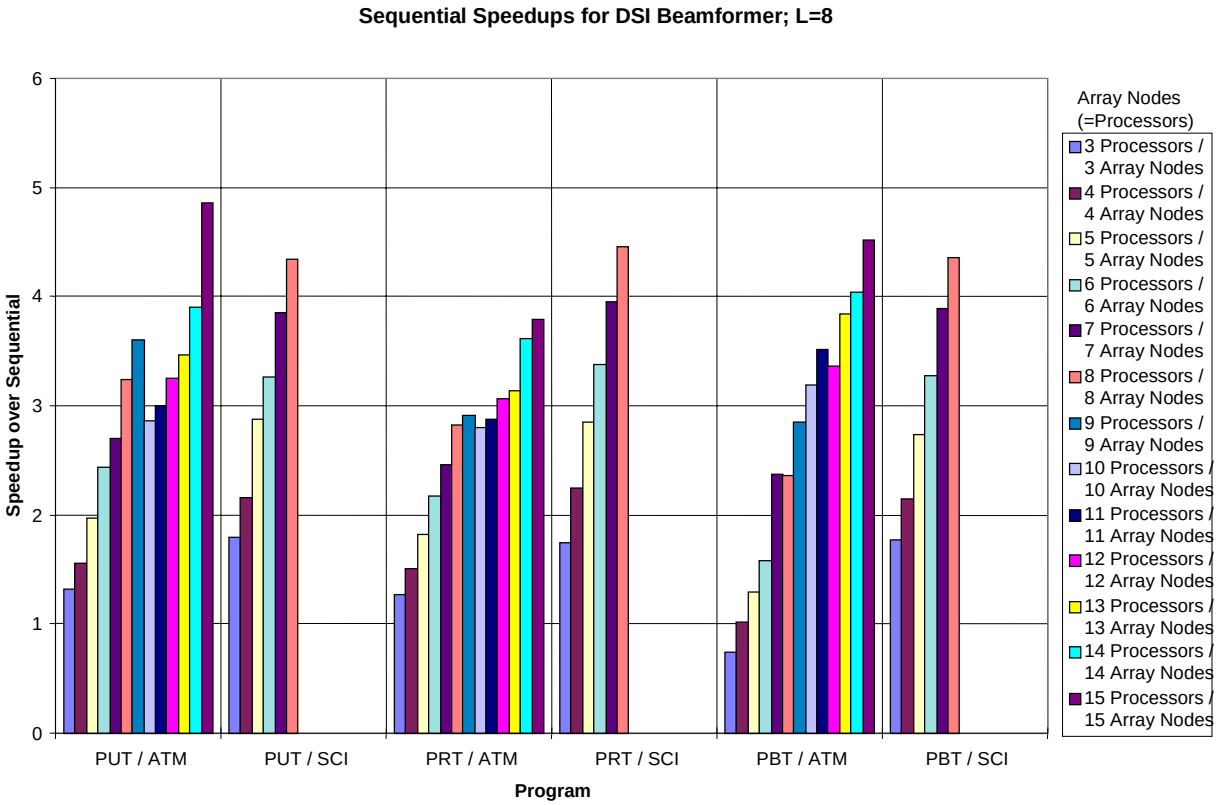

Figure 15 : Speedup vs. Pure Sequential

Speedups of all parallel programs versus the purely sequential program for interpolation $L = 8$. Each bar group represents a program. Each bar within a group indicates the number of array nodes, and hence the number of processors used.

The execution times for an interpolation value of eight are shown in Figure 14. On the horizontal axis, each group shows the values for a particular program, and bars within a group show how many processors were used to obtain the value in that bar. The speedups over the same number of array nodes are shown in Figure 15.

Looking at the speedup versus the pure sequential algorithm, it is clear that the parallel programs are still close to performing the same. Just as in the case of comparing to the baseline, no program stands out as a better performer, although the effect of the interconnect becomes more clear.

By looking only at the values for eight or fewer nodes, comparisons can be made between the ATM and SCI versions of the programs. In all cases, the SCI version performs better than the ATM version. For example, for eight nodes, PUT performs approximately 35% better when running over SCI. For PRF, this number is about 55%; and for PBF, it is 87%.

Since more time was taken in the study and decomposition of the time-domain algorithms, phenomenal speedups were not achieved in these preliminary implementations. It was shown that time-domain algorithms can be parallelized and that speedup can even be achieved using simplified, single-threaded simulation techniques. The lack of near-linear speedup of the delay-and-sum algorithm implemented does not, however, imply that linear speedup cannot be achieved, although the granularity of these algorithms makes them particular challenging on these candidate network architectures.

4. Frequency-Domain Beamforming

The Fast Fourier Transform (FFT) beamforming algorithm is the second algorithm presented in this paper for the DPSA. It is also used as the representative algorithm for several simulation methods, which include multithreaded programming and abstract message-passing communication.

4.1 Frequency-Domain Decomposition

The initial frequency-domain algorithm chosen to parallelize for the DPSA was a standard radix-2 FFT beamformer [SMIT95, HAMP93, NEIL91]. In this section, algorithms for sequential and several parallel FFT-based beamforming algorithms are presented.

4.1.1 Sequential FFT Algorithm

The first computation in each sample set in the FFT beamformer is to calculate the windowing functions for the various nodes and to multiply that node's data samples by that factor. Next, the algorithm takes the FFT of the windowed data, node by node. The algorithm then enters a loop that executes once for each steering direction. For each iteration of the loop, the algorithm multiplies the transformed data from each node by the node-dependent value for the steering factor. The next step is to sum all the data, sample by sample, for all 32 or 64 samples. The algorithm then inverse transforms this summed data and finds the magnitude, which gives the beamform result for the current steering direction. The algorithm stores this value before looping to the next steering direction. At the end of the algorithm, all the values obtained from the steering directions can be plotted versus steering direction so that the signal power can be seen spatially by the user. Furthermore, the FFT algorithm allows an additional independent variable (i.e. incoming frequency) and the associated results to be observed in a manner similar to Figure 16. The entire process is repeated for successive iterations with new sample sets. Figure 17 shows the flowchart for the sequential algorithm.

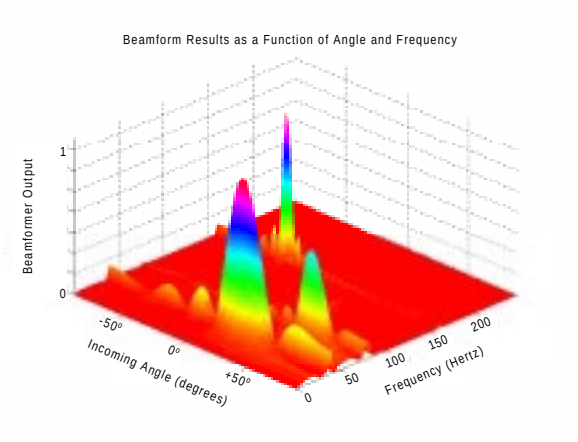

Figure 16 : FFT Beamformer Output

Along the two horizontal axes is the signal frequency and signal incoming direction.

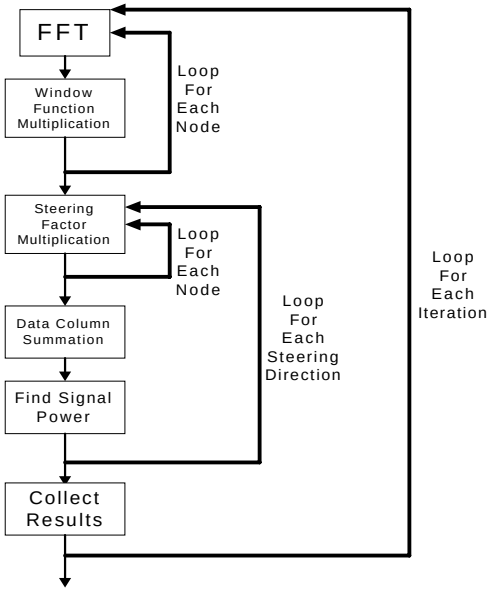

Figure 17 : Sequential FFT Beamforming Algorithm (*SEQFFT*)

The processor performs loops for each node and loops for each steering direction, all within a loop for each iteration.

4.1.2 Parallel Unidirectional FFT Algorithms

The parallel algorithm targeted to run on a unidirectional linear array is called the Parallel Unidirectional FFT, or *PUF*. The most downstream node in the array must always do the most work. This is because the algorithm requires a summation of all nodes' data. Since communication is only one-way, the only node capable of receiving data from all nodes in the most downstream node. This severely limits the degree of parallelism possible. Furthermore, the array is not evenly balanced as far as computational load is concerned.

Two versions of PUF were written. The paradigm used in the first version, *PUFv1*, is data parallelism; all transforms and all window-function multiplications can be done simultaneously. Next, the nodes send their data down the linear array to the first node which then carries out the remainder of the sequential algorithm. All receives by the front node must be made one after another. The nodes cannot combine their data ahead of time for the front node because the first node must multiply each column by node-dependent factors. Figure 18 shows the flowchart for how the parallelism in this algorithm takes place, where blocks lined up horizontally are computed simultaneously.

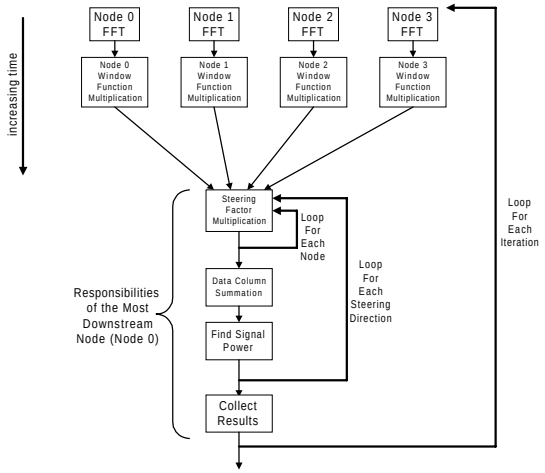

Figure 18 : Unidirectional *PUFv1 Algorithm*

This flow chart depicts how the parallelism in *PUFv1* is inserted in the FFT and windowing factor computations.

The second version, *PUFv2*, moves the steering direction loop from the first node and makes the operations in the loop part of the responsibilities of the several nodes. The major goal this accomplishes is to complete the summation as the communication is progressing. Rather than sending column after column to the front node, each node receives a column from upstream, adds its column to it, and sends the single summed column downstream. This approach is illustrated in Figure 19. A drawback in the new form of communication (with summation) is that it must be carried out for each steering direction.

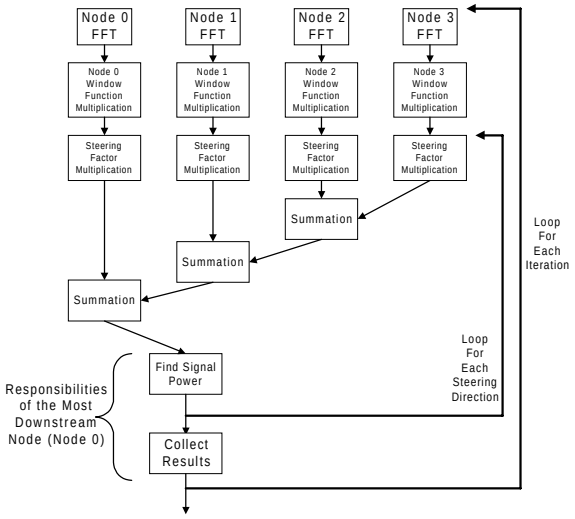

Figure 19 : Unidirectional *PUFv2 Algorithm*

The *PUFv2* flow chart demonstrates the lighter communication because each node sums data from the previous node so that it only has to send one column.

4.1.3 Parallel BDN FFT Algorithm

The next algorithm was developed for a fully-connected bounded-degree network (BDN), such as rings and bidirectional linear arrays The important concept introduced with this algorithm is that of a floating front-end processor, which allows for better link utilization and better processor computational usage. Rather than forcing all data to be sent to the first node every time, the data can be sent to a different node each iteration.

To assign which iterations go to which processes, this algorithm introduces a type of agenda parallelism, which shall be called here *specified agenda parallelism*. In the *specified* agenda parallel concept, the tasks which the working processes pull out of the agenda grab-bag are not random choices. Instead, a selection algorithm is created so that the iteration assignments are in a known, specified order. This formula simply follows a "round-robin" scheme in which the front-end for the next iteration is the neighboring node to the front-end for the current iteration.

The BDN algorithm takes the algorithm from *PUFv1* and implements the floating front-end. Once a front-end is chosen for a particular iteration, communication then proceeds down the ring or through the bidirectional linear array to that front-end just as it did in *PUFv1*. The front-end node then starts the loop for the several steering directions and puts together the results.

Pipelining in the iterations is employed. The nodes can collect another sample set from the audio transceivers and begin the second iteration before the first iteration is completed. At the beginning of the second iteration, the node doing the first iteration front-end work stops what it is doing for just enough time to FFT the second iteration data, multiply by the windowing factor, and send it off to the second-iteration front-end. Once the data has been sent to the new front-end, the node resumes front-end work for the previous iteration data. The process continues for additional iterations. As the iterations progress, more and more nodes are doing front-end work for their respective iteration numbers, all in different stages of completion.

4.2 Frequency-Domain Implementation

This section describes the implementation of the PUF, PRF, and PRB algorithms and performance results. These results include both execution times and speedup factors versus their baseline and purely sequential counterparts.

4.2.1 General Implementation for the FFT Algorithms

The algorithms were implemented by creating as many threads on the dual-CPU, SPARCstation-20 cluster testbed as the number of desired array nodes to simulate. The basic FFT code was adapted from [MORG94]. The threads are split evenly across the workstations. A main thread, called the parent, is in charge of setting up the simulation, reading parameters from the configuration file or the runtime system, and timing

All threads execute the same function, distinguished only by the passed input parameter indicating the thread's position in the array. The executed function hides the fact that each thread is not the same; nodes adjacent in the sonar array may sometimes be on the same workstation in the simulation and may sometimes be located on different workstations. In the case of intra-workstation threads (i.e. threads communicating locally between processors on the same workstation), any communication done between them is accomplished by two semaphore exchanges and a memory copy. The receiving thread will post a "ready-to-receive" semaphore when it is expecting another thread to send it something, and will then wait. The sending thread waits for the "ready-to-receive" semaphore and proceeds with the memory copy once it has found that semaphore. After finishing the copying, the sending thread posts a "send-is-done" semaphore, which is the indication to the receiving thread that it may continue execution knowing the communication was successfully completed. This process is shown in Figure 20. Discussion of Solaris semaphores may be found in [ROBB96]. For the case of interworkstation threads (i.e. threads communicating remotely between processors on different workstations), the Message-Passing Interface must be used to get information from one to the other. The sender calls *MPI_Send*, which returns only after the information is successfully communicated or at least successfully stored in the receiver's communication buffer. The receiver calls *MPI_Recv*, which blocks the process until the information is safely stored in the receiver's memory space.

A major achievement of the multithreaded simulation was the abstraction of communication so that the threads simply execute an opaque send function or receive function. The simulated threads need not be concerned about whether the thread with which it wants to communicate is on the same workstation or a different workstation. Furthermore, threads in an intraworkstation (i.e. local memory) communication do not worry about posting and waiting for the various necessary semaphores, and threads in an interworkstation (i.e. remote memory) communication do not worry about passing all the required parameters to the MPI function. With this communication abstraction, the threads simply call a generic send or receive function, specifying only the data size, the data location, the source thread, and the destination thread. There is an underlying layer which converts the abstract communication call into the necessary code.

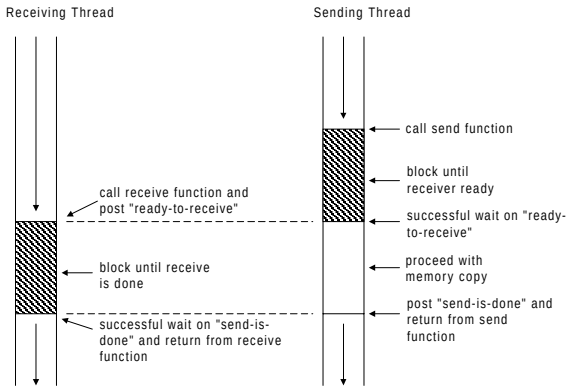

Figure 20 : Local Interprocessor Communication

This figure illustrates the threads of execution when semaphores are used for communication. Shaded areas indicate times the threads are blocked.

Another advantage of the multithreaded simulation is that it is possible to simulate more array nodes than there are workstations on the available testbeds. Multithreading allows this to be accomplished without changing the logical structure of the functions the nodes execute.

Another method used in the FFT algorithm implementations is the same rudimentary network simulation as was used in the time-domain programs. Simulating the complexity of a communication over different network architectures is implemented by using the communication calls over no more than one link at a time. For example, the implementation for a register-ring architecture is simulated by forcing each node to be able to only send to the thread immediately downstream rather than send to any thread in the array. This extra complexity in the communication results in overhead that is meant to reflect the latency in the underlying network architecture. Though not exact in the timing simulation for a particular architecture, this method is useful for general comparisons between different implementations of the same algorithm.

4.2.2 Baseline Specification

It is important to specify the baseline used for the parallel algorithms in the frequency-domain so that valid comparisons can be made. The method the baseline follows is that of a collection of dumb nodes whose only job is to collect data with an audio transceiver and send it forward. This communication is implemented by the method previously discussed in which the nodes send one link at a time in order to make a rudimentary simulation of the underlying unidirectional linear array architecture. The most upstream node sends its column of data (with no processing) down to the next node downstream. This node appends its data column to that of the upstream node and sends both columns downstream. The process continues until the front-end node has the entire matrix. The front-end processor of the baseline then performs all the calculations on the data as dictated by the sequential algorithm. The topology and the simulation model for the baseline are illustrated in Figure 21.

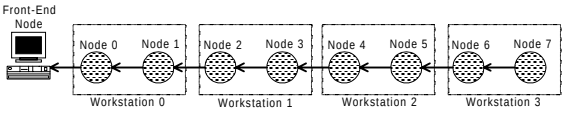

Figure 21 : Implementation Model for the Frequency-Domain Baseline

The model for implementing the algorithms is that several nodes (threads) are created across a distributed workstation testbed. One thread is created for the front end.

There are a few points to make about how the baseline performs. As more nodes (and thus threads) are added to the array, the execution time of the baseline increases. Furthermore, as more workstations are added with the same number of node threads, the execution time also slows. This trait is due to the extra interworkstation communication and fewer intraworkstation communications. Also of interest is how the baseline performance improves by using SCI instead of Ethernet, which is due to the better throughput and lower latency of the SCI interconnect versus ATM or Ethernet. The use of HCS Threads (an optimized multithreaded run-time library under development in the HCS Lab [GEOR97]) instead of Solaris threads also improves performance due to the fact that the HCS Threads remove much of the overhead of the Solaris threads, such as kernel context switches. All the upcoming parallel algorithms will be compared to the baseline which matches the interconnect and the thread library used in the parallel algorithm. For example, the timings for the baseline FFT over Ethernet using HCS Threads will be used in meaningful comparisons with the parallel ring algorithm over Ethernet using HCS Threads but not with the parallel ring over SCI or using Solaris threads. In this way, valid speedup comparisons can be made without needing to strictly quantify the time contributions from each factor.

4.2.3 Parallel Unidirectional FFT Beamformer

The first of the parallel algorithms implemented was the parallel unidirectional FFT, or PUF. The communication pattern for the first version of PUF, *PUFv1*, is not the same as the communication pattern for the baseline, but they are comparable. In the baseline program, if there are 32 samples taken at each node, then each column in the communicated matrix has 32 rows. The parallel unidirectional program differs because after the FFT, only 16 samples out of a 32-point FFT contain useful data. The other 16 values are simply mirror images of the first 16 samples, so the columns of the matrix only contain rows for the useful 16 values. Though *PUFv1* communicates fewer samples per column, each row in the column is now a complex number because the transform outputs complex numbers. In the implementations for this project, complex numbers are comprised of two single-precision, floating-point numbers. Therefore, baseline and *PUFv1* communications send the same number of floating point numbers, and any comparisons made between them are valid.

There is a granularity knob built into the *PUFv1* program that controls the size of the packets in the downstream communication. Decreasing the granularity knob means fewer columns are collected into each packet, and increasing the granularity knob results in more columns collecting into larger packets. The algorithm set to the coarsest-grain possible is the most efficient algorithm due to the overhead of collecting the various small packets in the finer-grained cases, so it is the coarse-grained results which are presented in this project.

Even with the granularity knob set to the coarsest possible grain, the algorithm's performance is not spectacular. In fact, it performs better than the baseline in very few cases. The SCI version generally gets speedups in the area of 1.2, but the Ethernet version does not get any speedup—only in the range 0.95 to 1.0. In fact, due to the inclusion of code to implement the granularity knob, *PUFv1* performs slightly worse than the baseline. Also, the communication time dominates the Ethernet version so that the parallelization of the FFT and window factor multiplication has hardly any effect. Due to the poor performance of *PUFv1*, it is used only as a stepping stone to the next unidirectional version and

the BDN version, thus the performance charts for *PUFv1* are omitted here.

Figure 22 shows the average iteration times of the parallel programs and the baseline programs. Figure 23 shows the speedup of all the parallel programs versus the baseline with the same interconnect.

The timing numbers for the second version of PUF, *PUFv2*, show that it provides a considerable improvement over *PUFv1*. Although performance on one workstation is not good, the performance improves as more processors are added. For a single workstation running this program, speedups over the baseline were in the range 0.28 to 0.6; however, the speedups over the baseline for eight workstations range from 1.4 to 7. This trend illustrates the scalability of the algorithm; as more processors are added, the algorithm will continue to improve performance; however, it is far from a linear speedup.

Also of interest is that for sixteen processors in eight workstations, the SCI version does not outperform the SCI baseline as much as the Ethernet version outperforms the Ethernet baseline. This illustrates how the communication costs were the major contributor to the Ethernet baseline as the number of workstations increased. Since the communication in *PUFv2* is much less than the communication in the baseline, the performance improvement is more noticeable over the slower interconnect.

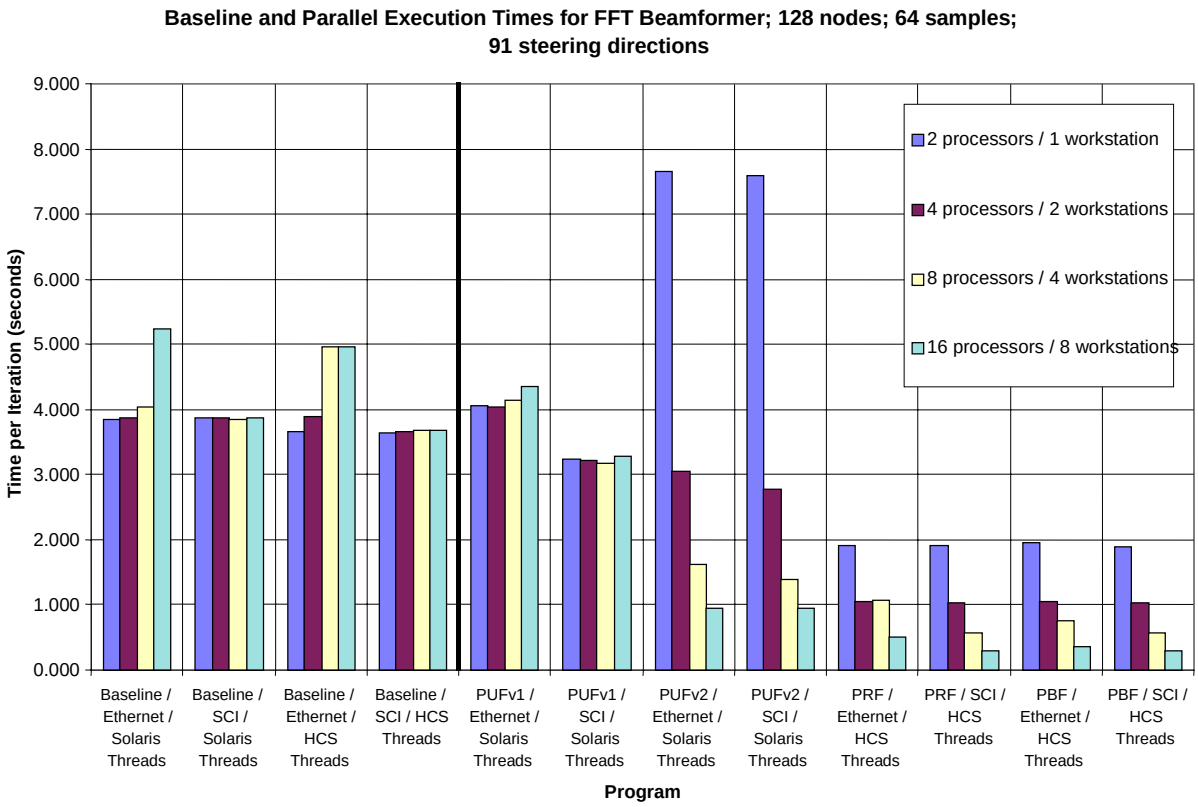

Figure 22 : Execution Times

Execution times of all parallel programs and of the 4 baselines (with different thread packages and different interconnects) for 128 nodes, 64 samples per FFT, and 91 steering directions.

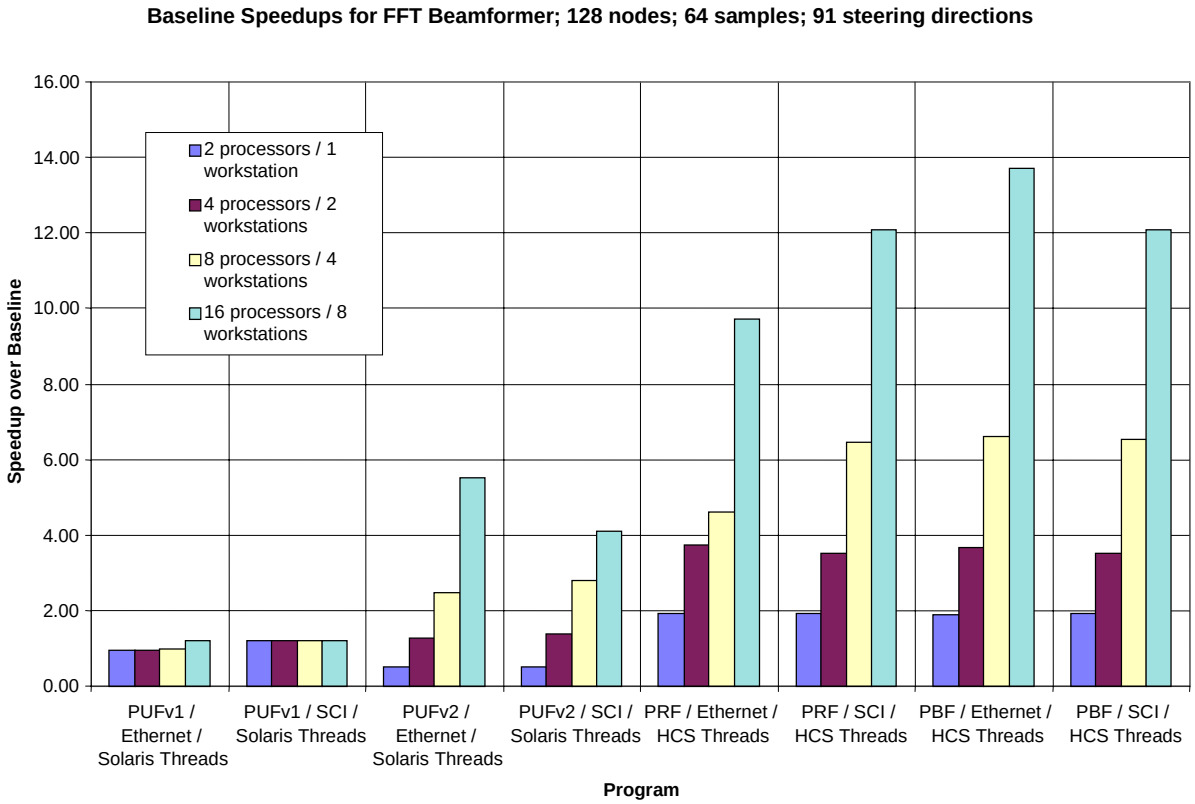

Figure 23 : Speedup vs. Baseline

Speedups of all parallel programs versus the "like" baseline for 128 nodes, 64 samples per FFT, and 91 steering directions.

4.2.4 Parallel Ring and Parallel Bidirectional FFT Beamformers

The first of the BDN programs timed is the parallel ring FFT. The performance of this program is considerably better than that of any of the previous algorithms. In fact, PRF scales very well and has a speedup curve approaching linear; that is, for 16 processors, the speedup gets up into the teens even for the worse cases, as opposed to the speedups for the parallel unidirectional FFT which stayed in single digits.

The last program in the frequency domain is the parallel bidirectional FFT (PBF). The figures show that PBF also performs very well against the baseline, just as did PRF.

Due to the fact that *PUFv2* does its summation before the communication, *PUFv2* outperforms *PUFv1* in cases where the problem size is large enough to offset the cost of putting the steering direction loop into the hands of the several nodes. Furthermore, the use of more processors helps *PUFv2* much more than *PUFv1* because of the increased parallelism in *PUFv2*. The ring (PRF) and bidirectional (PBF) programs considerably outperform the PUF programs simply due to the fact that the PRF and PBF algorithms are not constrained by the requirement that the first node do the bulk of all work. However, the performance of the ring and bidirectional algorithms does not improve at a rate as fast as *PUFv2* with the addition of more processors. This condition occurs since *PUFv2* uses a more efficient communication algorithm, and the ring and bidirectional programs use the growing freight train, which will take more time to communicate as more intraworkstation communications are replaced by interworkstation communications.

The comparison between the ring and bidirectional programs is not as decisive as comparisons with the PUF versions. Although the bidirectional shows performance improvements over the ring in some cases, the ring and bidirectional array algorithms do not clearly have a performance winner, since only the Ethernet version shows the performance increase in the bidirectional; the SCI

timings show no such improvement. Obviously, in a sonar array architecture that is not yet built, the true performance may be more like the Ethernet version or more like the SCI version. Furthermore, the software simulation in the case of these two algorithms cannot be declared as a perfect simulation of the true architecture. For example, even though the simulation will always have more overhead when it handles communication in two directions, the hardware may not have the same overhead. It will be imperative to simulate the code more precisely by using the BONeS/MPI interface currently under development in the lab.

The previous discussions emphasized the speedups of the various algorithms over the baseline. It was seen that the two versions of the parallel unidirectional FFT provided a slightly better performance than the baseline. The parallel ring algorithm and the parallel bidirectional algorithm provide considerable improvement over the baseline. Algorithms deemed suitable will need to be simulated with the upcoming BONeS/MPI interface so that exact timings may be made over a high-fidelity model of the sonar array networking hardware.

4.2.5 Parallel Speedup Over Sequential

The sequential, non-distributed form of the FFT-based beamforming program (previously referred to as *SEQFFT)* runs on a single processor on a single workstation. For the large problem size of 128 array nodes working with 64-point FFTs, the execution times are shown in Figure 24. All execution times are given as average-per-iteration values. The speedups for these same programs over the purely sequential is then given in Figure 25.

It is important to note that the parallel programs almost always perform better over SCI than they do over Ethernet, the exception being the case of one workstation where neither SCI nor Ethernet is used. The most glaring improvements provided by SCI show up in the eight-workstation situation because it includes the most communication over the testbed.

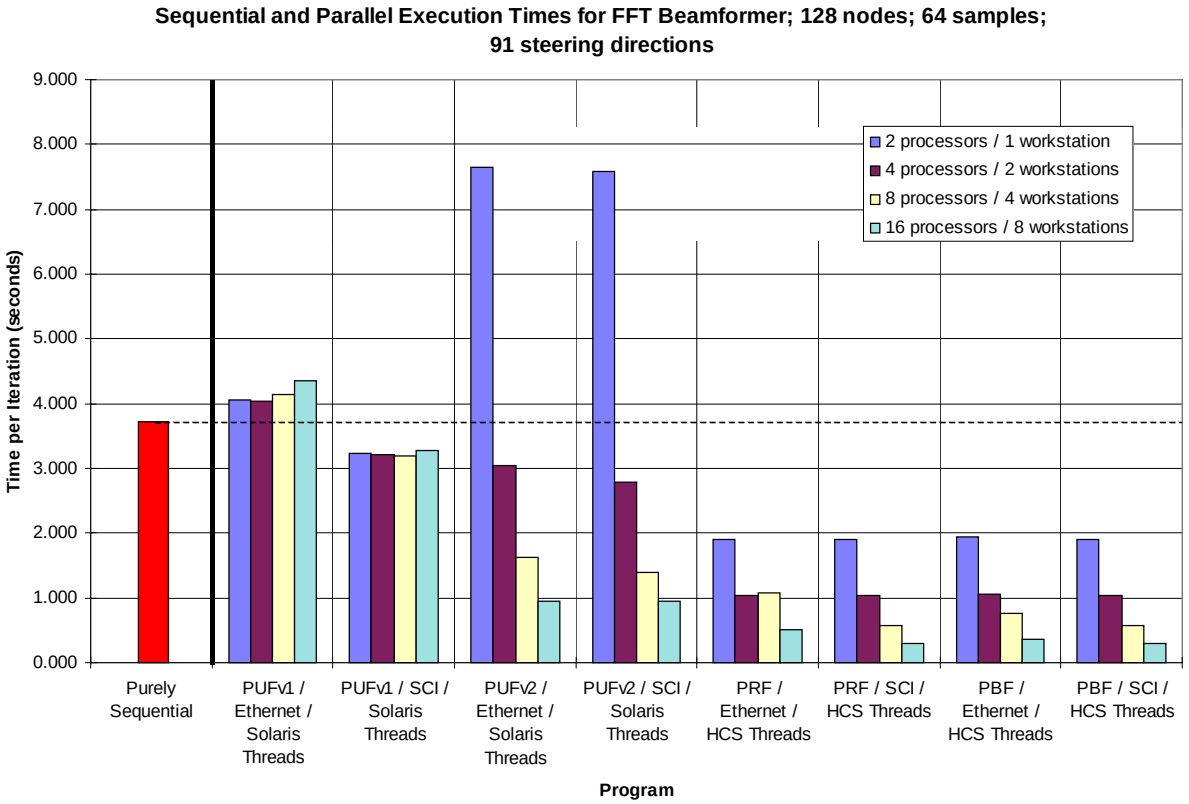

Figure 24 : Execution Times

Execution times of all parallel programs and the purely sequential program (SEQFFT) for 128 nodes, 64 samples per FFT, and 91 steering directions.

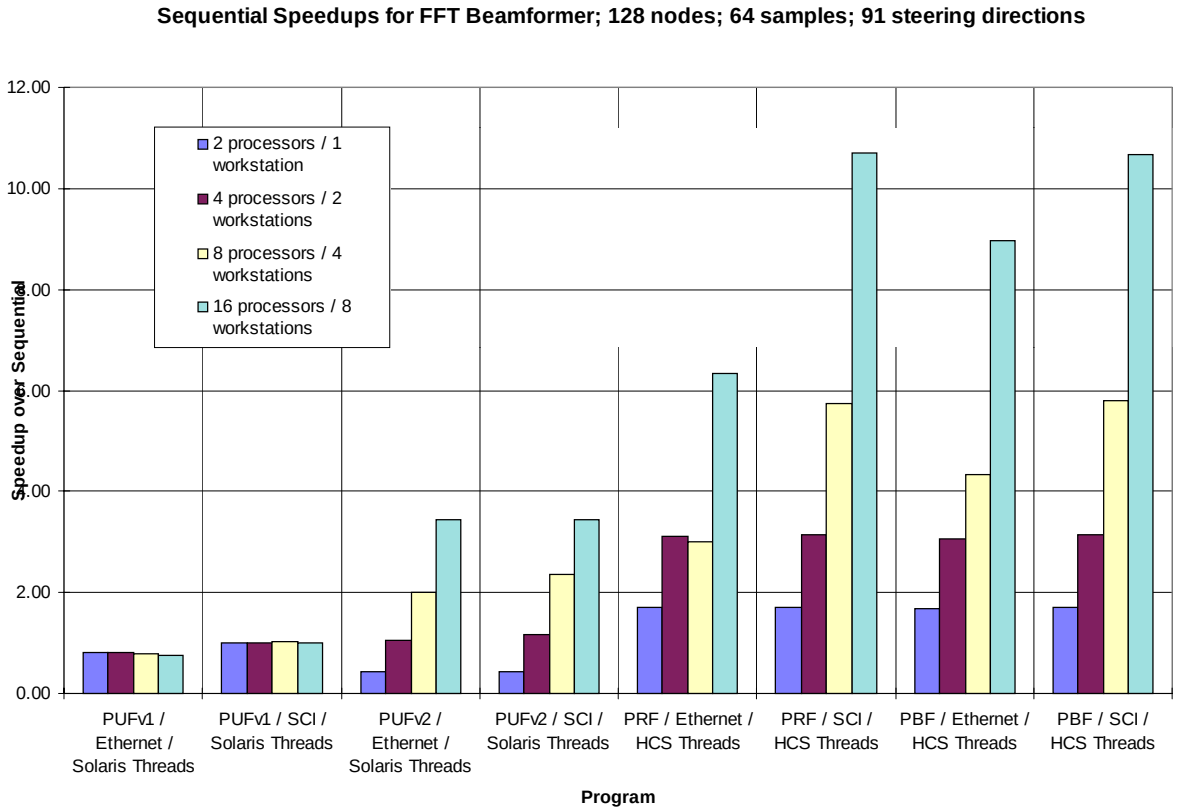

Figure 25 : Speedup vs. Pure Sequential

Speedups of all parallel programs versus the purely sequential program (SEQFFT) for 128 nodes, 64 samples per FFT, and 91 steering directions.

All the parallel programs except *PUFv1* provide some degree of scalability. This includes *PUFv2*, because even though it does not perform as well as the programs based on the fully connected network architecture, it does increase its performance as processors are added.

The speedups show that the Ethernet versions of PRF and PBF outperform the sequential program by factors of between 6.3 and 8.5 for sixteen processors in eight workstations. The SCI versions outperform the sequential program by factors of about 11 on the same number of processors. Also of note is that *PUFv2* has speedup of just under 4 in the large problem size, which is up from below 2 in the small problem size. This illustrates exactly how much the smaller amount of communication benefits *PUFv2* over *PUFv1*.

The last two FFT beamforming algorithms, the ring and the bidirectional linear array, clearly indicate increasing speedup versus the purely sequential algorithms, which is relevant for off-line, non-real-time beamforming systems, and versus the sonar array baseline, which is relevant for one-line, real-time beamforming applications.

5. Array Simulator

The final goal of the DPSA project is to prototype a distributed sonar array. Among the design choices in developing such a prototype are the network speed, network topology, algorithm decomposition method, and processor power. All these parameters must be combined to form a fine-grain prototype array simulator. Three simulation methods are needed in the development of the prototype. These include verification and validation of the network and processor models in BONeS; simulation and analysis of the parallel beamforming algorithms on the cluster of workstations; and an interface between the them to form the single, high-fidelity sonar array simulation, as shown in Figure 26.

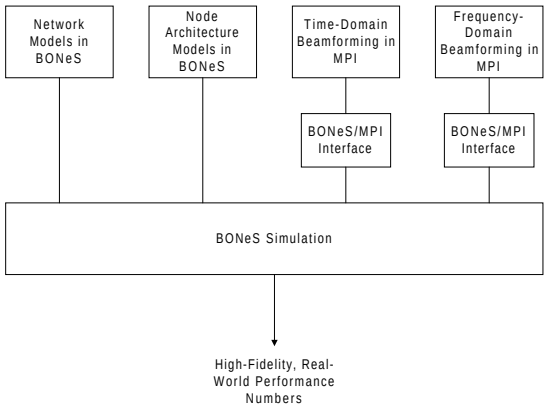

Figure 26 : BONeS/MPI Interface

This figure depicts the plan of attack for combining the myriad disciplines in this project.

The interface between BONeS and MPI programs is a promising task that will allow not only sonar arrays to be simulated, but also any parallel system having appropriate models in BONeS, and simulated in concert with actual parallel programs and their inplicit traffic patterns running over them. To accomplish this, the interface library replaces the MPI function libraries. The parallel program already coded will need no changes to be added to a BONeS simulation. The new MPI functions provided by this library act as the application layer for the lower layers of the BONeS model already developed. Furthermore, the libraries contain probes that collect detailed information on each communication, including number of bytes sent and the time step at which the communication occurred. Knowing the step-by-step information from the probes in the BONeS interface library, only the values for latency and throughput of the physical sonar array will be needed to construct a true-to-life representation of the timing of the final array.

With such development tools at a programmer's disposal, the track from beamforming algorithm to parallel prototype development is straight-forward. All aspects of the project come together to form a virtual prototype complete with highly-detailed performance analysis data.

6. Conclusions

This paper has presented ongoing research results in the development of parallel and sequential algorithms and network architectures for the advanced distributed, parallel sonar array (DPSA). A set of fine-grain network architecture models based on unidirectional linear array (i.e. the baseline), undirectional ring with register-insertion protocol, and bidirectional linear array has been completed, tested, and verified and a unidirectional ring with token-passing protocol is near completion. The models have and continue to be used for distributed and parallel performance studies, and are being expanded to include fault-injection capability for dependability experiments as well.

A number of time-domain and frequency-domain beamforming algorithms have been constructed. Preliminary parallel decompositions of several standard beamforming algorithms have been performed, including delay-and-sum with interpolation and FFT. Algorithms and programs for the sequential version of these beamforming techniques have been developed in sequential code to form a baseline by which true parallel algorithms and software are measured. Initial results from performance experiments with basic time-domain and frequency-domain parallel beamforming programs developed indicate the potential for near-linear speedup and a high degree of parallel efficiency when properly mapped to network architectures similar to those candidates under consideration. Preliminary tests have been successful in identifying and measuring the strengths and weaknesses of parallel programs being studied versus their baseline counterparts.

Given that all technical issues are driven by the network topology design, the network protocol design, and the node processor, the development of an efficient and effective architecture and set of protocols for this network-based multicomputer system for autonomous sonar arrays represents a number of key technical challenges. Many important multicomputer architecture considerations must be addressed in terms of speed, cost, weight, power, and reliability for each node. To address these considerations,

Tasks for future research include the development of the preliminary software system for the advanced sonar array and the development of a suite of simulation tools which support the design and analysis of both system performance and dependability. Parallel programs will be developed with inherent granularity knobs based on the results of algorithm decomposition and partitioning in the already conducted. Self-healing extensions of the selected algorithms will be developed and simulated. Finally and concurrently, a suite of simulation tools will be developed and integrated to support the rapid virtual prototyping of topology, architecture, protocol, and algorithm selections made in the first phase. A new interface between parallel beamforming programs written in MPI and the network and node architecture models constructed in BONeS is under development which promises to make it possible to conduct detailed performance

analyses with cutting-edge parallel algorithms on candidate distributed sonar array architectures in a rapid virtual prototyping fashion.

7. Acknowledgements

We gratefully acknowledge the support of CDR Mitch Shipley, Dr. John Tague, Dr. Donald Davison, Mr. Tommy Goldsberry, and others in the ASW Surveillance Programs at the Office of Naval Research.

8. References

[ALTA94] Alta Group of Cadence Design Systems, Inc. *BONeS Designer: Core Library Reference*. Foster City, CA: Alta Group, 1994.

[GEOR97] George, Alan D., William Phipps, Robert Todd, and Warren Rosen. "Communication Protocol Enhancements for SCI-based SCALE Systems." *Proceedings of the 7th International SCI Workshop* (submitted), March 1997.

[GORA95] Goralski, Walter J. *Introduction to ATM Networking*. New York: McGraw-Hill, 1995.

[GROP] Gropp, William and Ewing Lusk. "User's Guide for mpich, a Portable Implementation of MPI." http://www.mcs.anl.gov/mpi/mpiuserguide/paper.html .

[HAMP93] Hampson, Grant, and Andrew P. Papliński. "Beamforming by Interpolation." Technical Report 93/12. Clayton, Victoria, Australia: Monash U, 1993. ftp://ftp.rdt.monash.edu.au/pub/techreports/RDT/93-12.ps.Z

[JOHN93] Johnson, Don H. and Dan E. Dudgeon. *Array Signal Processing, Concepts and Techniques.* New Jersey: Prentice Hall, 1993.

[MORG94] Morgan, Don. *Practical DSP Modeling, Techniques, and Programming in C*. New York: Wiley, 1994.

[MPIF93] Message Passing Interface Forum. "MPI: A Message Passing Interface." *Proceedings of Supercomputing 1993*. IEEE Computer Society Press (878-83): 1993.

[NIEL91] Nielsen, Richard O. *Sonar Signal Processing*. Boston: Artech House, 1991.

[PARA95] Parallab. "Programmer's Guide to MPI for Dolphin's SBus-to-SCI Adapters." Bergen: Parallab, U of Bergen, 1995.

[ROBB96] Robbins, Kay A. and Steven Robbins. Practical UNIX Programming: A Guide to Concurrency, Communication, and Multithreading. Upper Saddle River, N.J: Prentice Hall, 1996.

[SCI93] *Scalable Coherent Interface.* ANSI/IEEE Standard 1596-1992. Piscataway, New Jersey: IEEE Service Center, 1993.

[SMIT95] Smith, Winthrop W. and Joanne M. Smith. *Handbook of Real-Time Fast Fourier Transforms: Algorithms to Product Testing*. New York: IEEE, 1995.