



High-performance Computing and Simulation (HCS)
Research Laboratory



Grant Number N00014-98-1-0188
to the University of Florida
January - December 1998

“Parallel and Distributed Computing Architectures and Algorithms for Fault-Tolerant Sonar Arrays”

Annual Report #3

Submitted to the:
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217-5660

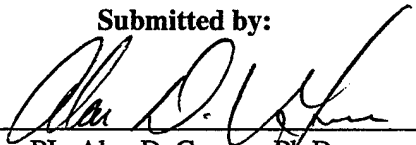
January 29, 1999

DISTRIBUTION STATEMENT A

**Approved for public release;
Distribution Unlimited**

Attention:
Dr. Don Davison
ONR 321

Submitted by:


PI: Alan D. George, Ph.D.

Assoc. Professor of ECE and Director, HCS Research Lab
Dept. of Electrical and Computer Engineering
University of Florida
PO Box 116200, 327 Larsen Hall
Gainesville, FL 32611-6200
Phone: (352)392-5225, e-mail: george@hcs.ufl.edu

19990209 059

Contributing Research Team Members:

Alan George, Ryan Fogarty, Ken Kim, Jeff Markwell,
Michael Miars, Jesus Garcia, Chris Gomez, and Shonda Walker

Table of Contents

1. INTRODUCTION	1
2. REAL-TIME SONAR BEAMFORMING ON HIGH-PERFORMANCE DISTRIBUTED COMPUTERS	4
2.1. INTRODUCTION.....	4
2.2. CONVENTIONAL BEAMFORMING	6
2.3. ITERATION DECOMPOSITION	10
2.4. ANGLE DECOMPOSITION	13
2.5. PIPELINED ANGLE DECOMPOSITION	15
2.6. COMPARATIVE ANALYSIS	17
2.7. CONCLUSIONS	20
2.8. REFERENCES	21
3. PARALLEL ALGORITHMS FOR SPLIT-APERTURE CONVENTIONAL BEAMFORMING	23
3.1. INTRODUCTION.....	23
3.2. OVERVIEW OF SPLIT-APERTURE BEAMFORMING	24
3.3. PERFORMANCE ANALYSIS OF SEQUENTIAL SPLIT-APERTURE BEAMFORMERS.....	27
3.4. PARALLEL ALGORITHMS FOR SPLIT-APERTURE BEAMFORMERS.....	29
3.4.1. <i>Iteration-decomposition method</i>	31
3.4.2. <i>Angle-decomposition method</i>	32
3.5. PERFORMANCE ANALYSIS OF SPLIT-APERTURE BEAMFORMERS.....	34
3.6. CONCLUSIONS	38
3.7. REFERENCES	39
4. AN INTEGRATED SIMULATION ENVIRONMENT FOR PARALLEL AND DISTRIBUTED SYSTEM PROTOTYPING	40
4.1. INTRODUCTION.....	40
4.2. RELATED WORK.....	41
4.3. ISE FRAMEWORK.....	43
4.3.1. <i>High-fidelity Models</i>	44
4.3.2. <i>High-level Parallel API</i>	48
4.3.3. <i>Putting It All Together</i>	49
4.4. CONTAINMENT OF SIMULATION EXPLOSION	50
4.5. VALIDATION AND CASE STUDY.....	52
4.5.1. <i>Description of Validation Experiments</i>	52
4.5.2. <i>Description of Case Study</i>	54
4.6. RESULTS.....	55
4.6.1. <i>Results of Validation Experiments</i>	55
4.6.2. <i>Results of Case Study</i>	58
4.7. CONCLUSIONS	59
4.8. REFERENCES	60
5. PRELIMINARY DESIGN AND MEASUREMENTS WITH THE DISTRIBUTED PARALLEL SONAR ARRAY PROTOTYPE	62
5.1. THE TMS320C542 DSKPLUS DEVELOPMENT BOARD.....	62
5.2. COMMUNICATIONS IMPLEMENTATION.....	64
5.3. PERFORMANCE RESULTS	66
5.4. CONCLUSIONS	68
6. CONCLUSIONS AND FUTURE RESEARCH	69

1. INTRODUCTION

Quiet submarine threats and high clutter in the littoral undersea battlespace require that higher-gain acoustic sensors be deployed for undersea surveillance. This development requires the implementation of high-element-count arrays and a corresponding increase in data rate and the associated signal processing. As a result, the processing requirements placed on the data collector/processor are becoming prohibitive in terms of cost and electrical power. This project addresses the potential of advanced distributed and parallel processing techniques to distribute the processing among the individual sensor elements of the large surveillance array.

Distributed and parallel processing techniques together with advanced networking technologies and architectures can be used to turn the telemetry nodes of autonomous sonar arrays into processing nodes of a large parallel processor. This approach will eliminate the need for a centralized data collector/processor, reduce the aggregate current drain, and increase overall system reliability. Furthermore, by using the spare processing capacity in the processors required to implement the low-power interface protocol together with the high data rate offered by fiber optics, this improvement can be achieved at essentially no increase to the per-node cost of the array.

The goals of this project are to decrease the cost and size and increase the dependability and performance of large, autonomous sonar arrays currently under development. The specific goals are to eliminate the centralized end-data collector/processor as a single-point-of-failure, potential performance bottleneck, and major cost driver and to decrease substantially the aggregate current drain and cost of the array. The target current drain and cost for this array are 44 mA per telemetry node plus an estimated 2 A for the end processor and a cost of \$500 per node.

In FY96 progress was made in several areas. First, an analysis of the effect of node outage on network reliability was conducted. Second, a survey of low-cost, low-power networking components was started. Third, preliminary parallel decompositions of several standard beamforming algorithms were performed, including delay-and-sum, delay-and-sum with interpolation, and FFT. Algorithms and programs for the sequential versions of these beamforming techniques were developed in Matlab, C, and MPI to form a baseline by which parallel algorithms and software are measured. Fourth, a fine-grain model of the baseline freight train protocol has been developed using the Block-Oriented Network Simulator (BONeS) tool, and models for candidate network architectures was under development for unidirectional ring and bidirectional linear array topologies.

In FY97 further progress was made in a number of areas. A taxonomy of decomposition and parallelization methods for conventional beamforming, both time- and frequency-domain, was developed. Frequency-domain parallel beamforming algorithms using iteration- and steering-decomposition methods were designed, developed, and analyzed. By coding these parallel algorithms in MPI as part of the preliminary software system, performance experiments were conducted on a cluster testbed via several simulated network candidates. Results indicated near-linear speedup on both ring and bidirectional array networks and this speedup is critical since it will permit reduction in node and network clock rates, thereby further reducing power consumption. Time-domain parallel beamforming algorithms using iteration pipelining, transpose methods, and global data scope extensions neared completion, and early indications were also promising. Fine-grain array network models were designed, developed, and verified including unidirectional, bidirectional, token ring, and register-insertion ring protocols, and a slotted-ring model was near completion. Medium-grain array node models were also under development and these models will permit parameterization and experimentation with clock

speed, precision, operational versus standby power mode, etc. Integrating these many developments together, a new modeling and simulation environment for rapid virtual prototyping of advanced sonar arrays, called the Integrated Simulation Environment (ISE), was designed and a working version was near completion. For the first time, ISE brings together the fine-grain network models, parallel beamforming software, and the medium-grain node architecture models in a manner that permits detailed experimentation with candidate algorithms, network architectures, and node architectures in a dynamic and integrated fashion. The evaluation of low-power networking and processing components was continued. The study of the effect of component failures was also continued and simple and efficient techniques for sidelobe restoration in the event of node failure were evaluated.

In FY98, the focus of this annual report, work has been thoroughly pursued for single-aperture conventional parallel beamforming, recently completed for split-aperture parallel beamforming, and just begun for more advanced adaptive or optimal processing algorithms. Specifically, five new parallel DFT beamforming algorithms have been developed. For conventional single-aperture arrays the algorithms include iteration decomposition, angle decomposition, and pipelined angle decomposition. Iteration decomposition and angle decomposition were further used to parallelize split-aperture arrays. Iteration decomposition of the DFT beamformer is based on a coarse-grained scheduling algorithm that pipelines complete iterations (i.e. beamforming cycles). Angle decomposition of the parallel DFT beamforming algorithm is based on a medium-grained algorithm that partitions the operations internal to a single iteration across all nodes. Lastly, pipelined angle decomposition optimizes angle decomposition by pipelining computation and communication stages.

To address the growing challenges in designing and analyzing non-traditional parallel and distributed sonar arrays, the first complete implementation of the novel approach and tool begun in FY97 and known as ISE was completed. On the hardware side, it is difficult to design an autonomous system that will run demanding parallel beamforming applications efficiently. On the software side, it is challenging to parallelize a beamformer to run well on an autonomous sonar array when that system is not available as a physical prototype. Because of the complexity involved, designers from both development viewpoints have had a growing need for a sonar array design environment to address these challenges. In support of this project, researchers at the High-performance Computing and Simulation (HCS) Research Laboratory developed the ISE modeling environment wherein parallel and distributed array architectures are simulated using a combination of fine-grain models and existing hardware-in-the-loop (HWIL) to execute actual parallel programs. ISE allows the designer to run these simulations over networked workstations; thus, the workload can be distributed when multiple parameter sets are applied to high-fidelity models. The current implementation of ISE interfaces virtual sonar array prototypes created in the Block-Oriented Network Simulator (a CAD simulation tool from Cadence Design Systems) with parallel beamformer applications written in the popular Message-Passing Interface (MPI) coordination language for C or C++ programs.

Finally, the implementation of a prototype for the distributed parallel sonar array has been completed. A parallel beamforming program has been ported, mapped, and executed on this prototype and preliminary performance measurements gathered. This prototype consists of eight single-board embedded computers, each comprised of a TMS320C54 DSP microprocessor and a daughter card for maximum off-chip memory expansion, connected by a serial TDM network using the internal TDM protocol in the C54 device.

The remainder of this report is organized as follows. In Chapter 2, detailed descriptions and analyses of the parallel algorithms and programs for conventional beamforming are presented. In Chapter 3, a similar presentation is given on the new split-aperture parallel beamforming algorithms, programs, and results. Chapter 4 provides an overview of the mechanisms and

advantages of the Integrated Simulation Environment for rapid virtual prototyping of distributed parallel sonar arrays, followed by preliminary results with the physical prototype in Chapter 5. Finally, conclusions and issues of critical future research are presented in Chapter 6.

2. REAL-TIME SONAR BEAMFORMING ON HIGH-PERFORMANCE DISTRIBUTED COMPUTERS

Advancements in beamforming algorithms are exceeding the computation and communication capabilities of traditional sonar array systems. Future sonar systems may take advantage of in-array processing by coupling transducer nodes with low-power DSP microprocessors to glean performance benefits, increased fault tolerance, and lower cost. This chapter explores parallel algorithms for conventional frequency-domain beamforming designed for an in-array processing system. The coarse-grained and medium-grained parallel algorithms presented offer scaled speedup while providing the basis for adaptations in advanced beamforming algorithms.

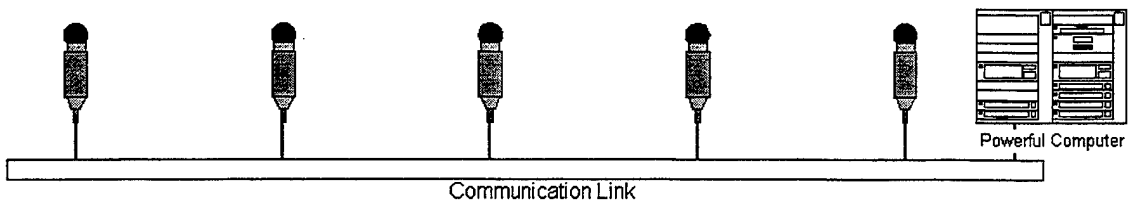
2.1. Introduction

Quiet submarine threats and high clutter in the littoral undersea environment demand that higher-gain acoustic sensors be deployed for undersea surveillance. The effect of this trend is high-element-count sonar arrays with increasing data rates and associated signal processing. The U.S. Navy is developing low-cost, disposable, battery-powered, rapidly deployable sonar arrays. These autonomous passive sonar array technologies face limitations which may be naturally overcome with the use of parallel and distributed computing (PDC) technology. Limitations include low fault-tolerance due to single points of failure, and computational complexity that cannot be supported in real-time by conventional means. The limitations are especially evident for a large number of receiving nodes and with the continuing development of higher-fidelity algorithms such as adaptive and matched-field processing.

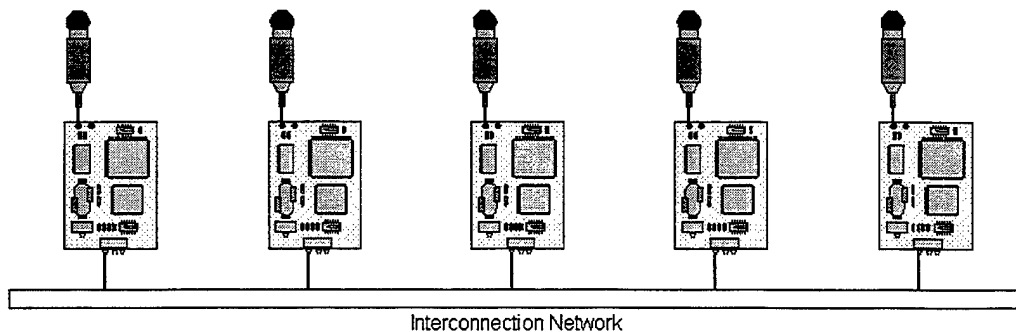
The next generation of passive array systems will support powerful signal processing algorithms, provide fault-tolerant mechanisms to overcome node and communication link failures, and operate at very low power levels so that they may be operated on battery power for mission times measured in weeks or months. These arrays will be capable of uplinking real-time beamformed data with better resolution than conventional systems while lowering cost by exploiting commercial off-the-shelf (COTS) microprocessors and subsystems. This paper presents new parallel algorithms for conventional beamforming (CBF) optimized for future in-array processing sonar systems. These embedded technologies have the advantages that the computational ability scales with array size and that fault tolerance is increased via elimination of single points of failure.

Many optimizations exist for conventional beamforming that improve the algorithm complexity. Mucci presents seven categories of CBF algorithms, each having different spectral characteristics and hardware requirements. These categories include delay-and-sum, partial-sum, interpolation, interpolation with complex sampling, shifted-sideband, discrete Fourier transform (DFT), and phase-shift beamformers [15]. Although other optimizations exist, such as Houston's fast beamforming (FBF) algorithm [8], these seven can serve as a basis. Each algorithm is related to the fundamental delay-and-sum CBF algorithm. The first five algorithms work in the time domain while the latter two (i.e. DFT and phase-shift beamforming) work in the frequency domain. The delay-and-sum algorithm has the unfortunate limitation that the spatial resolution is dependent on sampling frequency, resulting in a very large data space when large numbers of steering directions are desired. Each of the other algorithms aforementioned is optimized by minimizing this characteristic to varying degrees.

The DFT beamforming algorithm was chosen as the algorithm for the in-array parallel processing system. Advantages of the DFT beamformer include its ability to be updated for use with adaptive algorithms such as the Minimum Variance Distortionless Response (MVDR) beamformer [11]. Also, inverse transforming is not required since the frequency information is often advantageous for signal detection, localization, and classification in post-processing objectives [15]. DFT beamforming algorithms have the natural ability to support any number of steering directions at arbitrary angles, and take advantage of the efficiency inherent in the FFT algorithm. McMahon discusses some of the disadvantages of DFT including the inability of the Fourier domain to track certain classes of pulses, which is a general limitation of high-resolution Fourier analysis [12]. Also, even though the memory space required for the DFT beamformer is smaller than that for the delay-and-sum beamformer, Mucci points out that the DFT has a large data space in comparison to memory-efficient methods such as partial-sum, the interpolation techniques, and the phase-shift algorithm. Despite these limitations, DFT beamforming algorithms are attractive for parallel sonar arrays.



a. Conventional passive sonar array



b. Distributed parallel sonar array

Fig. 2.1 Conventional passive array vs. distributed parallel sonar array.

Conventional arrays may be described as a string of “dumb nodes” (i.e. nodes without processing power) with a large front-end processor, shown in Fig. 2.1a. These dumb nodes may be naturally outfitted with intelligent COTS microprocessors, shown in Fig. 2.1b. Emerging technologies for sonar signal processing arrays will exploit such intelligent distributed-memory multicomputer systems. These systems are typically programmed in a Multiple Instruction Multiple Data (MIMD) [4] fashion using a message-passing paradigm. Coarse-grained parallel decompositions are usually the preferred approach on distributed-memory multicomputers; however, medium-grained algorithms are also feasible with the advent of fast interconnection networks with lightweight communications. The coarse-grained and medium-grained parallel CBF algorithms introduced in this paper can be used as the foundation for more sophisticated beamformers that one day will be targeted for intelligent array systems. These advanced

techniques, in increasing order of complexity, include split-aperture beamforming, adaptive beamforming, matched-field tracking, and matched-field processing. The conventional beamformer exploited and parallelized in this paper will remain as a fundamental portion of such future algorithms, and thus the parallel algorithms presented can serve as a basis for all future work in this field.

Other research initiatives in the parallelization of beamforming algorithms include decompositions of conventional techniques over tightly coupled shared-memory multiprocessors. Although the amount of DSP research on array processing is abundant, little has been accomplished in the area of MIMD-style decompositions. Several projects from the Naval Undersea Warfare Center (NUWC) have developed real-time sonar systems by exploiting massive parallelism. Salinas and Bernecky [17] mapped the delay-and-sum beamformer to a MasPar, a Single Instruction Multiple Data (SIMD) architecture. Dwyer used the same MasPar machine to develop an active sonar system [3]. Lastly, Zvara built an active sonar processing system on Connection Machine's SIMD architecture, the CM 200 [22]. A handful of other papers discuss a variety of adaptive algorithms on parallel systems including systolic arrays, SIMD style machines, and COTS-based DSP multicomputers [1,19]. However, little of the previous work has focused on issues involved in either distributed or in-array processing.

This paper introduces three parallel DFT beamforming algorithms for distributed in-array processing: iteration decomposition, angle decomposition, and pipelined angle decomposition. In Section 2.2, the reader is presented with basic beamform theory and the sequential DFT beamformer. In Section 2.3, the parallel iteration-decomposition beamformer, which is based on a pipelined approach, is introduced. In Section 2.4, angle decomposition, a data-parallel approach, is discussed. The third parallel beamforming algorithm, pipelined angle decomposition, is presented in Section 2.5 and is shown to be a hybrid of the first two parallel algorithms. The sensitivity of each algorithm to array parameters is shown in Section 2.6 using a CAD-based rapid virtual prototyping environment. Finally, in Section 2.7, a brief set of conclusions and directions for future research is enumerated.

2.2. Conventional Beamforming

Conventional delay-and-sum beamforming may be performed in either the frequency or time domain. In either domain, the algorithm is essentially the same: signals sampled across an array are phased (i.e. delayed) to steer the array in a certain direction, after which the phased signals are added together. To phase the incoming signals, some geometry is needed to transform the steered "look" direction to the correct amount of delay at each node. Fig. 2.2 shows an incoming plane wave and the geometry needed to derive the delay.

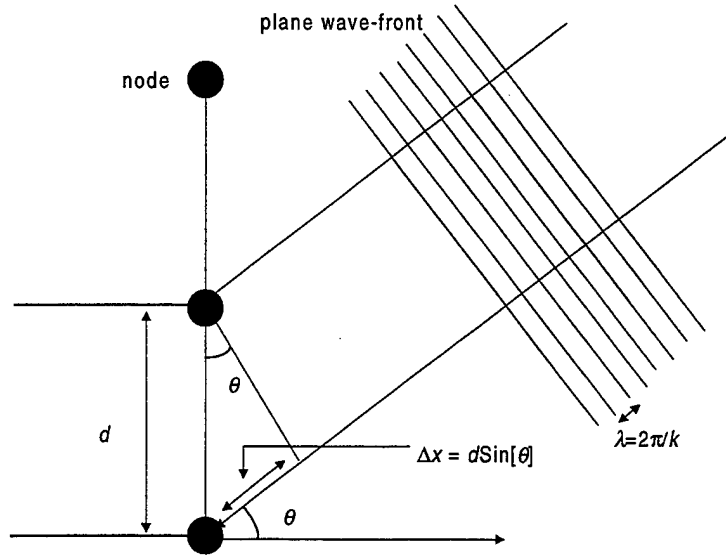


Fig. 2.2 Wave-fronts hitting array.

This delay is directly proportional to Δx as shown in the figure. Adjacent nodes would receive the wave-front at a difference in time of

$$\delta = \frac{d \sin(\theta)}{c}, \quad (\text{Eq. 2.1})$$

where d is the distance between adjacent nodes, θ is the steering angle from the array's perpendicular (i.e. from broadside), and c is the speed of sound in water (estimated as 1500m/s). Each node, indicated by its node number m , would receive a signal from angle θ at a relative delay of

$$(m-1)\delta, \quad (\text{Eq. 2.2})$$

with respect to node 0. In the frequency domain, a vector \underline{g} may be built which, when multiplied by the respective incoming signals, will properly phase the system. This vector is defined as

$$\underline{g} = [1, e^{-jkd \sin(\theta)}, e^{-j2kd \sin(\theta)}, \dots, e^{-j(M-1)kd \sin(\theta)}], \quad (\text{Eq. 2.3})$$

where $k = \frac{2\pi}{\lambda} = \frac{\omega}{c}$ (Eq. 2.4)

is equal to the wave number and M is the number of nodes. The final beamform equation is a summation, $\underline{x}(t)$, of the phased signals multiplied by a windowing weight matrix \underline{w}^t :

$$y(t) = \underline{w}^t \underline{x}(t) \quad (\text{time domain}) \quad (\text{Eq. 2.5})$$

or $y(\omega) = \underline{w}^\omega \underline{x}(\omega)$ (frequency domain). (Eq. 2.6)

Readers interested in a more complete discussion of beamforming algorithms are referred to [2,7,10].

The baseline DFT beamforming algorithm, adapted from [16], consists of five simple operations: a window multiplication, DFT (via the radix-2 FFT adapted from [14]), steering factor (i.e. phasing) multiplication, beamforming summation, and last, computation of each

angle's power and inverse Fourier transform. Though not necessary for some types of post-processing, the inverse transform is included in order to output the beamformed time series. The steps are shown in Fig. 2.3, where each of the operations is annotated with its computational order and the dimension of the data stream entering and exiting the block. The dimension maps for the *Window Factor Multiplication*, *FFT*, and *Steering Factor Multiplication* operations are shown on a per-node basis. The dimensions are representative of three significant parameters: *number of nodes (M)*, *number of frequency bins (N)*, and *number of steering directions (S)*. The computational complexity variable n is a generalization of the variables M , N , and S .

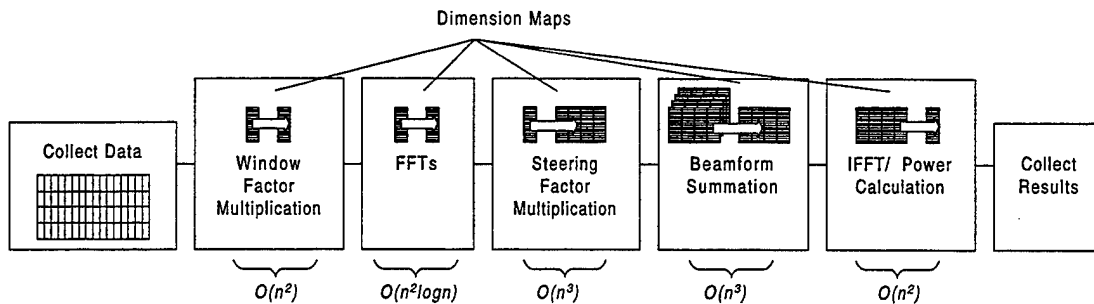


Fig. 2.3 Flowchart for sequential DFT beamformer.

The *Window Factor Multiplication* stage scales each node's input by some factor. For conventional beamforming, a windowing function such as Hanning, Blackman, or Dolph-Chebyshev is typically applied across the hydrophone array such that nodes toward the ends of the array contribute less in the *Beamform Summation* stage. This method improves the signal-to-interference ratio and is analogous to windowing on Fourier transformations [18]. Adaptive techniques adjust these scalar weights to optimize the beamform solution by minimizing output power in a constrained manner [5,21].

The *FFTs* stage takes the scaled temporal data stream and converts it into complex-valued spectra. Since the amount of data entering the stage equals the amount of data exiting, it is reasonable to assume that this operation should be performed within each node to preserve the natural data parallelism of the system (i.e. each node transforms its own sample sets).

The *Steering Factor Multiplication* stage is the major computational bottleneck of DFT beamforming and is not required in time-domain algorithms. It is interesting to note that this *Steering Factor Multiplication* stage is exactly the same operation as correlation by plane-wave replica vectors and thus leads naturally to matched-field techniques. The steering factor matrix may be recomputed with each iteration of the beamformer, or it may simply be stored and retrieved from memory. The former method will result in a memory efficient model while the latter will result in a computationally efficient model.

The *Beamform Summation* stage represents the other major computational bottleneck. Though the stage is $O(n^3)$, it is comprised of simple additions, which is less work than the comparable number of complex multiplies required in the *Steering Factor Multiplication* stage. This stage is the actual spatial filter, thus the reduced output data is ideally filtered spectra for each angle of interest. This operation is the same for any beamforming method in time or frequency, using conventional or adaptive techniques.

Last, the *IFFT/Power Calculation* stage transforms the data back into the time-domain and calculates the power of the beamformed spectra for each steering direction. This stage is often left for post-processing stages or completely ignored because spectral information is many times more useful than temporal data when human operators (or pattern recognition post-processors)

analyze the data. Nonetheless, for completeness and to better support in-array processing, both the IFFT operation and the power calculation are included in all our algorithms.

To further analyze the performance characteristics of this beamformer and its stages, the sequential baseline algorithm was coded and executed on an UltraSPARC-1 workstation running at 170 MHz with 128MB of memory. Fig. 2.4 shows the algorithm's performance when computing 91 and 181 steering directions for different numbers of input sensors (i.e. problem sizes). The top of each stacked bar represents the total time to complete, and each bar is partitioned by the five stages discussed above (where the *Window Factor Multiplication* is combined with the *FFT* stage and the inverse *FFT* is combined with the *Beamform Summation* stage), with *Other* representing the overhead of the timing functions employed for the measurements. As predicted, the *Steering Factor Multiplication* stage represents the largest bottleneck followed by the *Beamform Summation* stage. The remaining stages represent a small fraction of the total computation time. This algorithm was optimized for memory efficiency by recomputing the steering matrix factors at each iteration; therefore, the execution time of the steering multiplication could be significantly decreased if desired, but with the penalty of a larger memory requirement. We chose to ignore the communication time required to gather samples from a passive sonar array with dumb nodes since a well-designed sequential algorithm could always pipeline this process with practically no overhead, assuming that the network is moderately fast compared to the processing speed.

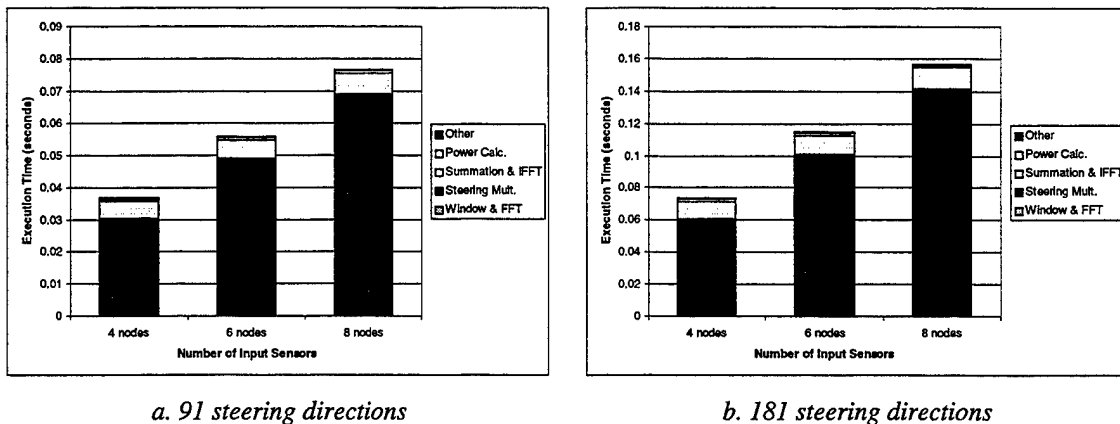


Fig. 2.4 Average execution times for the DFT beamformer (averaged over 1000 iterations).

In order to maximize the efficiency of the parallel DFT beamforming algorithms, the *Steering Factor Multiplication* and *Beamform Summation* stages must be parallelized so as to overlap or partition these computations while simultaneously minimizing communication requirements. Each of the algorithms we present in the following three sections (i.e. iteration decomposition, angle decomposition, and pipelined angle decomposition) ensure that these computationally complex stages are parallelized optimally.

Fine-grained decompositions of the DFT beamformer were not attempted since such a solution causes an excess of communication, which is difficult to support on a distributed-memory multicomputer. Such fine-grained decompositions include partitioning each node's *FFT* and *Window Factor Multiplication* stages, which would be ill-suited for a loosely coupled system.

2.3. Iteration Decomposition

The first parallel algorithm for in-array DFT beamforming focuses on the partitioning of iterations, where an iteration is defined as one complete beamforming cycle. Iteration decomposition of the DFT beamformer is based on a coarse-grained scheduling algorithm that pipelines the *Steering Factor Multiplication*, *Beamform Summation*, and *IFFT/Power Calculation* stages. The *Window Factor Multiplication* and *FFT* stages take advantage of the natural distribution of data across sonar nodes, transforming the data before relaying it to a scheduled processor. This initial data-parallel approach is adapted for each of the other parallel algorithms as well. Intuitively, it is often wise to decompose the program to take advantage of the proximity between available data and processors if that data does not have dependencies. Using this intuition, one might choose to compute the *Steering Factor Multiplication* stage at each node for local data since data dependencies do not exist until the *Beamform Summation* operation. However, the large space that would need to be communicated later for the summation would be many times larger causing significant communication complexity. This characteristic is due to the fact that, for each input signal, the *Steering Factor Multiplication* stage calculates a phased vector for every steering direction of interest. Therefore, it is desirable to parallelize the *Steering Factor Multiplication* stage to avoid such large memory and communication requirements.

With the iteration-decomposition parallel algorithm, each iteration (from the *Steering Factor Multiplication* stage through the end of the iteration) is scheduled to a processor in a round-robin fashion. At startup, *node 1* is scheduled the first iteration, *node 2* is scheduled the second, etc. The decomposition is split into two computational stages, as shown in Fig. 2.5, which are referred to as *Computation Stages 1* and *2*. The first stage is partitioned in the data-parallel fashion previously mentioned while the second stage uses a control-decomposition approach. Therefore, *Computation Stage 1* requires the computational resources from all the nodes, while *Computation Stage 2* finishes the algorithm on a single processor. To increase the efficiency of the algorithm, iterations are pipelined such that new iterations begin during the *Computation Stage 2* for previously started iterations. The pipeline works by interrupting an iteration's *Computation Stage 2* at well-defined points to begin the *Computation Stage 1* of a new iteration. A 4-node array would require 3 interruptions to start 3 new iterations before the *Computation Stage 2* from any one iteration is completed. Thus, the size of the pipeline depends on the number of nodes in the array.

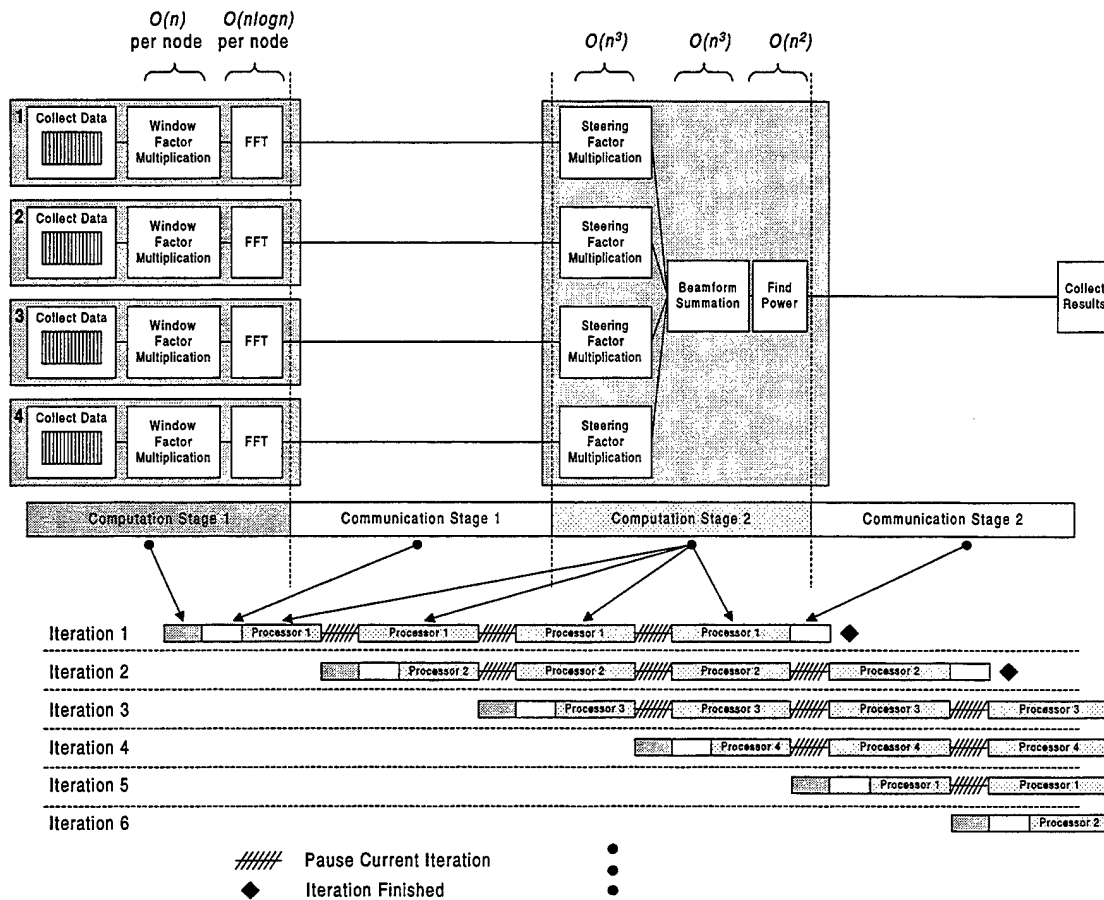


Fig. 2.5 Iteration decomposition.

Iteration decomposition normalizes the amount of computation required between interruptions. This trait is a result of the algorithm's linear dependence on number of nodes and is illustrated by the following example. Consider an 8-node array requiring $S = 32$ steering directions and $N = 10$ frequency bins. The *Steering Factor Multiplication* would require

$$S \cdot N \cdot M = 32 \cdot 10 \cdot 8 = 2560 \quad (\text{Eq. 2.7})$$

complex multiplications. However, the operations are pipelined over 8 iterations during which the processors only need compute 1/8th of the complex multiplies (or 320) per new iteration. If the system were scaled to 16 nodes, the number of total complex multiplies between interruptions would double. However, the processor would have twice the amount of time to compute the result, maintaining 320 complex multiplies per new iteration over a total of 16 pipelined iterations. Ignoring communication, an array that could support 8 nodes could support infinitely many! However, the result latency of any given iteration, which is the time from the original data collection for that iteration until the result is complete, also increases linearly with the size of the array and may conceivably be too long for very large arrays.

The amount of communication also increases linearly with the number of array nodes. More precisely, the complexity of *Communication Stage 1*, which contains the communication of the transformed input vectors to the scheduled processor, increases linearly with the number of nodes (M). *Communication Stage 2*, which contains the communication of the results to a designated I/O node, is independent of M . It may be possible to support arbitrarily sized systems by using

interconnects such as register-insertion rings, which have the ability to scale well [20]. The communication capability of such interconnects increase as nodes are added. Other less sophisticated networks such as bus-based solutions have a natural peak with respect to supportable array size since adding nodes increases contention without increasing network aggregate throughput.

To evaluate the performance of this parallel algorithm, several implementations were coded in C with the Message-Passing Interface (MPI) [13] and executed on a cluster of UltraSPARC workstations connected by a 155 Mb/s ATM network via TCP/IP. For each array size, the top of the stacked bar in Fig. 2.6 represents the average execution time of any given iteration. In Fig. 2.6a the results from computing 91 steering directions are shown, while in Fig. 2.6b the results for 181 steering directions are shown. The times are broken down in stages showing significant computational and communicational operations. As discussed in Section 2.2, the *Steering Factor Multiplication* and *Beamform Summation* stages dominate the computation times. The time spent in the *Window Factor Multiplication/FFT* and *Power Calculation* stages is negligible compared to the more significant of the computational stages. The *Window Factor Multiplication/FFT* stage is so small that it is barely visible in the bottom of each bar graph. There is also significant overhead incurred from communication latencies, which is undoubtedly a result of running the algorithm through the overly robust TCP/IP software layers on top of ATM. We can expect better performance with lighter communication software layers.

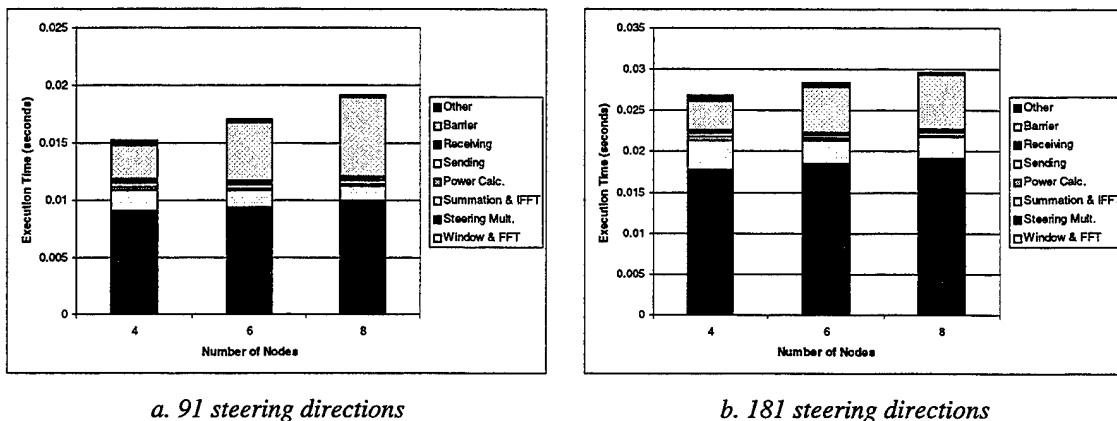
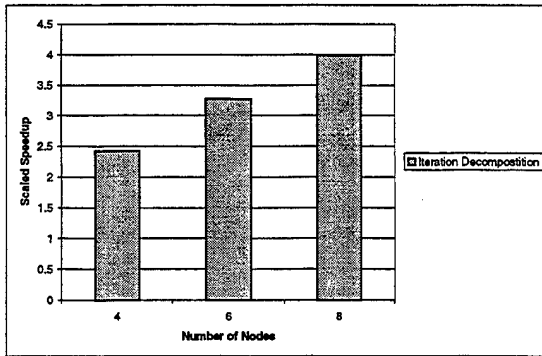
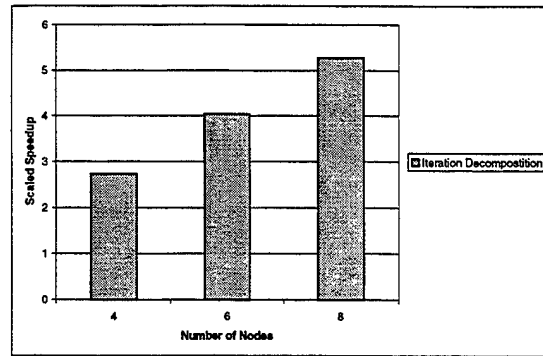


Fig. 2.6 Average execution times for the iteration decomposition method (averaged over 1000 iterations).

Although the result latency for an iteration-decomposition solution is fairly long, the average execution time for any iteration does very well. Iteration decomposition also yields speedup over the sequential program with a parallel efficiency of about 68%, as shown in Fig. 2.7. Because the problem size increases with the number of nodes, these speedup numbers reflect scaled speedup. Furthermore, the comparisons are made to a purely sequential algorithm that ignores communication latency, as previously discussed.



a. 91 steering directions



b. 181 steering directions

Fig. 2.7 Scaled speedup for the iteration decomposition method.

2.4. Angle Decomposition

The second parallel algorithm focuses on partitioning steering angle solutions to each of the nodes, a form of domain decomposition. Angle decomposition of the parallel DFT beamforming algorithm is based on a medium-grained algorithm that partitions the five operations discussed in Section 2.2 across all nodes. The *Window Factor Multiplication* and *FFT* stages (or *Computation Stage 1*) again take advantage of the natural distribution of data across sonar nodes and operate in exactly the same manner as in iteration decomposition. On the other hand, *Computation Stage 2* differs significantly from the method used in the prior decomposition. In angle decomposition, the entire algorithm operates in a data-parallel fashion. Instead of a single node computing the beamform solution for all angles, the nodes divide the S steering directions among the processors and independently beamform in those directions. The number of steering angles each node computes is thus S/M . Using this decomposition, the workload for a single iteration is distributed evenly to all nodes.

Fig. 2.8 shows a block diagram illustrating this method. With respect to *Communication Stage 1*, angle decomposition is immediately distinguishable from iteration decomposition. As shown in the figure, the communication in this stage is an all-to-all communication, which is this algorithm's greatest deficiency. The communication does not scale linearly but rather quadratically with the number of nodes, resulting in a more complex $O(n^2)$ communication. However, with broadcast-capable networks, this complex communication is reduced to $O(n)$. As in the previous algorithm, the total amount of communication in *Communication Stage 2* is independent of array size, although angle decomposition partitions the output data into M smaller segments.

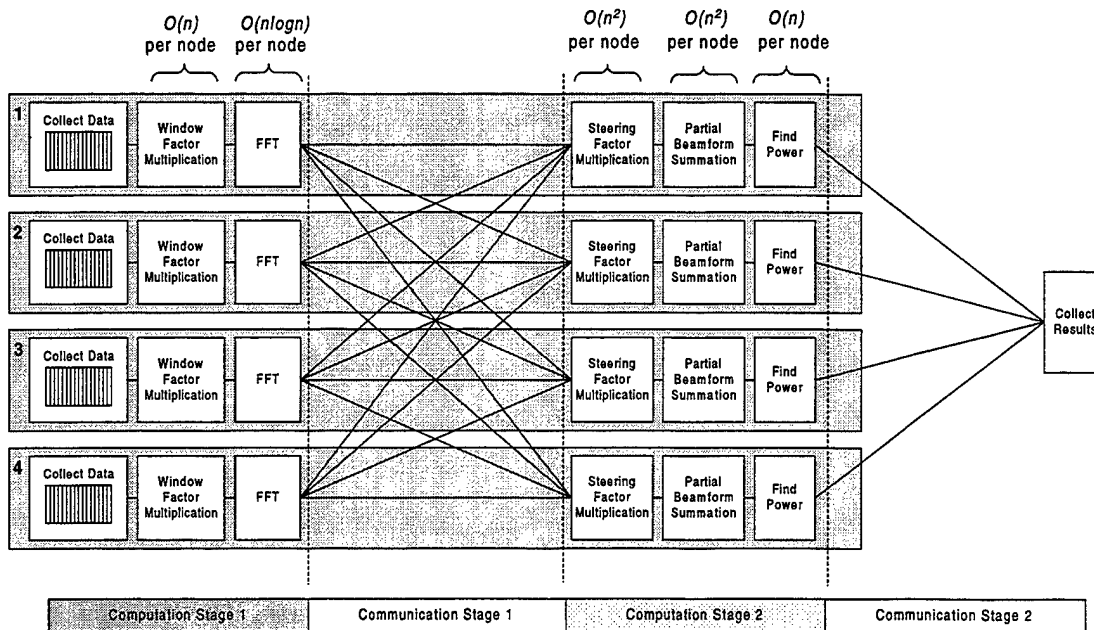


Fig. 2.8 Angle decomposition.

One advantage of this algorithm over iteration decomposition is the low result latency of any individual beamform solution. Recall that with iteration decomposition, each solution's latency is dependent on the size of the array. That is, larger arrays will cause more interruptions for starting new iterations, thus increasing the length of time required to finish computations for any given iteration. Angle decomposition ensures the lowest latency to produce a final result of any decomposition. Using much the same reasoning as for iteration decomposition, the latency of the beamform solution is computationally independent of the array size, a result that is due to the algorithm's linear dependence on M . This linear dependence implies that the latency of any single solution of the algorithm will be the same for eight nodes as it is for infinitely many, again ignoring communicational requirements. If the interconnect employed was fully connected or could scale as the square of M , then the network would also support linear scalability [9]. However, for many applications such a robust network is unlikely to be cost effective.

Another advantage of the angle-decomposition parallel algorithm is the efficient use of memory. Since each node performs a range of steering angles, memory required for the *Steering Factor Multiplication* may be distributed across all nodes. The iteration-decomposition algorithm requires either recomputation of the large steering matrix with each new iteration or copies of the matrix in every node.

For the domain decomposition used in this algorithm, the degree-of-parallelism (DOP) achieved is dependent on the number of steering directions, with high-resolution beamformers expected to glean the best speedup results. The number of frequency bins also increases the DOP but, similar to the number of nodes, has the unfortunate effect of increasing the communication requirements quadratically.

Fig. 2.9 charts the average execution times for computing 91 steering directions and 181 steering directions. The times are again broken down in stages to show significant computational and communicational operations. The *Steering Factor Multiplication* and *Beamform Summation* stages dominate the computation times while the *FFT and Window* and *Power Calculation* stages are negligible. The communication in the angle-decomposition algorithm represents more than

half of the total execution time. Again, these large communication latencies may be exaggerated in comparison to the actual hardware latencies since an in-array processing system will have a lighter communication stack than TCP/IP.

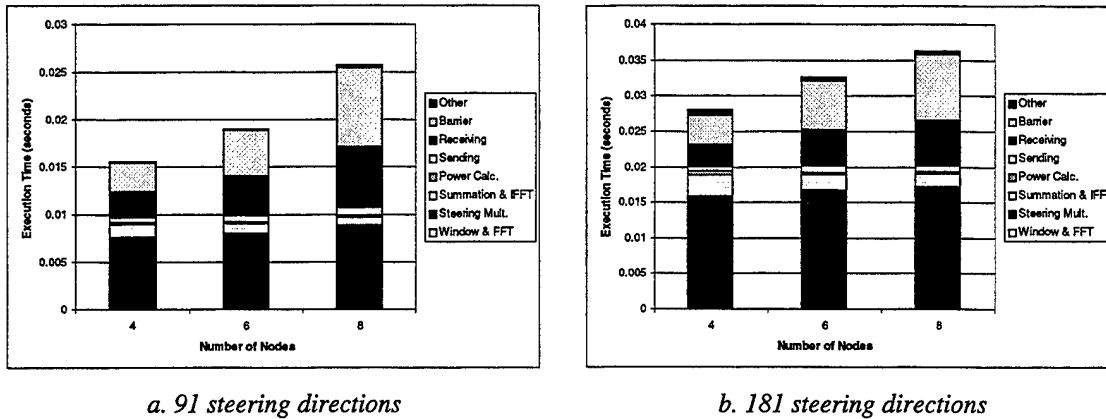


Fig. 2.9 Average execution times for the angle decomposition method (averaged over 1000 iterations).

As shown in Fig. 2.10, the scaled speedup of the angle decomposition is relatively poor, yielding a parallel efficiency of less than 50% in most cases. The large communication latencies were detrimental to the overall performance of the algorithm.

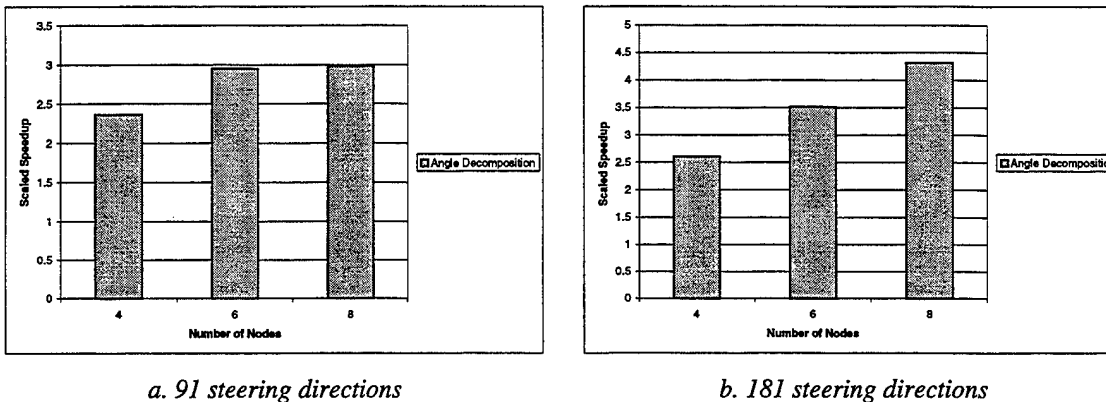


Fig. 2.10 Scaled speedup of the angle decomposition method.

2.5. Pipelined Angle Decomposition

The third parallel algorithm focuses on partitioning steering angle solutions to each of the nodes (i.e. data parallelism), but it also employs pipelining (i.e. control parallelism) to improve efficiency. Pipelined angle decomposition overlaps the communication and computational stages of angle decomposition at the expense of higher result latency for any single beamform iteration. Therefore, pipelined angle decomposition is a compromise between the iteration- and angle-

decomposition algorithms. Pipelined angle decomposition decreases result latency from that in iteration decomposition and achieves better speedup than angle decomposition.

The pipelining, shown in Fig. 2.11, is achieved by overlapping *Communication Stage 1* with *Computation Stage 2* from the preceding iteration and *Computation Stage 1* from the succeeding iteration. The result collection in *Communication Stage 2* is overlapped in a similar fashion. After an iteration's *Communication Stage 1* is initiated, the algorithm picks up the previous iteration at *Computation Stage 2*. At the end of this computational stage, the result collection, *Communication Stage 2*, for that iteration is begun. Finally, before completing *Communication Stage 1*, the algorithm begins *Computation Stage 1* of a new iteration. Thus, an effective overlapping of communication and computation stages is achieved.

Pipelined angle decomposition is not as efficient as iteration decomposition but constrains the length of the pipeline to 3 iterations, as opposed to iteration decomposition whose latency grows as a function of M . Therefore, ignoring the increasing communication latencies, the result latency to produce a single beamform solution in pipelined angle decomposition remains independent of array size, yet is longer than the latency in pure angle decomposition. The DOP of the algorithm remains identical to that of angle decomposition, but average speedup is improved.

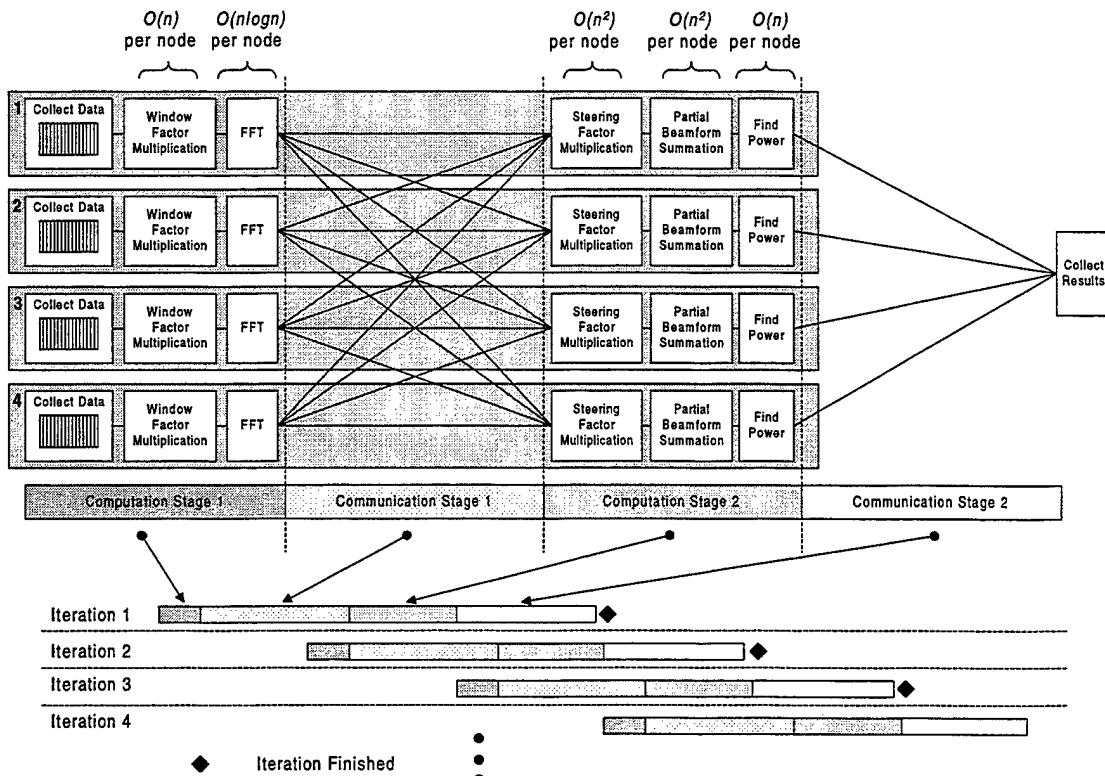


Fig. 2.11 Pipelined angle decomposition.

In Fig. 2.12, the execution times for computing 91 steering directions and 181 steering directions are shown. As expected, the figures show that the pipelining has improved the average execution time from that of angle decomposition. The *Steering Factor Multiplication* and *Beamform Summation* still account for the majority of computational load while the execution times of *FFT*, *Window Factor Multiplication* and *Power Calculation* stages are still insignificant.

Communication latencies of pipelined angle decomposition are minimized, which is clearly shown by the time spent receiving with respect to angle decomposition.

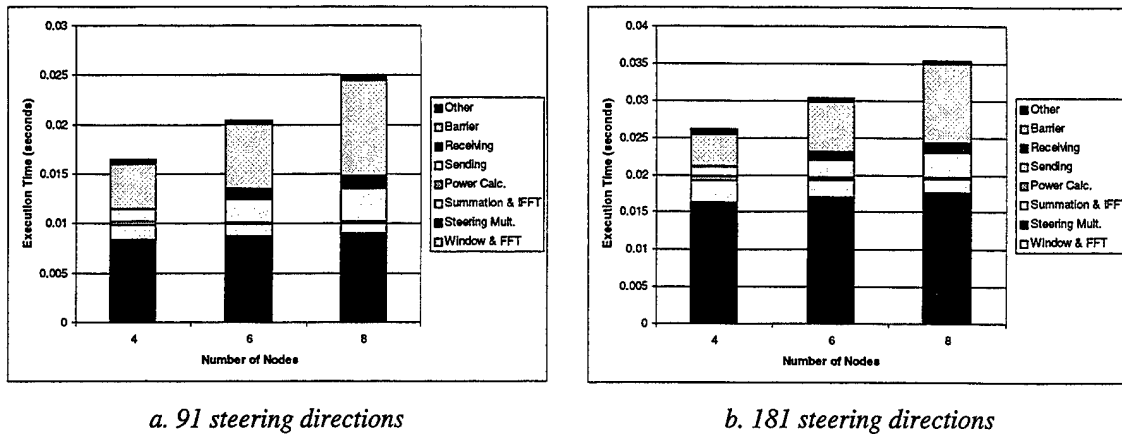


Fig. 2.12 Average execution times for pipelined angle decomposition (averaged over 1000 iterations).

The scaled speedup for pipelined angle decomposition, shown in Fig. 2.13, has improved marginally over angle decomposition for large array sizes by increasing parallel efficiencies to over 50%. Small arrays take advantage of the overlap in communication and computation, increasing parallel efficiency to almost 70%.

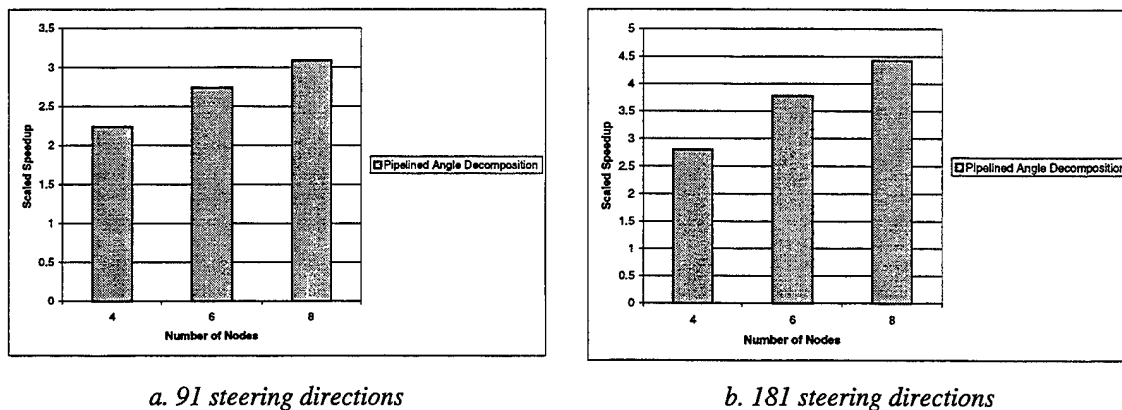
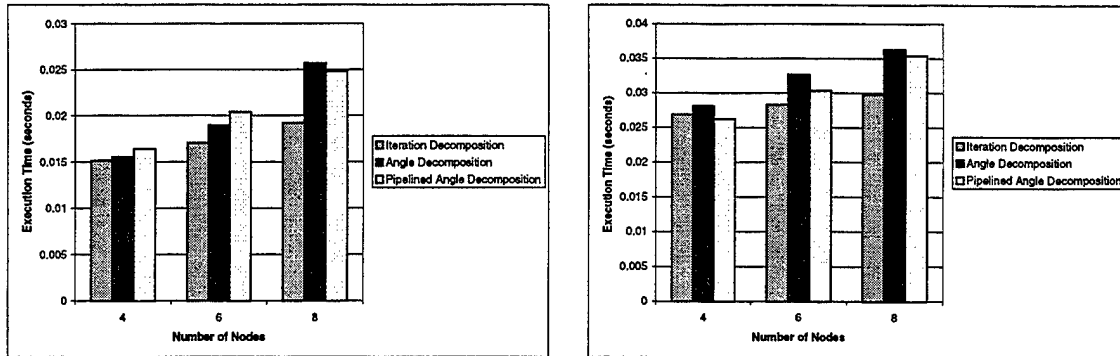


Fig. 2.13 Scaled speedup for the pipelined angle decomposition.

2.6. Comparative Analysis

In this section, the advantages and disadvantages of each of the three algorithms are discussed, and comparisons are made between their performances. Fig. 2.14 shows the average execution time for each of the parallel beamformers. The iteration-decomposition algorithm shows the best performance, whereas the angle-decomposition algorithm generally performs the worst. Of course, the trade-off between these two algorithms is the large result latency incurred for iteration decomposition. The pipelined angle decomposition, as predicted in Section 2.5,

compromises the angle decomposition's short result latency for a somewhat better average execution time. Large array sizes may cause the communication in the two angle-decomposition algorithms to grow quickly, especially if the network cannot support broadcast traffic. A study of this effect is discussed in more detail below.

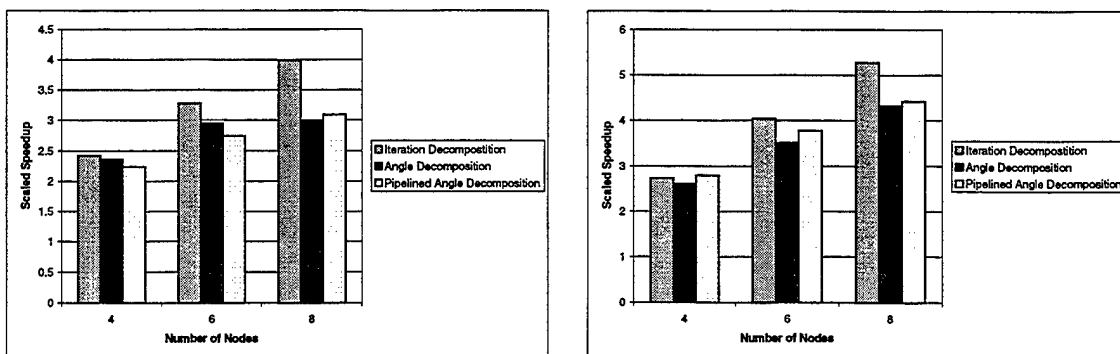


a. 91 steering directions

b. 181 steering directions

Fig. 2.14 Average execution times for the three parallel algorithms (averaged over 1000 iterations).

The scaled speedup plots in Fig. 2.15 indicate that iteration decomposition delivers the best performance of the three algorithms with pipelined angle decomposition performing almost as well in some cases. Although angle decomposition is conceptually less efficient than its pipelined cousin, the added complexity of setting up the pipelined angle decomposition led to more efficient execution by angle decomposition for small arrays with lower steering resolution. Angle decomposition certainly performs less favorably for larger arrays and higher-resolution systems.



a. 91 steering directions

b. 181 steering directions

Fig. 2.15 Scaled speedups for the three parallel algorithms.

To provide a sensitivity study of array parameters on an in-array processing system, each of the parallel beamformers was also executed on the Integrated Simulation Environment (ISE), a rapid virtual prototyping tool developed at the University of Florida [6]. ISE has the ability to run real applications written in MPI over simulated systems built in the Block Oriented Network Simulator (BONeS), a commercial product of Cadence Design Systems. A number of

interconnection schemes have been developed specifically in ISE for prototyping a distributed parallel sonar array. These interconnects include a register insertion ring, a bidirectional register insertion array, and a slotted ring, each of which support linear scalability with increasing numbers of nodes.

ISE allows researchers to experiment with systems that may be too impractical or expensive to prototype in the traditional sense. The following experiments show the sensitivity and scaling ability of the three parallel beamforming algorithms over a larger range of parameters. From this data, a more accurate account of how the decompositions will perform on an actual in-array processing system is gained.

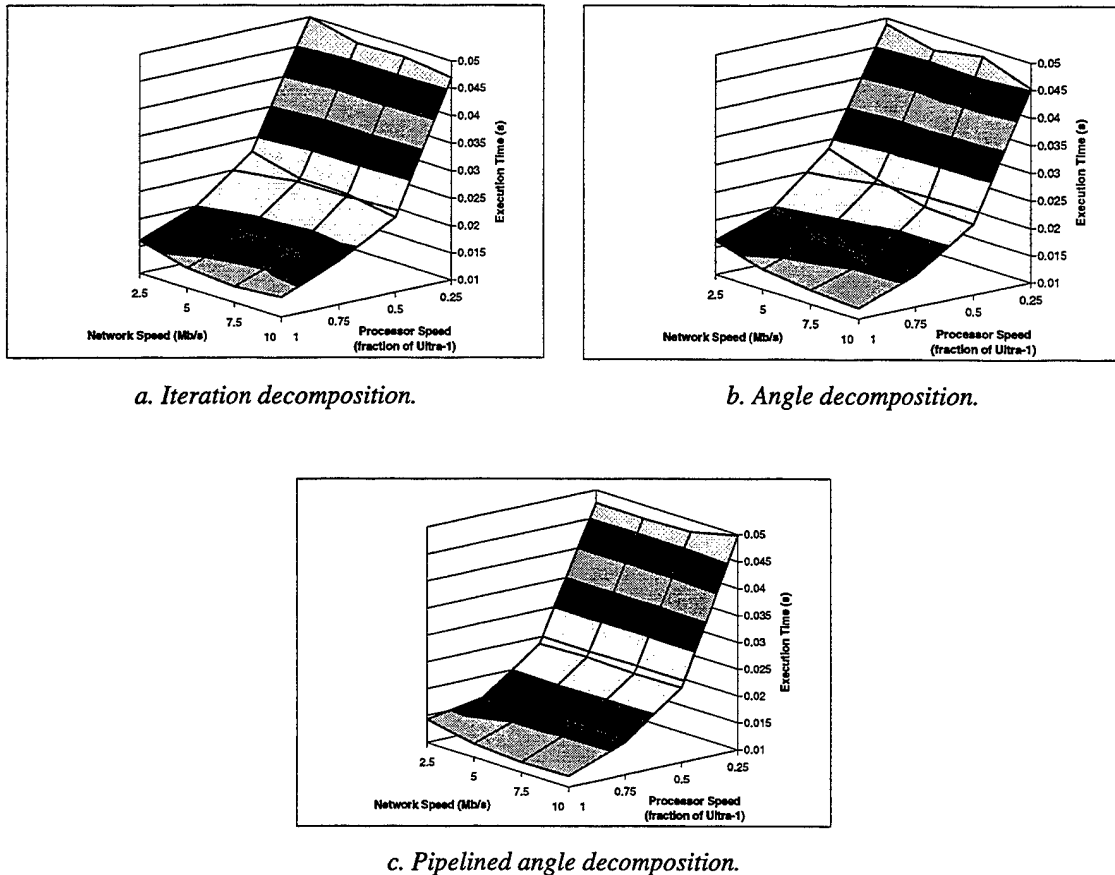


Fig. 2.16 Contour plots for each algorithm ranged over network and processor speed.

The first sensitivity study involves the variation of parameters for network speed and processor speed in an 8-node array with a bidirectional register-insertion network. To study the effect of the speed of the eight processors in the virtual prototype, variations from 25 percent to 100 percent of the performance of an UltraSPARC processor running at 170 MHz are used. In addition, variations in network speed range from 2.5 to 10 Mb/s. Each of the three parallel beamformers was executed on virtual prototypes with these permutations, and execution times are shown in Fig. 2.16. As can be seen, the iteration and angle-decomposition methods provide similar performance results as the speed of the system is varied. The processor speed has the most significant effect on the execution time of these beamformers. As network speed is

decreased, the performance of the algorithms is also decreased, though to a lesser extent. Graphically, this trend appears as a gradual slope in the network-speed dimension. However, pipelined angle decomposition shows little performance change as network speed is varied, indicating the additional ability of this method to overlap computation with communication. However, this result is likely to change for very large arrays, in which communication latencies may be more dominant than computational latencies. Dependency on network speed for this 8-node system only appears when the processor speed is so fast and the network speed so slow as to cause the communication to last longer than the $O(n^3)$ computations. As with the other algorithms, processor speed remains the dominant constraint for pipelined angle decomposition.

The second sensitivity study, shown in Fig. 2.17, demonstrates the scalability of the algorithms as the number of nodes is changed. The execution time increases linearly as the number of nodes is increased; however, the slope is gradual for all three algorithms. This trend bodes well for the angle-decomposition methods, which might be expected to incur large communication latencies for large arrays. From 2 to 32 nodes, the execution time per iteration increases approximately 30% for all decompositions. The results of this experiment also support the premise that problem size grows linearly with number of nodes as stated in Section 2.3. Note that as problem size doubles, the increase in the execution time is approximately 12%. This loss of efficiency for larger arrays is due to synchronization overhead and communication complexity. It is also important to note that a lightweight communications layer was used in this virtual prototype rather than the TCP/IP stack in the cluster testbed of the previous experiments.

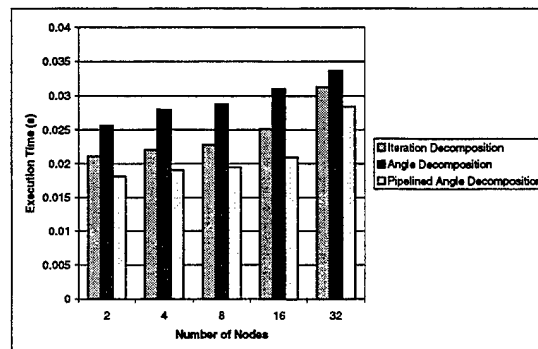


Fig. 2.17 Execution time for each algorithm versus number of nodes.

2.7. Conclusions

The continuing development of beamforming algorithms is creating the need for integrated solutions that leverage algorithmic optimizations with parallel embedded system architectures. These architectures must support a large number of nodes, steer with increased fidelity, and offer better fault tolerance in the event of node failures, all of which place increasing strain on memory requirements, processor speed, network efficiency, and software intelligence. Parallel system architectures hold the potential to eliminate single points of failure and improve hardware fault masking and tolerance, while parallel algorithms for these architectures can support large arrays due to linear dependence of problem size on the number of nodes. The goal of this research was to provide general solutions for in-array parallel beamforming, first for conventional beamformers, but ultimately extensible to split-aperture, adaptive, and matched-field techniques.

The configuration of a sonar array maps particularly well to loosely coupled multicomputer architectures, thus the parallel solutions were constrained to coarse-grained and medium-grained

decompositions. Although fine-grained solutions are not impossible, they are certainly inefficient on these systems. Using the sequential DFT beamformer as a baseline, new parallel algorithms were developed using both coarse-grained iteration and medium-grained angle decomposition. In addition, by combining attributes of both iteration and angle decomposition, a hybrid form of parallel algorithm was developed.

The iteration-decomposition algorithm was found to exhibit the best efficiency and the best scalability at the cost of a large result latency. Of the algorithms presented, it is also the least complex to implement and is easily adaptable in the event of hardware faults. The angle-decomposition method shows the lowest result latency and is memory-efficient since the steering factor multiplication matrix may be distributed across the entire array. However, it also exhibits the worst parallel efficiency and may be difficult to restructure in the event of node failures. The pipelined angle decomposition improves the efficiency of angle decomposition at the cost of a longer result latency and maintains the memory efficiency of the angle-decomposition algorithm. However, it also suffers from difficulty in supporting fault-tolerant procedures. Each of the parallel beamforming algorithms presented is scalable to different degrees for large arrays, as was shown by sensitivity analyses with the aid of a rapid virtual prototyping tool.

Future directions from this research will focus on leveraging the contributions from these new parallel algorithms for in-array processing to support additional beamformers with increasing sophistication in their acoustic and signal processing attributes. Currently, extensions under development include parallel algorithms for split-aperture beamforming and MVDR adaptive beamforming. Furthermore, research on the implementation of fault-tolerance mechanisms to support the three parallel algorithms is underway.

2.8. References

- [1] S. Banerjee and P.M. Chau, Implementation of Adaptive Beamforming Algorithms on Parallel Processing DSP Networks, *Proceedings of SPIE – The International Society for Optical Engineering*, 1770 (1992) 86-97.
- [2] P.M. Clarkson, *Optimal and Adaptive Signal Processing*, (CRC Press, Ann Arbor, 1993).
- [3] R.F. Dwyer, Automated Real-Time Active Sonar Decision Making By Exploiting Massive Parallelism, *IEEE Oceans Conference Record*, 3 (1996) 1193-1196.
- [4] M.J. Flynn, Some Computer Organizations and Their Effectiveness, *IEEE Transactions on Computers*, 21 (9) (1972) 948-960.
- [5] O.L. Frost III, An algorithm for linearly constrained adaptive array processing, *Proceedings of IEEE*, 60 (1972) 926-935.
- [6] A. George, R. Fogarty, J. Markwell, and M. Miars, An Integrated Simulation Environment for Parallel and Distributed System Prototyping, *Simulation*, submitted July 1998.
- [7] S. Haykin, *Array Signal Processing*, (Prentice-Hall, 1985).
- [8] K.M. Houston, A Fast Beamforming Algorithm, *IEEE Oceans Conference Record*, 1 (1994) 211-216.
- [9] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, (McGraw-Hill, New York 1993)
- [10] W.C. Knight, R.G. Pridham, and S.M. Kay, Digital Signal Processing for Sonar, *Proceedings of the IEEE*, 69 (11) (November 1981) 1451-1506.
- [11] F. Machell, Algorithms for broadband processing and display, *Applied Research Laboratories Technical Letter No. 90-8* (ARL-TL-EV-90-8), Applied Research Laboratories, The University of Texas at Austin, 1990.
- [12] D. McMahon, A. Bolton, Signal-to-Noise Ratio Enhancement of Cyclic Summation, *Proceedings of the International Symposium on Signal Processing and its Applications (ISSPA)*, 2 (1996) 710-713.
- [13] Message-Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, (U. of Tennessee, Knoxville, TN, May 1994).

- [14] D. Morgan, Practical DSP Modeling, Techniques, and Programming in C (Wiley, New York, 1994).
- [15] R.A. Mucci, A Comparison of Efficient Beamforming Algorithms, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-32 (3) (June 1984) 548-558.
- [16] R. Nielsen, Sonar Signal Processing, (Artech House, Boston, 1991).
- [17] J. Salinas and W.R. Bernecky, Real-time Sonar Beamforming on a MasPar Architecture, *IEEE Symposium on Parallel and Distributed Processing - Proceedings*, (1996) 226-229.
- [18] W.W. Smith and J.M. Smith, Handbook of Real-Time Fast Fourier Transforms: Algorithms to Product Testing, (IEEE Press, New York, 1995).
- [19] W. Robertson and W. J. Phillips, A System of Systolic Modules for the MUSIC Algorithm, *IEEE Transactions on Signal Processing*, 41 (2) (1991) 2524-2534.
- [20] W. Stallings, Data and Computer Communications (Macmillan, New York, 1994).
- [21] C.R. Ward, P.J. Hargrave, and J.G. McWhirter, A Novel Algorithm and Architecture for Adaptive Digital Beamforming, *IEEE Transactions on Antennas and Propagation*, AP-34 (March 1986) 338-346.
- [22] G.P. Zvara, Real Time Time-Frequency Active Sonar Processing: A SIMD Approach, *IEEE Journal of Oceanic Engineering*, 18 (October 1993) 520-528.

3. PARALLEL ALGORITHMS FOR SPLIT-APERTURE CONVENTIONAL BEAMFORMING

Quiet submarine threats and high clutter in the littoral undersea environment increase the processing demands on beamforming arrays, particularly for applications which require in-array autonomous operation. Whereas traditional single-aperture beamforming approaches may falter, the Split-Aperture Conventional Beamforming (SA-CBF) algorithm can be used to meet stringent requirements for more precise bearing estimation. Moreover, by coupling each transducer node with a microprocessor, parallel processing of the split-aperture beamformer on a distributed system can glean advantages in execution speed, fault tolerance, scalability, and cost. In this chapter, parallel algorithms for SA-CBF are introduced using coarse-grained and medium-grained forms of decomposition. Performance results from parallel and sequential algorithms are presented using a distributed system testbed comprised of a cluster of workstations connected by a high-speed network. The execution times, parallel efficiencies, and memory requirements of each parallel algorithm are presented and analyzed. The results of these analyses demonstrate that parallel in-array processing holds the potential to meet the needs of future advanced sonar beamforming algorithms in a scalable fashion.

3.1. Introduction

Sonar beamforming is a class of array processing that optimizes an array gain in a direction of interest to detect the movement of hulls and propellers in water, which create signals over a wide frequency band. The determination of the direction of arrival relies on the detection of the time delay of the signal between sensors. Incoming signals are steered by complex-number vectors. If the beamformer is properly steered to an incoming signal, the multi-channel input signals will be amplified coherently, maximizing power in the beamformed output; otherwise, the output of the beamformer is attenuated to some degree. Thus, peak points in the beamforming output indicate directions of arrival for sources.

Performance of a beamformer depends on several factors. One of the important elements of concern is node configuration. Previous research has shown that uniform sensor spacing is not the best choice from the point of view of minimizing bearing error [1,2]. Cramer-Rao lower bound (CRLB) analysis, which is used to set an absolute lower bound on the bearing error, determines the optimum positioning for the sensors of a linear array in order to obtain optimum estimates for direction of arrival. By using split-aperture conventional beamforming (SA-CBF) on linear arrays, the lower bound for bearing error as estimated by CRLB analysis can be approached, yielding more improved bearing estimation than single-aperture arrays.

As advancements in acoustics and signal processing continue to result in beamforming algorithms better able to cope with quiet sources and cluttered environments, the computational requirements of the algorithms also rise, in some cases at a pace exceeding that of conventional processor performance. Moreover, as the number of sensors increases, so too does the problem size associated with these algorithms. To implement modern beamforming algorithms in real-time, considerable processing power is necessary to cope with these demands. A beamformer based on a single front-end processor may prove insufficient as these computational demands increase; thus, a number of parallel approaches have been proposed in order to overcome the limits of a single high-end processor in beamforming applications. Several projects from the Naval Undersea Warfare Center (NUWC) involve the development of real-time sonar systems by exploiting massive parallelism. The delay-and-sum beamformer has been mapped by Salinas and

Bernecky to a MasPar Single Instruction Multiple Data (SIMD) architecture [3]. The MasPar machine was also used by Dwyer to develop a real-time active beamforming system [4]. Zvara built a similar system on the Connection Machine CM 200 architecture [5]. Other work has concentrated on a variety of adaptive algorithms on such parallel systems as systolic arrays, SIMD multiprocessors, and DSP multicomputers [6,7]. The work presented in this paper extends this knowledge base of parallel beamforming, particularly with respect to split-aperture sonar processing for in-array beamforming. An in-depth performance analysis of the stages of a sequential version of SA-CBF is shown in order to examine the sequential bottlenecks inherent in the system. In addition, new parallel algorithms for SA-CBF are designed for use with intelligent distributed processing arrays, and their performance is analyzed.

Most of the computations in beamforming consist of vector and matrix operations with complex numbers. The regularity in the patterns of these calculations simplifies the parallelization of the algorithms. Two parallel versions of SA-CBF have been developed: iteration-decomposition and angle-decomposition techniques. Iteration decomposition, which is a form of control parallelism, is a coarse-grained scheduling algorithm. An iteration is defined as one complete loop through the beamform algorithm. A virtual front-end processor collects the input data set from each sensor. This virtual front-end then proceeds to execute a complete beamforming algorithm independently of the operation of the other nodes. Other processors are concurrently executing the beamforming algorithm with different data sets collected at different times. Angle decomposition, a form of data parallelism, is a medium-grained scheduling algorithm in which different steering angle jobs for the same data set are assigned to different processors. Application of these methods to the SA-CBF algorithm extends the original development of algorithms for control and domain decomposition in traditional single-aperture beamforming [8].

A theoretical background of SA-CBF is presented in Section 3.2 with a focus on digital signal processing. A sequential version of the SA-CBF algorithm is given in Section 3.3. Two different memory models of the sequential algorithm are shown to determine a baseline for the parallel algorithms. In Section 3.4, two parallel SA-CBF algorithms are presented. In Section 3.5, the performance of the parallel SA-CBF algorithms, in terms of execution time, speedup, efficiency, and memory requirements are shown. Finally, a summary of the strengths and weaknesses of the algorithms and a discussion of future research are presented in Section 3.6.

3.2. Overview of Split-Aperture Beamforming

SA-CBF is based on single-aperture conventional beamforming in the frequency domain. The beamforming array is logically divided into two sub-arrays [9,10,11]. Each sub-array independently performs conventional frequency-domain beamforming using replica vectors on its own data. The two sub-array beamforming outputs are cross-correlated to detect the time delay of the signal for each steering angle. The cross-correlated data, with knowledge of the steering angles and several other parameters, will map the final beamforming output.

Unlike the single-aperture beamforming algorithm, the SA-CBF algorithm does not need to steer at every individual desired angle in the steering stage. The cross-correlation creates some redundant information between the adjacent sub-array steering angles. By cross-correlating to time delays slightly offset from the sub-array steering delays, each sub-array steering angle can be used to generate a range of the time delay plot. The discrete cross-correlation function is defined in Eq. 3.1:

$$c_{xy}(n) = \frac{1}{N} \sum_{i=0}^{N-1} x(i)y(i+n) \quad \xleftrightarrow{\text{Fourier Transform}} \quad C_{xy}(k) = X(k)Y(k)^* \quad (\text{Eq. 3.1})$$

$$n = 0, 1, \dots, N-1 \quad k = 0, 1, \dots, 2N-2$$

where vectors x and y are the sub-array beamforming outputs, N is the number of samples in x and y , and operator $*$ indicates complex conjugation. We are only interested in the small number of angles or time delays in the cross-correlation adjacent to the beamforming angle of the sub-array.

Before the inverse Fourier transform is applied to obtain the cross-correlation as a function of time delay, the Smoothed Coherent Transform (SCOT) is used to prefilter the cross-correlation. The spectral whitening obtained by SCOT results in an improved signal-to-interference ratio and an enhancement of the correlation between the two sub-arrays. Additional advantages of SCOT, as well as other prefiltering techniques, can be found in Ferguson [12]. The SCOT weighting function is given by Eq. 3.2.

$$|W(k)|^2 = \frac{1}{\sqrt{|X(k)|^2 |Y(k)|^2}} \quad (\text{Eq. 3.2})$$

SCOT is accomplished either by taking the instantaneous magnitude of the cross-correlation in frequency or a running average of the magnitude.

As mentioned previously, each beam formed by the sub-arrays can be used to calculate multiple points in plotting the output bearing. The process by which this increase in resolution is accomplished is called τ -interpolation. For each output angle, τ -interpolation works with only the two cross-correlation results that are nearest to the desired angle. Raised-cosine weights are used to calculate the beamforming output for the interpolated angle from a linear combination of the two adjacent cross-correlation values. Because we need only a limited range of cross-correlation values to calculate the final beamforming output, the inverse discrete Fourier transform is then performed using Eq. 3.3 rather than the conventional FFT algorithm. This method will decrease computational cost of the inverse Fourier transform stage.

$$c(\tau_j) = \frac{1}{N_{FFT}} \sum_{m=M_1}^{M_2} 2 \text{Re}[\tilde{C}_m e^{j2\pi(m-1)\Delta f \tau_j}] \quad (\text{Eq. 3.3})$$

In Eq. 3.3, M_1 and M_2 are the minimum and maximum frequency bin numbers in which we are interested, \tilde{C}_m is the weighted and normalized frequency-domain cross-correlation, Δf is the frequency resolution of the FFT, τ_j is the time delay between phase centers, and $\text{Re}()$ is the function that returns the real value of a complex number.

The two nearest cross-correlation vectors, subscripted as $c_L(\tau_j)$ and $c_R(\tau_j)$, are used to evaluate each interpolated output angle via τ -interpolation as defined in Eq. 3.4 and Eq. 3.5.

$$c(\theta_{out}) = h_L c_L(\tau_j) + h_R c_R(\tau_j) \quad (\text{Eq. 3.4})$$

$$h_L = \frac{1}{2} \left[1 + \cos \pi \left(\frac{\theta_L - \theta_{out}}{\theta_L - \theta_R} \right) \right]; \quad h_R = 1 - h_L \quad (\text{Eq. 3.5})$$

In these equations, θ_L and θ_R are the sub-array beamformed angles to the left and right, respectively, of the output interpolated angle θ_{out} , and $c_L(\tau_j)$ and $c_R(\tau_j)$ are cross-correlation values

for angles θ_L and θ_R , respectively. The final output of the SA-CBF beamformer finds $c(\theta_{out})$ versus each interpolated θ_{out} . Further information on the design of the sequential SA-CBF algorithm can be found in Machell [10] and the experimental evaluation of SA-CBF can be found in Stergiopoulos¹¹. To consolidate the above processes, Fig. 3.1a shows the block diagram of the SA-CBF algorithm, and Fig. 3.1b is a sample output of the SA-CBF.

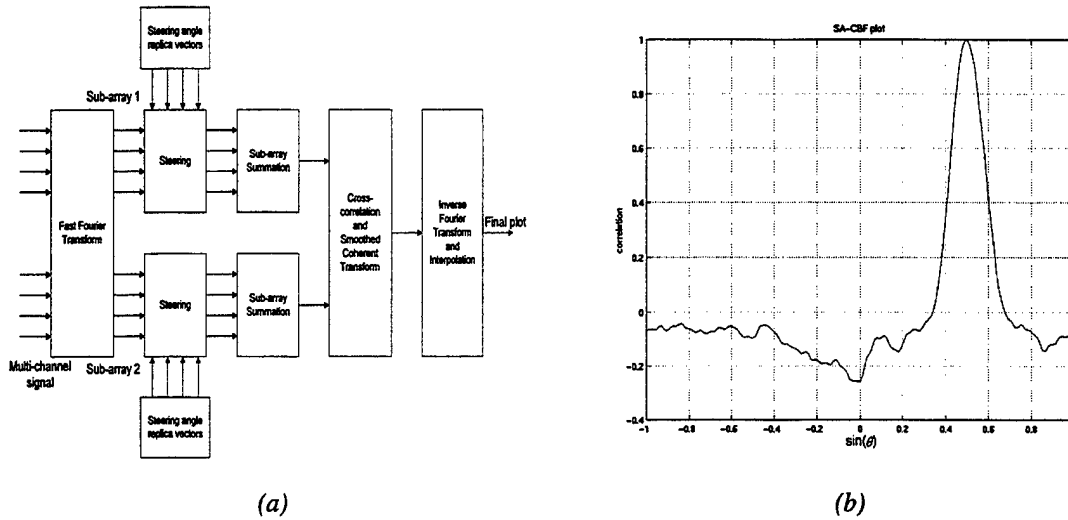


Fig. 3.1 Block diagram of the sequential SA-CBF algorithm for 8 nodes (a), and a sample output of the SA-CBF for 8 nodes with a source at $\theta = 30^\circ$ (b). Arrows in the block diagram indicate data stream vectors.

It is often highly desirable to increase the number of input nodes and the number of steering angles in a beamforming system. By increasing the number of input sensors, the beamformer resolves finer angles because the main lobe of the beam pattern narrows and the side lobes of the beam pattern decrease. Fig. 3.2a, which is constructed using a polynomial representation of a linear array [13], shows this result. Even if there are enough nodes to obtain a sharp beam pattern, the number of steering angles remains an important factor in producing high-resolution beamforming output. The number of steering angles decides the number of output points of the beamformer; thus, a small number of steering angles may cause a blurry beamforming output.

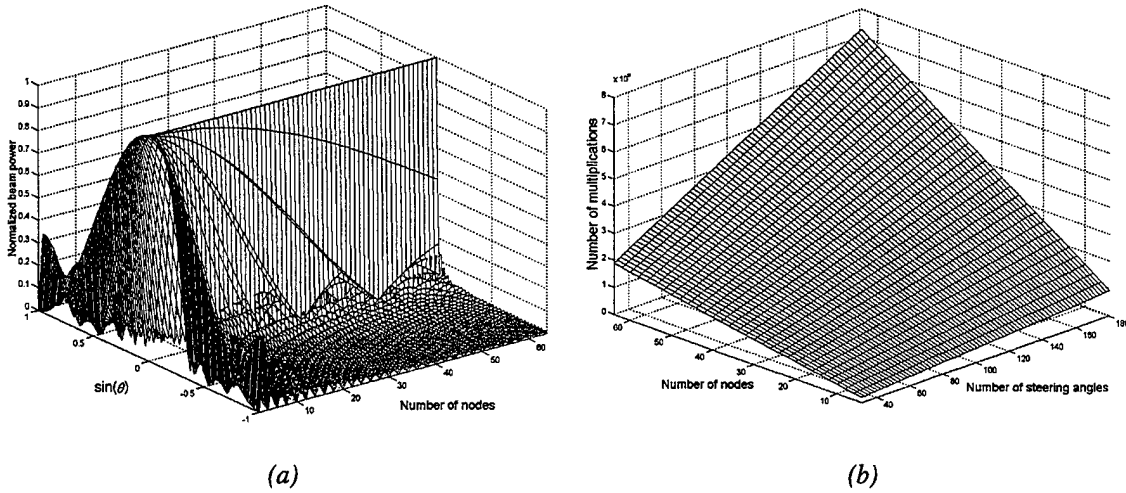


Fig. 3.2 Normalized beam pattern of the SA-CBF as a function of the number of nodes and $\sin(\theta)$ (a), and the required number of multiplication operations as a function of the number of steering angles and the number of nodes (b).

As both of these parameters increase, the number of multiplication operations required to generate beamforming output is increased rapidly, as shown in Fig. 3.2b. According to this figure, powerful processing is essential to generate high-resolution beamforming output with acceptable latency and throughput. In cases where current technology cannot provide sufficient real-time performance in a single processor, a trade-off will be required between response time and resolution. The scalable performance of parallel processing will help to overcome limits imposed by a single front-end processor.

3.3. Performance Analysis of Sequential Split-Aperture Beamformers

The SA-CBF algorithm previously discussed can be split into five distinct stages: *FFT*, *Steering*, *Sub-array Summation*, *Cross-correlation/SCOT*, and *Inverse Fourier Transform/Interpolation*. High-level pseudo-code for SA-CBF is shown in Fig. 3.3. Up to and including the *sub-array summation*, SA-CBF is a frequency-domain conventional beamforming algorithm except that there are two phase centers. The succeeding stages improve the visual resolution and bearing estimation of the beamformer output using DSP techniques.

```

Do FFT for every incoming signal vector;
For j=1, number of steering angles
  Steering(j);
  Sub-array summation(j);
  Cross-correlation and SCOT(j);
End
For k=1, number of output angles
  Inverse Fourier transform and interpolation(k);
End

```

Fig. 3.3 Pseudo-code for the sequential SA-CBF algorithm.

In building a baseline for the parallel SA-CBF algorithms, two different sequential implementations were created, one using a minimum-memory model and the other using a minimum-calculation model. The minimum-memory model tries to save memory by doing more calculations, and the minimum-calculation model saves redundant calculations by using more memory. To illustrate the trade-off involved in selecting one model over the other, consider the steering vectors and the inverse Fourier transform basis. Under normal operation, these vectors are not subject to change from iteration to iteration. The minimum-memory model computes these space-consuming vectors on the fly for every iteration. However, in the minimum-calculation model, the vectors are already calculated in an initial phase and saved into special memory locations to access easily whenever needed without recalculation. Thus, the minimum-calculation model compromises memory space for faster execution time. In the event of node failures, the minimum-calculation model will be forced to recalculate all values in these vectors. Node failure increases the distance between nodes so new steering vectors need to be established based on new parameters. A new inverse Fourier transform basis is also necessary when the processing frequency bins are changed. By contrast, the minimum-memory model encounters significantly fewer disturbances in the event of failures since it would have recalculated all values in any case.

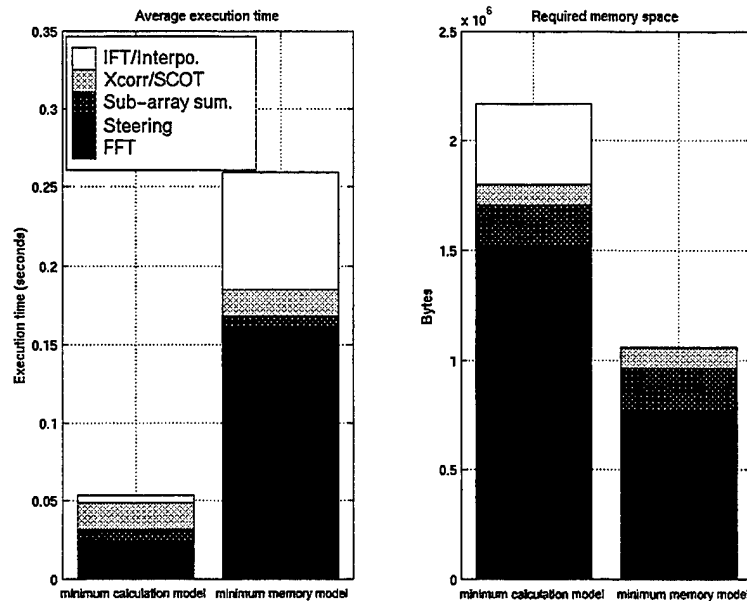


Fig. 3.4 Average execution time per iteration for 1000 iterations, and required memory space, for the SA-CBF with 16 nodes, 45 steering angles, 177 output angles, and 4-byte floating-point values.

Due to the recalculation in the *steering* and *inverse Fourier transform* stages, execution times of these stages are severely increased for the minimum-memory model. Conversely, required memory space for these stages is significantly smaller than that required in the minimum-calculation model. Experiments were conducted to examine the execution time and memory requirements of the two sequential models. The platform used was a SPARCstation-20 workstation with an 85MHz SuperSPARC-II processor and 64MB of memory, and running Solaris 2.5. The experimental results in Fig. 3.4 show that execution time of the minimum-calculation model is five times less than that of the minimum-memory model with twice the memory required. The execution time and memory requirement of the *FFT* stage are the same

for both models due to the fact that the same FFT algorithm is used in both. The *sub-array summation* and *cross-correlation/SCOT* stages are also unaffected by the model since these stages have no vectors that are the same from iteration to iteration.

The model selected for the baseline of the performance analysis of parallel algorithms depends on the focus of the study. In the next section, to estimate attainable speedup with each of the parallel programs, execution time is the most important factor to be measured. Therefore, the minimum-calculation model is preferred as the sequential baseline. All parallel algorithms are implemented with the same minimum-calculation model, and it is assumed that each processor has sufficient memory to hold the program and all data.

3.4. Parallel Algorithms for Split-Aperture Beamformers

The parallel algorithms in this paper were designed to operate in conjunction with a distributed, parallel, sonar system architecture. This architecture is composed of intelligent nodes connected by a network. Each of the smart nodes, comprised of a hydrophone and a microprocessor, has its own processing power as well as requisite data collection and communication capability. By using such a distributed array architecture, the algorithmic workload is distributed and cost is reduced due to the elimination of an expensive front-end processor. Such an architecture ties the degree of parallelism (DOP) to the number of physical nodes in the target system. Of course, an increase in the number of nodes will increase the amount of input data and thus the problem size.

The best performance of a parallelized task is achieved by minimizing processor stalling and communication overhead between processors. With a homogeneous cluster of processors, dividing tasks evenly among processors serves to maximize performance by reducing these hazards. While task-based parallelism is possible with the SA-CBF algorithm (via assigning *steering* to one node, *sub-array summation* to another node, etc.), the workload would not be homogeneous and would result in degraded performance. Fig. 3.5a shows this unbalanced workload amongst the various sequential tasks and serves as a justification for not using task-based parallelism.

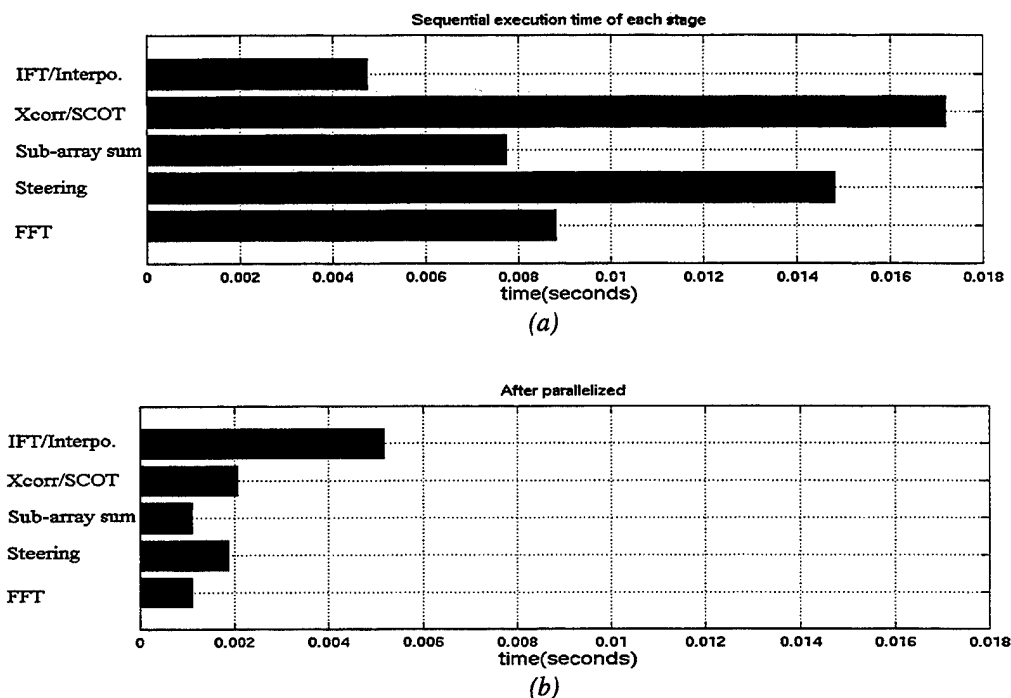


Fig. 3.5 Averaged execution time comparison between sequential (a) and partially parallelized (b) programs for SA-CBF in an 8-node configuration.

The SA-CBF algorithm computes many vector and matrix operations using nested loops. Therefore, we can partition iterations of the outer loop across the processors to create a balanced workload for each processor. When loop partitioning is applied to the parallel SA-CBF algorithm, special attention is required because there are two external loops that are repeated: number of steering angles and number of output angles, as shown in Fig. 3.3. These loops run separately within the SA-CBF algorithm but their information is tightly coupled and must be used together to generate the final beamforming output. If we parallelize only the first loop and not the second, which includes the *inverse Fourier transform* and *interpolation* stages, severe performance slowdown would result as the number of processors increases. Fig. 3.5b shows how loop partitioning of the stages in the first loop results in the second loop becoming a bottleneck. As the number of processors increases, the execution time of each stage decreases linearly except that of the *IFT/Interpolation* stage since this stage is not parallelized. Since the total execution time of the beamformer in this figure is found by summing the execution times of the stages, it becomes clear that the execution time of the *IFT/Interpolation* stage will become increasingly dominant as the number of processors increases. Of course, according to Amdahl's law, a small number of sequential operations can significantly limit the speedup achievable by a parallel program [14]. Thus, in this case the sequential bottleneck caused by the *IFT/Interpolation* stage limits the speedup to no more than 5 for an 8-node configuration or 9 for a 32-node configuration. Though parallelization of the *IFT/Interpolation* stage is nontrivial due to strong dependencies with previous stages, it will significantly increase the efficiency of the overall algorithm.

The two parallel algorithms presented in the rest of this section make use of loop partitioning in two different ways. Furthermore, these algorithms parallelize the second loop (i.e., the *IFT/Interpolation* stage) to achieve better efficiency. The next two subsections present an overview of the two parallel algorithms, followed by performance results in Section 3.5.

3.4.1. Iteration-decomposition method

The first decomposition method involves the partitioning of iterations, the solutions of a complete beamform cycle. Iteration decomposition is a technique whereby multiple beamforming iterations, each operating on a different set of array input samples, are overlapped in execution by pipelining. The algorithm follows the tradition of overlapped concurrent execution pipelining, where one operation does not need to be completed before the next operation is started. The beamforming task for a given sample set is associated with a single node in the parallel system. Other nodes work concurrently on other iterations. Pipelining is achieved by allowing nodes to collect new data from the sensors and begin a new iteration before the current iteration is completed. At the beginning of each iteration, all nodes stop processing the beamforming iterations assigned them just long enough to execute the FFT on their own newly collected samples and send the results to the node assigned the new iteration. Once this data has been sent, all nodes resume the processing of their respective iterations. Using this pipelining procedure, there are as many iterations currently being computed as there are processors in the system, each at a different stage of completion. A block diagram illustrating this algorithm in operation on a 4-node array is shown in Fig. 3.6.

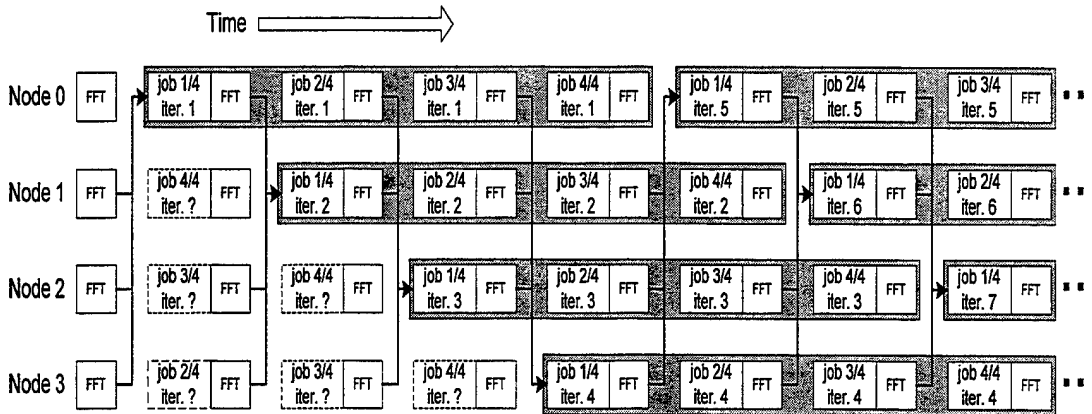


Fig. 3.6 Block diagram of the iteration-decomposition algorithm in a 4-node configuration. Solid arrows indicate inter-processor, all-to-one communication and shaded boxes represent independent complete beamforming cycles (i.e. iterations).

Individual beamforming is separated by inter-processor communication stages and each processor takes responsibility for a different beamforming job. A number of difficulties exist for iteration decomposition. First, since every partial job is synchronized at the communication points, an unbalanced processing load can develop across nodes, which may lead to processor stalling. Second, each iteration of the beamforming algorithm must be completed by a node before its pipeline cycle is complete so as to avoid collision between jobs. Therefore, to maximize the performance of the iteration-decomposition algorithm, the beamforming jobs should be evenly segmented by the number of processors. The pseudo-code illustrating the basic algorithm followed by each processor is shown in Fig. 3.7.

```

For t=1, Total number of iterations
  For j=1, number of processors //beamforming is divided by # of processors
    index=(j+my_rank)%(number of processors); //my_rank is node number
    Do FFT for their own node data;
    Communicate with other nodes;
    For k=start_steering_angle(index), end_steering_angle(index)
      Steering(k);
      Sub-array summation(k);
      Cross-correlation and SCOT(k);
    End
    For i=start_output_angle(index), end_output_angle(index)
      Inverse Fourier transform and interpolation(i);
    End
  End
End

```

Fig. 3.7 Pseudo-code for the iteration-decomposition algorithm.

Each processor calculates an index based on its node number, the current job number, and the number of nodes. This index is used to access arrays which tell the node from which point in its iteration it must continue after executing the *FFT* and communicating new data and at which point it must again pause in order to begin another new iteration. Specifically, arrays containing the starting steering angle and ending steering angle for a computation block instruct the node on how to partition the first of the two loops. Arrays containing the starting output angle and ending output angle are used to decompose the second loop. Upon completion of this procedure, a new iteration is started, and each node calculates a new portion of the necessary steering and output angles for its iteration. Managing these arrays requires a nontrivial amount of overhead, but such overhead is a necessary part of the correct operation of the pipelining method.

3.4.2. Angle-decomposition method

The second parallel algorithm decomposes SA-CBF using a medium-grained approach in which the internals of a complete beamforming iteration are segmented. The angle-decomposition algorithm distributes processing load by decomposing the domain, the steering angles. Each node calculates the SA-CBF results for a certain number of desired steering directions from the same sample set. Before doing so, all participating nodes must have a copy of the data from all other nodes. After completing this all-to-all communication, each node computes different beamforming angles for the same data. This algorithm introduces considerably more communication than the iteration-decomposition algorithm, and the interconnection scheme between processors will have a more significant effect on performance. The communication requirements are further studied in George [8]. A block diagram illustrating this algorithm in operation on a 4-node array is shown in Fig. 8.

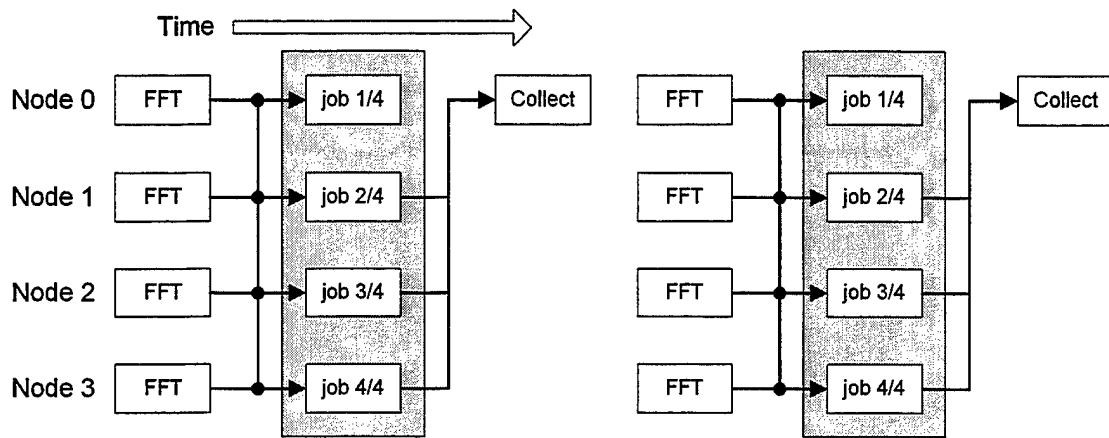


Fig. 3.8 Block diagram of the angle-decomposition algorithm in a 4-node configuration. Solid arrows indicate inter-processor communication and shaded boxes represent independent beamforming jobs.

For the purposes of decomposing the two loops, four variables that indicate beginning and ending steering and output angles are calculated in an initial phase. These four variables serve much the same purpose as the arrays in the iteration-decomposition method, though for angle decomposition these values are a function only of the node number. The steering direction and the output angle on which a node is to begin computing are determined by that node's relative location from a virtual front-end node. The number of steering directions a node is to compute is based on dividing the total number of desired steering directions by the number of nodes. The number of output angles per node is also derived from the steering direction information. Fig. 3.9 shows the pseudo-code for this approach. After a node is finished computing the results for its steering directions, it must communicate them to a specially designated node for final collection. This collection node can be fixed or, to provide fault tolerance, free-floating perhaps via round-robin scheduling. Although this algorithm involves a more complex communication mechanism best served with a broadcasting network, it does not require the additional overhead necessary to manage pipelining as does the iteration-decomposition method.

```

Calculate angle information based on the node number
For t=1, Total iteration number
  Do FFT for their own node data;
  Communicate with other nodes;
  For k=my_start_steering_angle, my_end_steering_angle
    Steering(k);
    Sub-array summation(k);
    Cross-correlation and SCOT(k);
  End
  For i=my_start_output_angle, my_end_output_angle
    Inverse Fourier transform and interpolation(i);
  End
  Collect result from other nodes;
End

```

Fig. 3.9 Pseudo-code for the angle-decomposition algorithm.

3.5. Performance Analysis of Split-Aperture Beamformers

In order to understand the strengths and weaknesses of the two parallel algorithms, their performance characteristics were measured on a physical multicomputer testbed. The results of these experiments are presented in this section. The testbed used consists of a cluster of SPARCstation-20 workstations connected by a 155-Mbps (OC-3c) Asynchronous Transfer Mode (ATM) network.

The algorithms were implemented via message-passing parallel programs written in C-MPI (Message-Passing Interface) [15]. MPI is a library of functions and macros for message-passing communication and synchronization that can be used in C/C++ and FORTRAN programs. In the program code, we call a time-check function at the beginning of a stage and save the return value. After each stage we call the function again, and subtract the earlier return value from the new return value. The difference is the execution time of the stage. In order to obtain reasonable and reliable results, all parallel and sequential experiments were performed 500 times and execution times were averaged.

The first experiment involves the execution of the sequential SA-CBF algorithm on a single workstation, where the number of sensors is varied to study the effects of problem size. The results are shown in Fig. 3.10. Execution times of the *FFT*, *steering*, and *sub-array summation* stages increase linearly with an increase in sensors. The number of sensors determines the number of data stream vectors in Fig. 3.1a; therefore, the processing load of these stages is greater with increased numbers of sensors. Conversely, the number of data stream vectors remains constant after the *sub-array summation* stage. No matter how many data stream vectors enter the *sub-array summation* stage, the number of output data stream vectors is always two. Furthermore, after *interpolation*, only one data stream vector is left. Thus, the execution times of *Xcorr/SCOT* and *IFT/Interpolation* stages remain fixed as the number of sensors is increased.

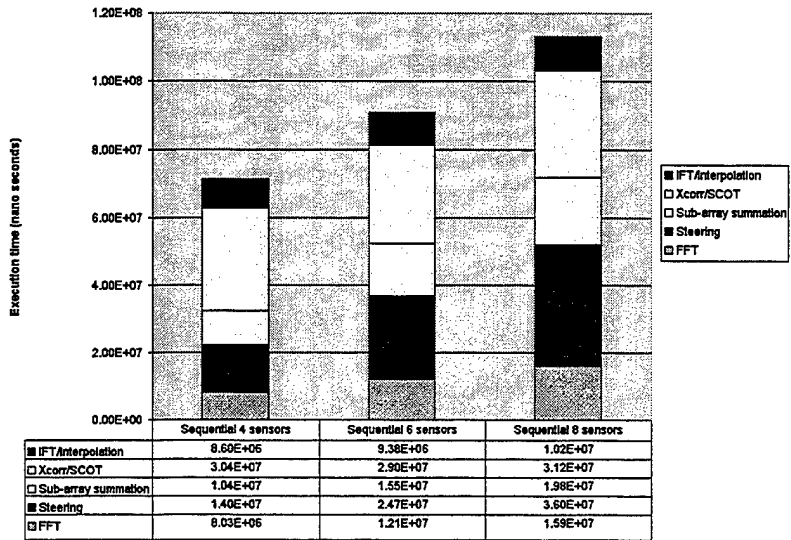


Fig. 3.10 Average execution time per iteration as a function of array size for the sequential SA-CBF algorithm with 500 iterations on the SPARC20/ATM cluster.

For each of the two parallel algorithms, Fig. 3.11 shows average execution times for three system sizes: 4, 6, and 8 nodes. For iteration decomposition, the execution time shown represents the effective execution time and not the result latency. As was previously shown for a 4-node configuration, the results for a given beamforming cycle are output after four pipeline stages. In fact, as is typical of all pipelines due to the overhead incurred, this result latency is longer than the total execution time of the sequential algorithm. Instead, the figure plots the effective execution time, which represents the amount of time between outputs from successive iterations once the pipeline has filled. In the angle-decomposition case, we can measure the execution time directly because there is no overlap of beamforming jobs.

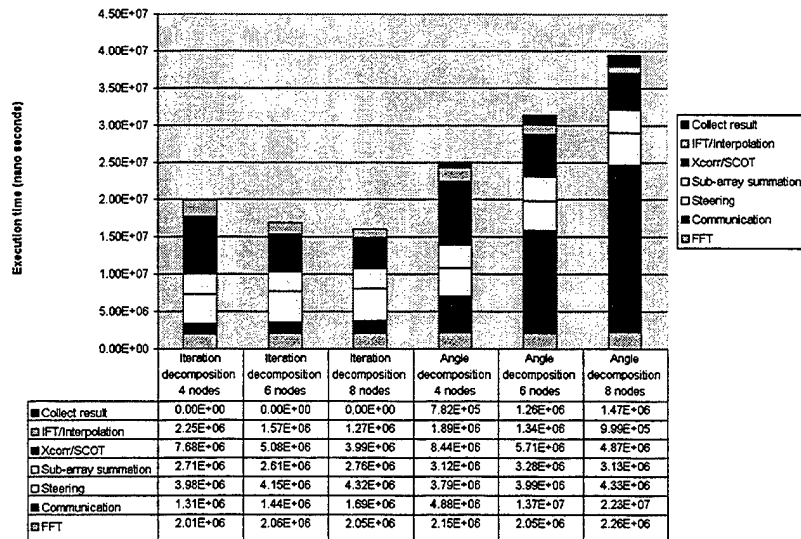


Fig. 3.11 Average execution time per iteration as a function of array size for the parallel SA-CBF algorithms with 500 iterations on the SPARC20/ATM cluster

As seen in Fig. 3.11, the execution times of the *FFT*, *steering*, and *sub-array summation* stages do not change significantly as the number of nodes increases. As mentioned earlier, the distributed sonar architecture uses smart nodes at each input sensor; therefore, the number of data stream vectors is identical to the number of processors. The additional workload caused by increasing the number of nodes is evenly distributed across the processors. Therefore, the number of nodes does not influence the execution time of these stages. However, the execution times for the *Xcorr/SCOT* and *IFT/Interpolation* stages decrease as the number of nodes and processors increase. In these stages, each processor does less work as the number of nodes increases because the workload for a fixed number of data stream vectors is divided among the processors. In spite of a growing problem size, the overall computation times (i.e. execution time minus communication time) of both decomposition algorithms decline as the number of nodes increases.

In this experiment, communication time is defined as the time spent in communication function calls such as *MPI_Send* and *MPI_Recv*. The communication pattern can be shown with the graphical profiling program Upshot [16]. A sample Upshot output is shown in Fig. 3.12, which displays a portion of the profiling log collected from both parallel algorithms running on four nodes in the testbed. In this figure, communication blocks are represented by rectangles defined in the legend, and computation blocks are represented by the horizontal lines between successive communication blocks. Iteration decomposition uses a fairly simple communication pattern but angle decomposition communicates an all-to-all message at the initial phase of each beamforming job. Communication time of iteration decomposition and collection time of angle decomposition have little contribution to total execution time. However, with angle decomposition, the size of the first data communication of each iteration increases rapidly with the number of nodes. With this increase in communication comes an increase in the MPI overhead, an increase in network contention, and poorer performance, which eventually comes to dominate the total execution time. Clearly, the relatively small amount of communication in iteration decomposition is an advantage for that algorithm. Fig. 3.11 indicates that total execution time of iteration decomposition decreases and total execution time of angle decomposition increases with an increasing number of nodes.

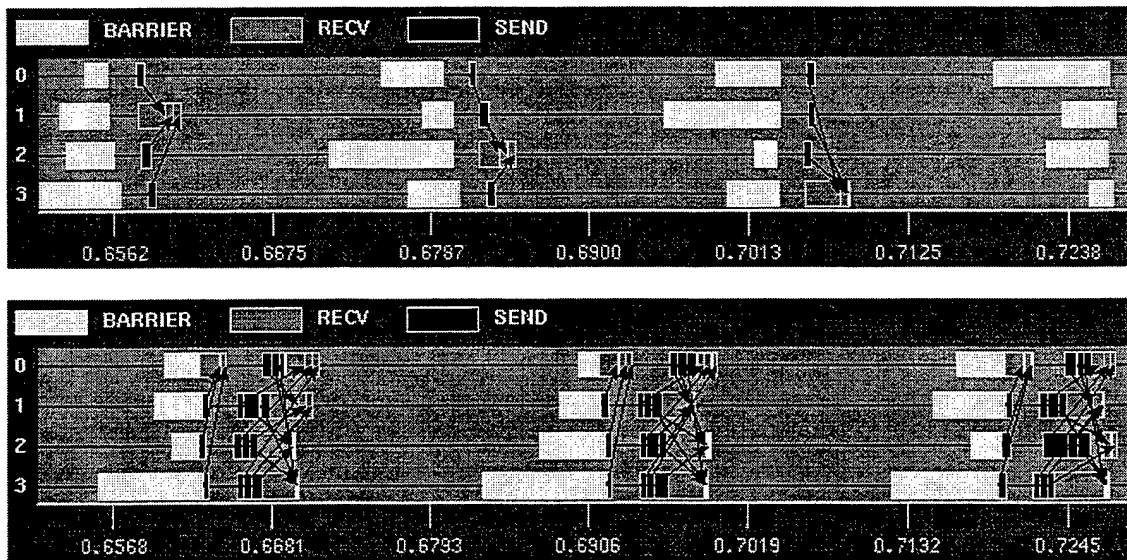


Fig. 3.12 Upshot profiles for both of the parallel SA-CBF algorithms with 4 nodes. The upper snapshot shows a profile from the iteration decomposition, while the lower snapshot shows a profile from the angle decomposition.

Fig. 3.13a shows the scaled speedup of the two decomposition methods over the testbed cluster of SPARCstation-20 workstations. The baseline for comparison is the sequential SA-CBF algorithm running on one workstation of the same testbed. Since the algorithms incur additional workload as sensors are added, the plots show scaled speedup. Fig. 3.13b displays scaled efficiency which is defined as scaled speedup divided by the number of processors used. Together these figures show the pronounced effect of the communication overhead in the angle-decomposition method, whereas iteration decomposition exhibits near-linear scaling. Improvement of the performance of angle decomposition may be achieved by employing more complex network architectures, such as broadcast-efficient networks, and by implementing more robust communication pipelining. Unfortunately, these methods may be impractical to implement on a distributed sonar array topology with limited communication capabilities.

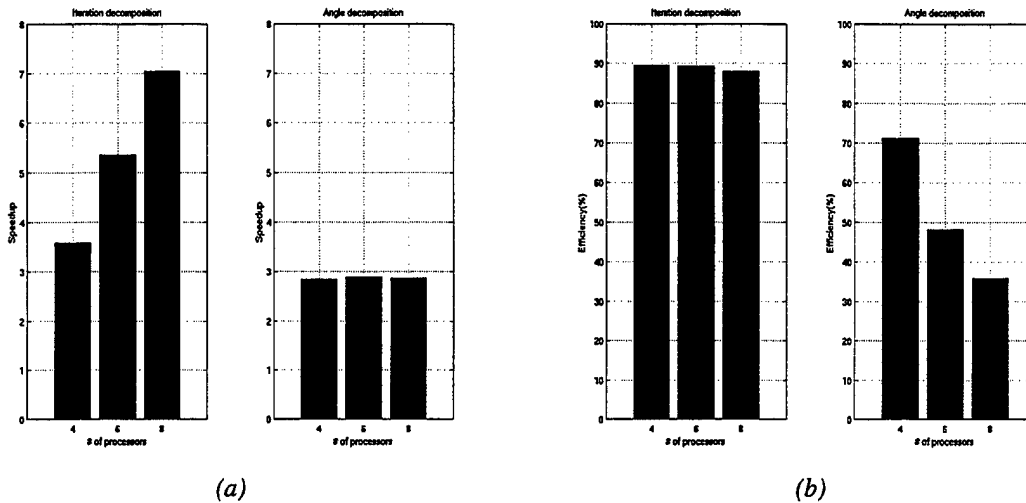


Fig. 3.13 Scaled speedup (a) and scaled efficiency (b) as a function of the number of processors for the iteration-decomposition and angle-decomposition algorithms.

In Section 3.3, we chose the minimum-calculation model as the baseline for this investigation. In this model, the majority of the memory requirement arises from the *steering* stage, as shown in Fig. 3.4. Iteration decomposition requires the full amount of steering-vector and steered-signal storage because each processor implements a whole beamforming task for an incoming data set. By contrast, angle decomposition needs only part of the memory space for steering since individual processors generate only part of the beamforming result for a given data set. For both the sequential algorithm and iteration decomposition, the demands for memory space for the *steering* stage grow linearly when the number of nodes is increased, as shown in Fig. 3.14. However, little change is observed for angle decomposition. These results illustrate the significant trade-off of execution time versus memory requirements. Despite the fact that iteration decomposition shows considerably better performance than angle decomposition, angle decomposition may be preferred when memory requirements are stringent.

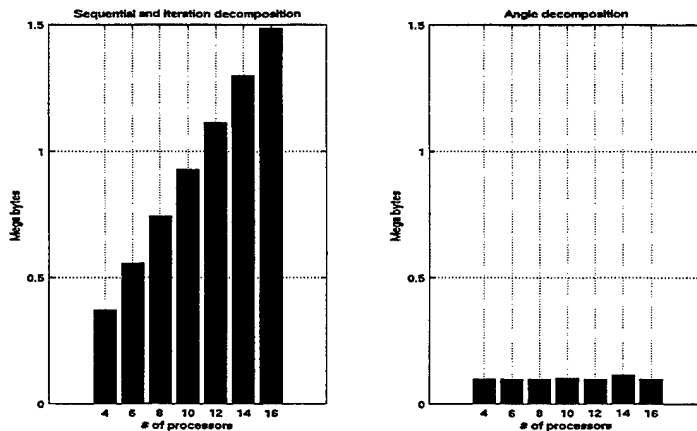


Fig. 3.14 Memory requirement of the *steering* stage as a function of the number of processors for both parallel algorithms. The memory requirements for the sequential algorithm are comparable to those of iteration decomposition.

3.6. Conclusions

The iteration-decomposition algorithm distributes its job in time space with overlap between processors, and the angle-decomposition algorithm parallelizes its job in processor space. The iteration-decomposition algorithm for SA-CBF shows more than 80 percent scaled efficiency. However, as the number of nodes is increased, the performance of the angle-decomposition method begins to worsen due to inefficiency in the communication stages. In fact, the communication time is the most significant difference between the two parallel algorithms, whereas little difference is observed in computation times. Of the two algorithms, iteration decomposition would be the better choice in architectures with enough memory for each processor to accommodate the large memory requirements. Furthermore, due to the limited amount of communication in iteration decomposition, it would also be well suited for architectures with a low-performance network. By contrast, for a system with a higher-performance network but restricted on memory capacity, angle decomposition may be the better choice. Because angle decomposition uses an all-to-all communication in each iteration, an architecture using an efficient broadcast network would also improve its performance relative to iteration decomposition.

The parallel beamforming techniques described in this paper present many opportunities for increased performance, reliability, and flexibility in a distributed parallel sonar array. These parallel methods provide considerable speedup with multiple nodes, thus enabling previously impractical algorithms to be implemented in real time. Furthermore, the fault tolerance of the sonar architecture can be increased by taking advantage of the distributed nature of these parallel algorithms and avoiding single points of failure. Future work will involve parallelizing more intricate computations, including the advanced matrix manipulations needed for adaptive beamforming algorithms and matched-field processing. With the additional complexities of these algorithms over conventional algorithms, parallel and distributed systems and software such as those described in this paper will be necessary to provide sufficient performance. The decompositions examined here not only provide important parallelization of the split-aperture algorithm, but also can serve as the foundation for these more complicated sonar signal processing algorithms.

3.7. References

- [1] G. C. Carter and E. R. Robinson, "Ocean effects on time delay estimation requiring adaptation," *IEEE J. Ocean Eng.* **18**(4), 367-378 (1993).
- [2] V. H. MacDonald and P. M. Schultheiss, "Optimum passive bearing estimation in a spatially incoherent noise environment," *J. Acoust. Soc. Am.* **46**(1), 37-43 (1969).
- [3] J. Salinas and W. R. Bernecky, "Real-time Sonar Beamforming on a MasPar Architecture," *IEEE Symposium on Parallel and Distributed Processing - Proceedings* (1996), 226-229.
- [4] R. F. Dwyer, "Automated Real-Time Active Sonar Decision Making By Exploiting Massive Parallelism," *IEEE Oceans Conference Record* **3** (1996), 1193-1196.
- [5] G. P. Zvara, "Real Time Time-Frequency Active Sonar Processing: A SIMD Approach," *IEEE J. Ocean Eng.* **18**(4), 520-528 (1993).
- [6] S. Banerjee and P. M. Chau, "Implementation of Adaptive Beamforming Algorithms on Parallel Processing DSP Networks," *Proc. of SPIE - The International Society for Optical Engineering Vol.* **1770**, 86-97.
- [7] W. Robertson and W. J. Phillips, "A system of systolic modules for the MUSIC algorithm," *IEEE Trans. Signal Processing*, **41**(2), 2524-2534 (1991).
- [8] A. D. George, J. Markwell, and R. Fogarty, "Real-time sonar beamforming on high-performance distributed computers," submitted to *Parallel Computing*, Aug. 1998.
- [9] A. D. George, R. Fogarty, J. Garcia, K. Kim, J. Markwell, M. Miars, and S. Walker, "Parallel and distributed computing architectures and algorithms for fault-tolerant sonar arrays," Annual Report, prepared for the Office of Naval Research, Arlington, Virginia, February 1998.
- [10] F. Machell, "Algorithms for broadband processing and display," ARL Technical Letter No. 90-8 (ARL-TL-EV-90-8), Applied Research Laboratories, Univ. of Texas at Austin, 1990.
- [11] S. Stergiopoulos and A. T. Ashley, "An experimental evaluation of split-beam processing as a broadband bearing estimator for line array sonar systems," *J. Acoust. Soc. Am.* **102**(6), 3556-3563 (1991).
- [12] B. G. Ferguson, "Improved time-delay estimates of underwater acoustic signals using beamforming and prefiltering techniques," *IEEE J. Ocean Eng.* **14**(3), 238-244 (1989).
- [13] D. E. N. Davies, "Independent angular steering of each zero of the directional pattern for a linear array," *IEEE trans. antennas and propagation AP-15*, 296-298 (1967).
- [14] G. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," *Proc. AFIPS 1967 Spring Joint Conf.* **30** (1967), 483-485.
- [15] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," Technical Report CS-94-230, Computer Science Dept., Univ. of Tennessee, April 1994.
- [16] W. Gropp, E. Lusk, N. Doss and A. Skjellum, "A High-performance, Portable Implementation of the MPI Message Passing Interface Standard," *Parallel Computing* **22**(6), 789-828 (1996).

4. AN INTEGRATED SIMULATION ENVIRONMENT FOR PARALLEL AND DISTRIBUTED SYSTEM PROTOTYPING

The process of designing parallel and distributed computer systems requires predicting performance in response to given workloads. The scope and interaction of applications, operating systems, communication networks, processors, and other hardware and software lead to substantial system complexity. Development of virtual prototypes in lieu of physical prototypes can result in tremendous savings, especially when created in concert with a powerful model development tool. When high-fidelity models of parallel architecture are coupled with workloads generated from real parallel application code in an execution-driven simulation, the result is a potent design and analysis tool for parallel hardware and software alike. This chapter introduces the concepts, mechanisms, and results of an Integrated Simulation Environment (ISE) that makes possible the rapid virtual prototyping and profiling of legacy and prototype parallel processing algorithms, architectures, and systems using a networked cluster of workstations. Performance results of virtual prototypes in ISE are shown to faithfully represent those of an equivalent hardware configuration, and the benefits of ISE for predicted performance comparisons are illustrated by a case study.

4.1. Introduction

Parallel and distributed systems have become a mainstay in the field of high-performance computing as the trend has shifted from using a few, very powerful processors to using many microprocessors interconnected by high-speed networks to execute increasingly demanding applications. Because there are now more processors, greater care must be taken to ensure that processors are executing code efficiently, accurately, and reliably with a minimum of overhead. This overhead appears in the form of additional coordination hardware and software that make the job of designing and evaluating new parallel systems a formidable one.

Designing high-performance parallel and distributed systems involves challenges that have increased in magnitude from traditional sequential computer design. On the hardware side, it is difficult to construct a machine that will run a particular parallel application efficiently without being an expert with that code. On the software side, it is challenging to map and tune a parallel program to run well on such a sophisticated machine. Because of the complexities involved, designers from both development viewpoints have had a growing need for design environments to address these challenges [29]. To address these difficulties, we have developed the Integrated Simulation Environment (ISE) wherein parallel and distributed architectures are simulated using a combination of high-fidelity models and existing hardware-in-the-loop (HWIL) to execute real parallel programs on virtual prototypes. ISE allows the designer to run these execution-driven simulations over networked workstations; thus, the workload can be distributed when multiple parameter sets are applied to the models. The current implementation of ISE interfaces network models created in the Block-Oriented Network Simulator (BONeS) with parallel programs written in the popular Message-Passing Interface (MPI) coordination language for C or C++.

The remainder of the paper is structured as follows. In the next section, an overview of related work is presented. In Section 4.3, an introduction to the components of ISE and how they work together is given. In Section 4.4, the methods with which ISE achieves containment of simulation explosion are described. Section 4.5 includes a discussion of validation experiments followed by an actual case study conducted with ISE, and the results are provided in Section 4.6.

Finally, conclusions about the uses and contribution of ISE to parallel and distributed computing development as well as future directions for ISE are discussed in Section 4.7.

4.2. Related Work

The ability to run real user software on simulated hardware has long been a desire for researchers and designers in the field of computing. Only in recent years have methods to achieve this goal emerged. By running sequential program code on VHDL representations of processors, the co-simulation movement (also called co-design or co-verification) took the lead in attempts to combine application software and simulated hardware. Contemporary to the co-simulation work, methods were developed to run applications on physical prototypes using reconfigurable FPGA hardware. In recent years, a new emphasis has emerged for parallel systems and their simulation. Several research institutions have created methods for running parallel or multithreaded programs over simulated shared-memory multiprocessors. Even fewer have attacked the difficulties of simulating message-passing programs over multicomputer systems, a deficiency addressed by ISE.

Co-simulation is the process by which real application code, written in a high-level programming language such as C, is fed to a processor model, written in a low-level hardware description language such as VHDL or Verilog [1]. The VHDL or Verilog simulator is in charge of measuring the simulated performance of the application on the hardware, thus aiding the hardware designer in debugging and verification of design. A subsystem connected to the simulator is in charge of executing the code and creating real data results, thus aiding the software designer. To integrate the two sides of co-simulation, Liem, et al. inject application code by including calls to that code from inside the VHDL simulation [27]. Other methods, such as those from Becker, et al. [4] and Soininen, et al. [37], create a completely separate process for the application code, which communicates its simulative needs to the VHDL or Verilog simulator via well-defined messages passed using UNIX inter-process communication. In order to make the simulation environments easier to use, such simulators have been extended to automatically generate the code needed for the interaction between the application software and the processor simulation. These systems, which include the environments from Kim, et al. [25] and Valderrama, et al. [38], allow the user to input unmodified code without the need to insert special simulator-communication calls. The co-simulation technique has proven itself so useful that numerous companies, such as ViewLOGIC [39] and Cadence, include co-simulation mechanisms in their commercial products. In addition, the Open Model Forum is working to standardize and promote a generic interface between various the simulators and their models, called the Open Model Interface (OMI) [22]. As proposed by Dunlop and McKinley [15], the first applications of this interface will focus on OMI-compliant VHDL or Verilog models and OMI-compliant C applications in order to plug and play user applications into hardware models.

Recognizing the need to simulate ever larger hardware designs while keeping simulation time down, a considerable amount of work has been spent in interfacing application software to physical rapid-prototyping systems, such as FPGA breadboards or single-board computers. Many of the same interfacing principles as co-simulation are used with the new goal of interfacing to FPGA hardware rather than interfacing to simulation models. Using physical hardware in conjunction with a simulation environment is referred to as Hardware-in-the-Loop (HWIL) simulation. Cosic's Workstation for Integrated System Design and Development [10] and the Borgatti, et al. hardware/software emulator [7] have this capability. Furthermore, some simulation systems, such as those from Bishop and Loucks [6], Kim, et al. [25], and Le, et al. [26], allow portions of the application to run on reconfigurable hardware while at the same time allowing other portions to run on a VHDL or Verilog simulator. Such systems can provide the

board or chip designer with considerable testing and verification capabilities during the design process.

All of these environment tools strive to integrate software with microchip or circuit board simulation and emulation. However, more recent attention has been paid to simulation of full-fledged parallel computer systems with parallel application software. The challenges associated with managing all the components of such a large simulation are numerous. For example, each parallel application is comprised of multiple autonomous processes, and each must be correctly synchronized with the simulation and must be able to communicate real data through the simulation and to the other processes. Though not a parallel system simulator, the end-to-end system from Huynh and Titrud [21] provides useful insight into methods used to interface several distributed processes into a single simulation.

The first forays into parallel and distributed system simulation with real applications involved the simulation of shared-memory multiprocessors. Dahlgren presents a taxonomy of methods for injecting traffic into network models [11]. Among the methods, real application-driven generators provide the best realism and fidelity. Dahlgren's simulator uses clock-driven processor models for each node and an event-driven interconnect between nodes. During execution, memory references in the processor to remote locations are trapped by the environment and used to set events in the network model. The network model can be oblivious to the actions of the processor model during the code blocks between these remote accesses. Other environments obtain the code-block times by augmenting the application assembly code with instruction-counting mechanisms. The number of instructions executed between external memory references is then multiplied by a clock cycle time. Such shared-memory multiprocessor simulation systems include the Wisconsin Wind Tunnel [33], Stanford Tango [12], MIT PROTEUS [8], and USC Trojan [32].

A small number of environments, ISE among them, have been targeted toward the execution of software applications over simulated message-passing multicomputers or networks of workstations. The additional complexity with such environments resides in the trapping of communication, which is no longer as simple as checking whether a memory reference falls outside of the local memory range. The Delaitre, et al. modeling environment [13] uses a graphical design tool to create message-passing programs which can be converted to PVM code. Code blocks between message-passing calls are described by a fixed delay, which can be measured from separate timing experiments. Once the application is described in the environment, the simulator executes it over a modeled network, making network calls at the specified times. The SPASM environment [36] extends this work by allowing the user to input C or C++ code instead of working through the graphical application design tool. The user's code is integrated with a network model written in CSIM [16] and makes calls to the network model via sending and receiving primitives. Code-block times are obtained from counting instructions using the same assembly-augmentation used by the shared-memory simulation environments. WORMulSim [30] adds an MPI-compliant interface to the network communication calls so that the user's code is portable between the simulator and real testbeds. The pSNOW environment [24] for simulation of Networks of Workstations (NOWs) [3] is much the same except the user's code makes its network calls with Active Messages.

ISE represents a culmination of many of these methods for the realm of rapid virtual prototyping of parallel and distributed systems. In addition, ISE has advantages in convenience, fidelity, extensibility, and modularity above other environments. ISE allows the use of unmodified parallel application software written with MPI to be used in the simulation, whereas many other environments require the code to originate from their design environment or require manual additions to be made to the code for timing purposes. Most of the environments cited above that interface to message-passing applications use very simple network models based on

collections of links and switches or on queuing structures. Though complex models can be created in the environments that use CSIM, they are subject to severe design limitations. Due to the graphical and block-oriented paradigm used by BONEs, network modeling with ISE can be considerably easier than coding a network simulation from scratch in C using CSIM. Furthermore, the network model and simulator are completely separate from the user application in ISE. No linkage must be made to integrate the two sides. Instead, the model is compiled, the application is compiled, and the two entities find each other at runtime. This approach yields unparalleled modularity and expandability, where network models and applications can be swapped in and out with ease. ISE goes even further with the timing of code blocks. Where other parallel systems simulators require augmentation of the assembly code in order to count instructions, processes in ISE measure their code-block times using real-world time on the host machine. This method has the advantage that system calls in the user application are included in the timing. In this way, ISE takes full advantage of the HWIL by measuring more realistic code-block times which include the time the parallel application spends in both user space and kernel space. Using this method, other delays can be included or excluded as desired, such as forced context switches by the operating system upon time-slice expiration or preemption by higher-priority users. Thus, the resulting performance from the virtual system simulation can reflect the performance of the application on shared multi-user systems in addition to dedicated embedded systems, again creating a more realistic result. Furthermore, the use of HWIL saves the designer the difficulty of creating a processor model or an FPGA prototype board, as required by many of the other simulation environments. These factors combine to make ISE a powerful, efficient simulation tool for rapid virtual prototyping using high-fidelity hardware models with real application software.

4.3. ISE Framework

ISE is comprised of several entities, each playing a specific role in the exchange of data and timing information required to assess the performance of a parallel system running parallel code. The structure of ISE depends to some degree on how the parallel and distributed system will be modeled. Given the configuration in which the processors will be real HWIL, rather than simulated processors, the structure of ISE is shown in Fig. 4.1.

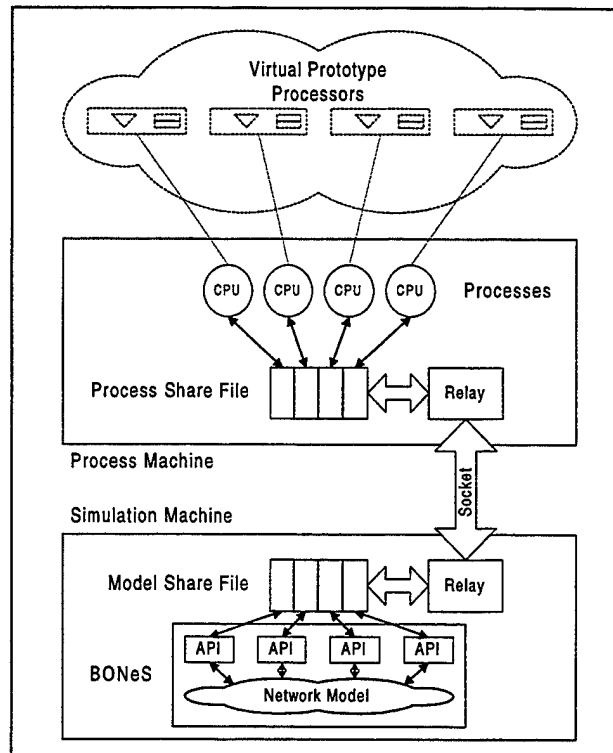


Fig. 4.1 ISE structure with HWIL processors

The *process machine* is responsible for HWIL execution of the program running in parallel on the various processors in the virtual prototype. The ISE runtime process manager creates a *process share file* where each process has a column that it uses to read and write information that is passed to and from the *model share file* during execution. Relay programs on the process machine and *simulation machine* keep the process and model share files consistent. Each process has a corresponding application programming interface (API) agent within the network model that translates between communication calls and simulated network packets.

When a data block is sent from a transmitting process, the data, data size, and destination values are sent to the transmitting API that translates this information for use by the network model. The model dutifully carries the information to the specified destination where it is read by the receiving API, which translates it to the receiving process. For the purposes of relieving the network model of the need to propagate large data blocks, the data is stored in an intermediate location in ISE, and only the address of the data is passed through the network. The performance of the network is unaffected by the actual data values contained in the data block, but it is dependent on the frequency and size of data blocks, and the ISE represents these factors accurately.

4.3.1. High-fidelity Models

In ISE, the network, processor, and software components of the parallel system can be implemented as discrete-event, high-fidelity models of fine granularity that handle the data communications of the parallel application. The models are created using a commercial CAD modeling tool from Alta Group, a division of Cadence, called the Block-Oriented Network Simulator (BONEs) [2, 35] to minimize creation time and maximize model fidelity by using standard library components and simulation management. An additional advantage of using a

structured tool to develop these various models is that the design and measurement techniques of each model remain consistent. Such consistency is important in order to keep the models modular and interoperable, and to allow valid comparisons to be made between models.

The BONEs Designer package consists of editors, libraries, and management functions intended to support the development of event-driven network and processor models. The term *event-driven* refers to simulations that sequentially execute through a dynamic list of events. Simulated time stands still until all events for that time have been completed, at which point events scheduled for the next point in time are started. A diagram of the structure of the BONEs development environment is shown in Fig. 4.2. The Data Structure Editor (DSE) is used for the creation and modification of data structures (e.g. packet formats). The organization and connection of hierarchical process blocks (i.e. the model itself) is accomplished with the Block Diagram Editor (BDE). The Primitive Editor is used to create blocks for use in the BDE from scratch using C++ code. The Simulation Manager controls execution of the model, and any data that was collected during the simulation is viewed with output graphs generated by the Post Processor. The core library contains many elements that are common to network model creation, and consequently, it reduces development and debugging time.

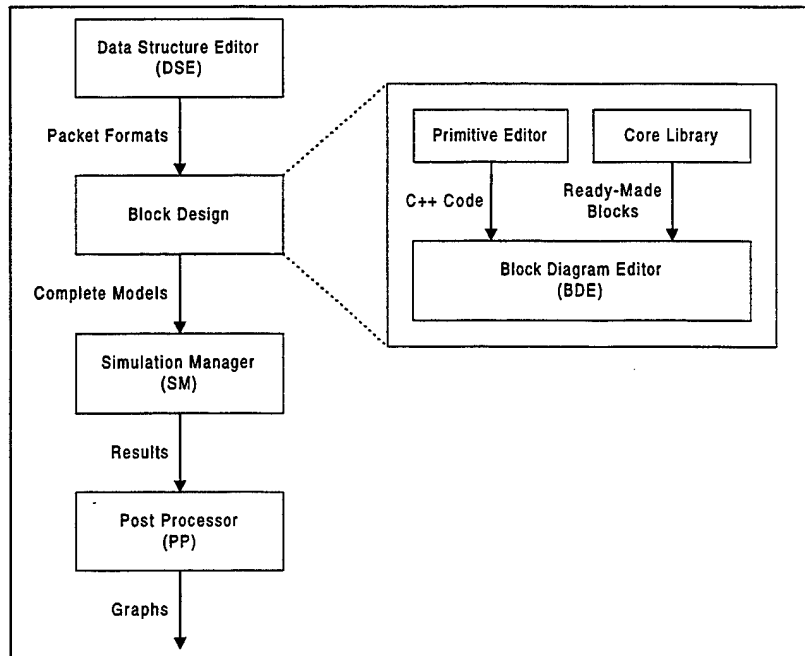


Fig. 4.2 BONEs modeling and simulation components

A typical network model in BONEs is created directly from a description of the message formats, queue structures, mechanisms, and overall protocol operation. In the case of modeling well-known networks and other interconnects, this description may come straight from the specification standard. Often, the message formats are translated directly into the DSE with a few added fields, such as a time stamp or sequence number, that are used solely for data collection purposes and do not affect the behavior of the protocol being modeled. The queue structures and mechanisms of the protocol are modeled using pre-existing blocks from BONEs libraries, user-defined blocks written with C++ code (with the Primitive Editor shown in Fig. 4.2), or a hierarchical combination of both types of blocks. The transmission speed and propagation delays

are accounted for by using delay mechanisms along the network data paths, which causes the simulation clock to increase by an amount determined by the bandwidth and length of the transmission medium. Sample packet types and an internal network node structure are shown below in Fig. 4.3 and 4.4, respectively. The network chosen for this sample is the Scalable Coherent Interface (SCI) which is explained in more detail in a later example.

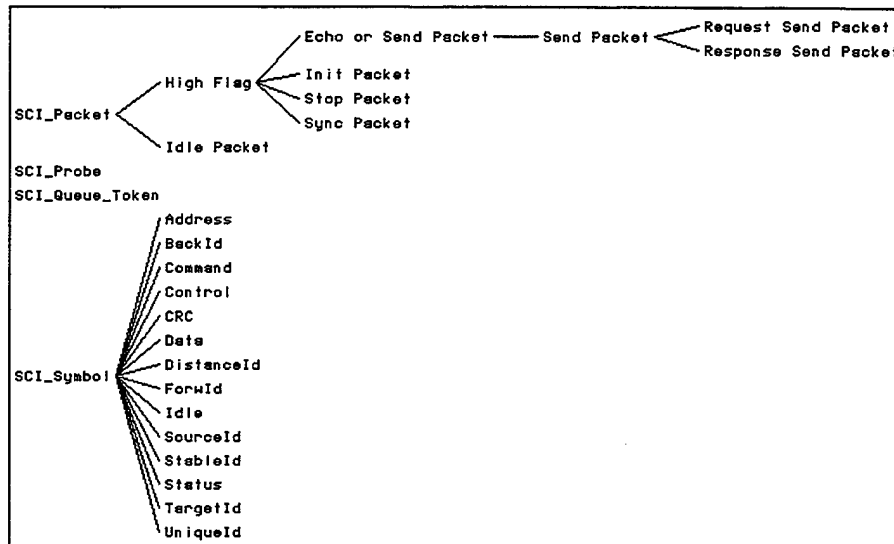


Fig. 4.3 Sample packet types in BONEs

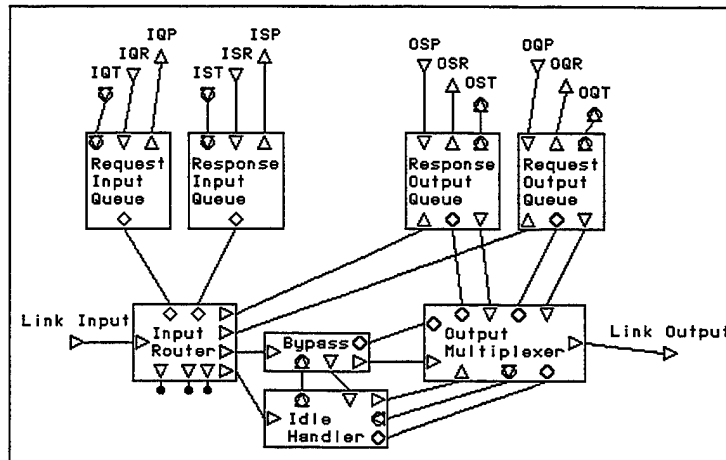


Fig. 4.4 Sample network node structure in BONEs

In a traditional network simulation study, the individual network nodes are constructed from the specification, and then multiple nodes are connected together into a scenario with an appropriate topology. The number of nodes is fixed in a given scenario, so multiple scenarios must be created to study the behavior of different network sizes. To drive the models with statistical distribution, synthetic parametric traffic generators are connected to each node. These conventional traffic generators, in conjunction with statistics probes, are typically used to measure average throughput, latency, and other performance characteristics that may be checked with mathematical analysis to validate the overall protocol operation. An illustration of this

methodology is given in Fig. 4.5 below. By contrast, in ISE the traffic generators are replaced by API agents that generate and receive real network traffic based on the demands of the external processes.

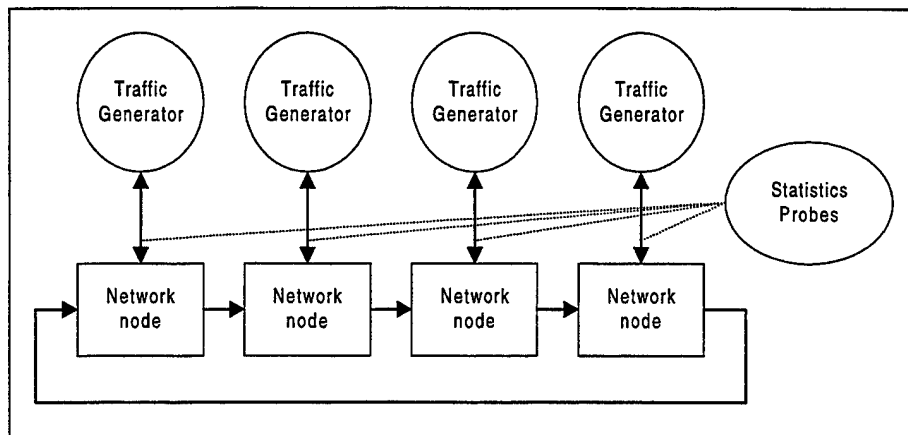


Fig. 4.5 Typical 4-node network simulation scenario with traffic generators

To be compliant with ISE, a network model must support two vital operations. First, since the network is performing real data communication, a field in the data structure must be defined that contains a pointer to the data block location in external memory. Second, the size of the data block must be used properly by the mechanisms that simulate link and propagation delay.

The current implementation of ISE has been used over BONEs network models with the configuration shown previously in Fig. 4.1. However, BONEs communication software models can also be included in the ISE's virtual prototype by simply inserting them between the ISE communication APIs and the nodes of the network model. For example, a model such as that written by Chen, et al. [9] might be used to inject highly accurate TCP or UDP overhead to the simulation.

The addition of a processor model to execute code instead of using the HWIL capabilities would require a modification to the configuration in Fig. 4.1. The ISE structure would change to eliminate the process machine, and the runtime process manager would spawn processes directly to the processor models. This modification would simplify the current ISE runtime manager duties but would require the addition of an equivalent operating system, compiler, linker, and assembler and would significantly increase the complexity of the simulation model. Currently, the ISE uses only HWIL processors to execute parallel code segments rather than processor models that mimic the behavior and performance. Given the increasing standardization and use of RISC processor architectures in the field, from workstations to supercomputers, the processors in an ISE virtual prototype can in many cases be realistically simulated by the execution of the processor in the simulation platform itself. By using a HWIL approach in this fashion, the disadvantages of long simulation times and the difficulties of model creation for each kind of processor can be avoided. To accommodate the inclusion of the widest range of possible processors, ISE allows the designer to automatically scale up or down the execution times measured on these HWIL processors to match the requirements of the processors in the virtual prototype. The limit to this scaling philosophy occurs when the architecture of the processor in the target system (e.g. a digital signal processor) is largely different from the RISC processors used in the host workstations. In this case, there has been previous evidence that processor architectures implemented as software models can accurately relate the performance of actual

machine language code [17, 18]. The time and effort needed to create such models should be weighed in advance against the expected effect on the final simulation results.

4.3.2. High-level Parallel API

In order to use a high-level parallel coordination language, the application programmer must be provided with a standard library of “black-box” function calls to support the message-passing communication. The behavior of these function calls is defined by a parallel language specification, the exact implementation of which depends upon the underlying network structure. In a conventional parallel API implementation, function procedures are ported to the specific underlying communication network with appropriate optimizations to exploit architectural features. In ISE, the same idea has been used to create an API library for the MPI parallel coordination language [28]; the only significant difference here is that the underlying communications network exists solely within the high-fidelity network models described previously. ISE’s API is built into BONEs blocks and is interfaced with the network model by simple connection paths between corresponding ports on the modules. In addition to defining how the MPI function calls are handled, the API interfaces BONEs to the external environment by creating, reading from, and writing to the share files shown in Fig. 4.1.

MPI_Init	Setup the MPI system and strip MPI-specific command-line options.
MPI_Comm_rank	Retrieve the rank number for the process.
MPI_Comm_size	Retrieve the number of nodes in the MPI system.
MPI_Send	Send data to another node; returns when the data is accepted by the network.
MPI_Recv	Blocking receive for data from another node.
MPI_Finalize	End the process’s participation in the MPI system.
MPI_Probe	Blocking probe on reception of data from another node.
MPI_Iprobe	Non-blocking check on reception of data from another node.
MPI_Bcast	Send broadcast data to all nodes or receive broadcast data.
MPI_Barrier	Block until all other nodes have arrived at a barrier.
MPI_Wtime	Retrieve the current time.
MPI_Reduce	Participate in a vector reduction operation, such as element-by-element minimum, maximum, sum, product, etc.

Table 4.1 MPI function calls currently supported by ISE

The initial implementation of the ISE interface to MPI is compliant with a subset of the MPI standard. ISE currently supports the key function calls that are necessary to perform the most common communications over MPI. A list of the function calls supported along with their descriptions is provided in Table 4.1. These functions serve to make ISE a versatile tool by allowing almost any unmodified message-passing parallel program to be interfaced to the virtual prototype.

4.3.3. Putting It All Together

ISE provides several mechanisms by which the hardware models and software applications discussed above are integrated into a single powerful simulation environment. The software-in-the-loop (SWIL) methodology in which the real software is used during simulation provides considerable insight into the intricacies of system development. The hardware-in-the-loop (HWIL) mechanisms used in ISE add realism to the processor portion of the simulations and decrease simulation time. Finally, a profiling interface to ISE simulations gives designers easy access to a graphical performance-monitoring tool.

The concept behind SWIL is the ability to use real application software in the simulation. Using such a mechanism results in considerable portability advantages, which was in fact one of the original goals of the MPI coordination language. To use an MPI application over ISE, the user takes the same code that compiles and runs on real parallel and distributed systems with any commercial or freeware MPI implementation. The only difference is that the code is linked with ISE's version of the MPI library instead of the library for the real parallel system. Movement from simulation to real implementation, or vice versa, can be made without apprehension about modifying the functionality of the software during the move.

The HWIL concept is just as integral to ISE as SWIL. Using the processor in the host workstation to execute the user's application and timing the code blocks with real-world time, ISE is able to include many of the complexities of modern workstations into the simulation as desired. For instance, by timing code blocks using real-world time, ISE results will include the performance effects of system calls and standard library calls from the user's application. Furthermore, by optionally using a host workstation that is executing one or more competing jobs, the measured code-block times will include the time the SWIL application spends waiting while sharing the processor with the other applications of lesser, equal, or greater priority. Thus, ISE results can reflect the performance of that parallel application in a multi-user environment with deterministic or random activities in competition for system resources.

In order to analyze the performance of the virtual prototypes created with ISE, it is often helpful to view all the events in the parallel system using a graphical profiling tool. All simulation runs with ISE automatically capture the necessary timing information to create a log for use with the freely available Upshot profiling program [19]. The sample profile in Fig. 4.6 shows the computation and communication times for a manager/worker parallel decomposition of a matrix multiplication. Each horizontal row of the output corresponds to one processor in the parallel system, and time progresses from left to right. The thin sections of each row represent the computational code blocks executed by the HWIL of ISE. The thick bars are communication calls handled by the network model, with arrows indicating sender-receiver pairs and the legend indicating which particular MPI call (e.g. *Send*, *Recv*, *Bcast*) is occurring. When event durations are too small to view adequately, such as the *Send* blocks in Fig. 4.6, Upshot is able to zoom in so that the designer can examine portions of the execution more closely. Using this profiling interface, the numerous design choices possible with ISE become more easily quantified and evaluated.

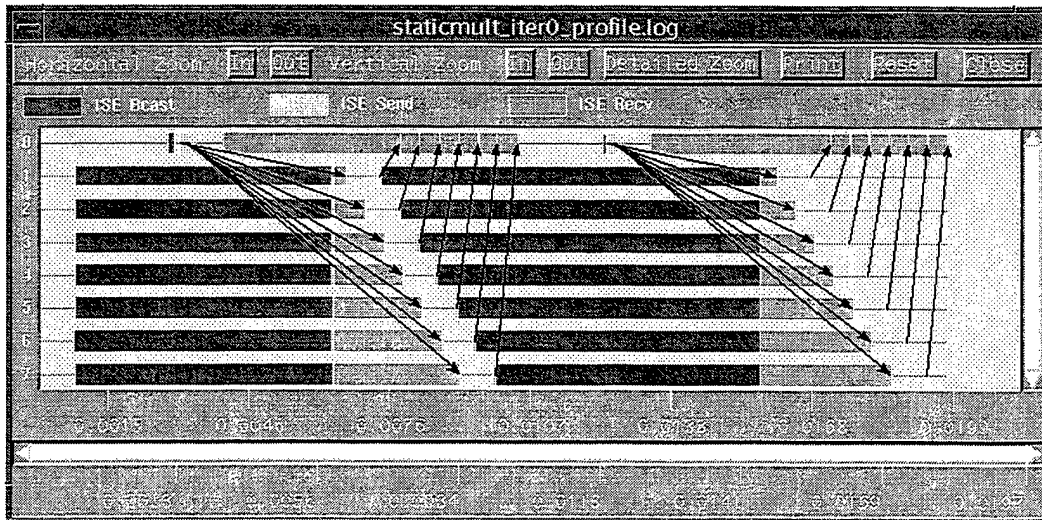


Fig. 4.6 A sample of the parallel profiling output in ISE via Upshot

With these mechanisms available to the designer, rapid virtual prototyping of complex parallel and distributed systems becomes possible. Due to the SWIL and HWIL capabilities, the application software created during this process is immediately available for real parallel and distributed systems without the need for any modifications. The interface to BONEs provides the designer with the ability to create high-fidelity network models, the interface to MPI provides access to the most popular message-passing coordination language in use today, and ISE itself provides the glue to make integrated simulation more flexible and realistic than was previously possible.

4.4. Containment of Simulation Explosion

The use of high-fidelity models in the simulative process may become unwieldy due to the amount of time and computing power needed to complete such simulations. Particularly in the case of simulating large distributed systems comprised of detailed models of every component, the simulation time may become such a dominant factor that the user may complain that rapid virtual prototyping is not very rapid. Methods invoked by ISE in order to reduce the necessary computing power and reduce simulation time include distributed simulation techniques and the novel method of *fast forwarding*. It is also worth mentioning that the use of HWIL processors for execution of the user's application, in addition to providing more realistic timing results, saves the simulation the burden of a processor model, again helping to contain simulation explosion.

There is a whole field dedicated to optimization of simulating large, complex models in a parallel or distributed manner using warping, checkpoints, rollback-recovery, and the like. Many such methods are described in Fujimoto [20] and Righter, et al. [34]. These methods attack the model explosion problem at a fine level of granularity so that multiple CPUs work together to speed up a single simulation based on one set of parameters. However, if the parameters of the simulation are varied, as is often the case, then unique permutations are created that require independent simulations. Therefore, as proposed by Biles, et al. [5] and others, farming these similar yet independent simulations to many loosely coupled workstations in a coarse-grained parallel manner can yield the same speedup as fine-grained methods but with less coordination overhead and complexity.

The key to effective distribution of simulations is the use of varied parameters to create enough simulations to keep the workstations busy. Fig. 4.7 illustrates how a large simulation task consisting of several parameter sets may be partitioned at the fine-grained and coarse-grained levels. In this case, suppose that three instances of a network model with bandwidths of 10 Mb/s, 100 Mb/s, and 1 Gb/s are to be studied. Further suppose that each instance requires 1 hour to simulate and that there are 3 workstations available to the simulation environment. With the fine-grained approach to simulation parallelism, the first simulation could be executed in parallel on all 3 workstations in 20 minutes. The second simulation would then be spawned, completing in 20 minutes, followed by the third simulation. The result is that all 3 simulations are completed in 1 hour. By contrast, the coarse-grained approach employed by ISE executes the 3 simulations concurrently on the 3 workstations (i.e. one simulation per machine), resulting in all 3 simulations again completing in 1 hour. Although this comparison has been simplified and disregards the effects of overhead and synchronization, these effects would likely lower the efficiency of the fine-grained approach more than that of the coarse-grained.

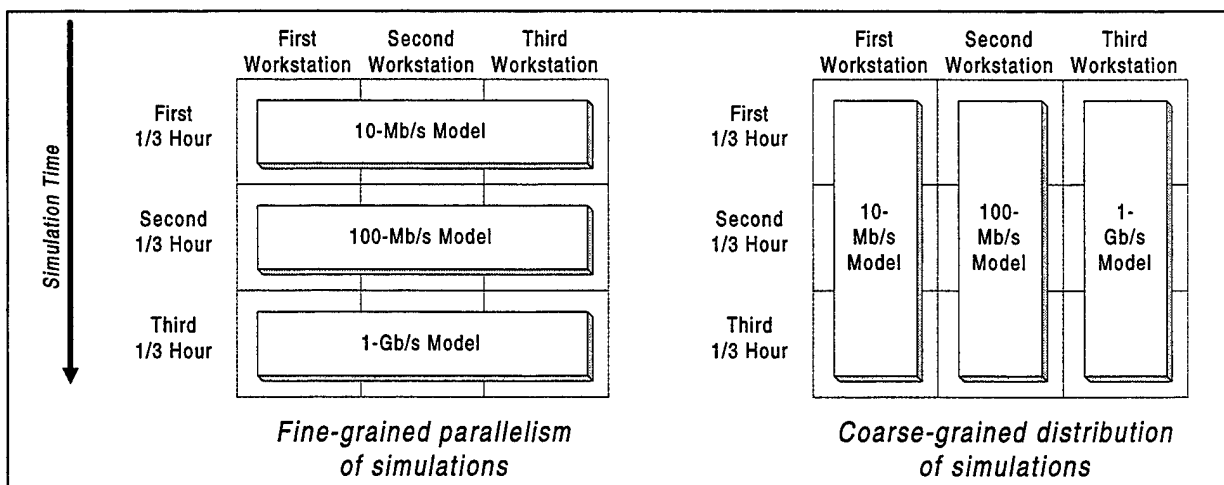


Fig. 4.7 Fine-grained parallelism vs. coarse-grained distribution of simulations

ISE is designed to fully exploit BONEs's ability to spawn multiple concurrent iterations of a single simulation scenario. Each iteration corresponds to a single permutation of a parameter set that occurs when the effect of changing one or more variables is desired. The use of *range variables* in BONEs causes the simulation manager to spawn the iterations as batch jobs to the available networked workstations selected at simulation time. ISE supports multiple iterations by carefully matching the multiple process iterations on the process machine (see Fig. 4.1) with the corresponding iterations on the simulation machines. Further support is provided by the Upshot profiling interface. ISE creates profiling output for each node of each iteration and allows the user to not only view the results of nodes for a particular iteration, but also multiple sets of results for the same node from different iterations. This capability adds to the usefulness of ISE for comparative analysis of the virtual prototypes.

The second method employed by ISE to control simulation explosion is fast forwarding. As is often the case, the specification for the network that is to be used as the network model in ISE may require idle packets or symbols to be continuously sent during periods of network inactivity. A high-fidelity model of such a network will faithfully create, communicate, and sink these idle symbols whenever the transmitting device has no packets to send. Such will be the case during all periods of sequential computation. To simulate the communication of idle symbols during

these times may very well be unimportant to the fidelity of the simulation and would only add to simulation explosion. At the user's request, ISE can fast forward through periods of network inactivity, thus saving considerable simulation time. Without fast forwarding, several seconds of network idle time would require several hours or days to simulate; however, with fast forwarding it will take only seconds. Thus, during these periods, simulation time is on the order of the simulated execution time of the virtual prototype.

The two methods of coarse-grained distributed simulation and fast forwarding add significant convenience for the designer, which is often a deciding factor in whether a tool is deemed useful or not. In fact, both these methods were extremely helpful in speeding up the process of collecting results for the comparative case study shown in the next section, which involves the simulation of 32 design permutations on the virtual prototype for a parallel and distributed system.

4.5. Validation and Case Study

As with any new simulation environment, a demonstration of the validity and usefulness of ISE is needed. In this section, a validation of ISE is introduced using SCI experiments. Two validation experiments are described, a network latency experiment for validating communication delays and a parallel computing experiment for validating both communication and computation. The goal for both validation tests is to compare the performance results from ISE to those from a real testbed. Also included in this section is the description of a comparative case study in which two parallel signal-processing algorithms are simulated over distributed architectures with varying network and processor speeds. The results of these experiments are presented in Section 4.6.

4.5.1. Description of Validation Experiments

Since the primary goal of ISE is to serve as a tool for predicting parallel and distributed system performance, it is desirable to have the results of a simulation closely match that of the real system. In order to validate the models used in ISE, the results of programs run on the real system can be compared to the ISE-simulated system and delay parameters can be set accordingly. The modeled system presented here is based on an SCI network connecting RISC workstations.

IEEE Standard 1596-1992 Scalable Coherent Interface [23] is based on a structure of scalable register-insertion rings with shared-memory split transactions. The SCI standard specifies a data rate of 8 Gb/s per link with current implementations from Dolphin Interconnect Solutions, the leading SCI vendor, supporting 1.6 Gb/s. A node diagram in Fig. 4.8 below illustrates the internal node structure of SCI. A request or response packet entering the node on the input link is relayed by the address decoder to the request or response input FIFO, respectively, if the packet's destination address matches the address of the node. If it does not match, the packet is relayed through the bypass FIFO to return to the network via the output link. Similarly, requests and responses made by the host are queued and then multiplexed onto the output link along with the output of the bypass FIFO. The SCI specification has been implemented in the design of the SCI cards from Dolphin [14], and in the SCI node model we have constructed in BONEs, verified, and integrated into ISE.

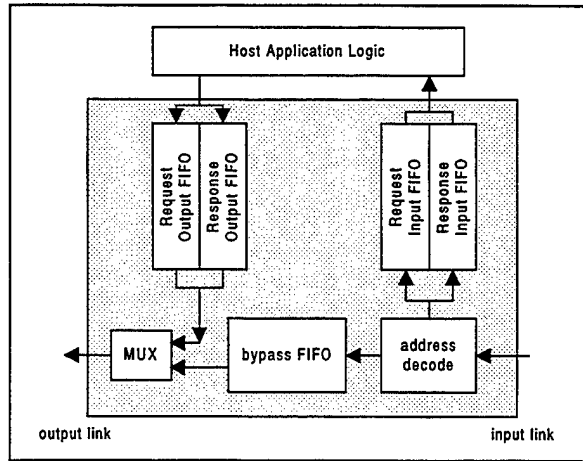


Fig. 4.8 SCI node model

The two validation experiments are designed to measure the two major types of delays that are incurred within a parallel and distributed system: communication and computation. A high-fidelity model of a network protocol and communications stack is used to simulate and determine the communication time, and the execution of code blocks over a real or simulated processor constitutes the computation time. To validate the communication and computation time reported by ISE, these experiments are run over the SCI cluster, a parallel and distributed computing testbed. This testbed consists of 200-MHz UltraSPARC-2 workstations running Solaris 2.5.1 and interconnected by 1.6 Gb/s SCI. The network interface cards are SCI/Sbus-2B devices from Dolphin, and the MPI implementation used by the programs is MPISCI [31]. Fig. 4.9 illustrates how SCI in the real and ISE-simulated system configurations compare.

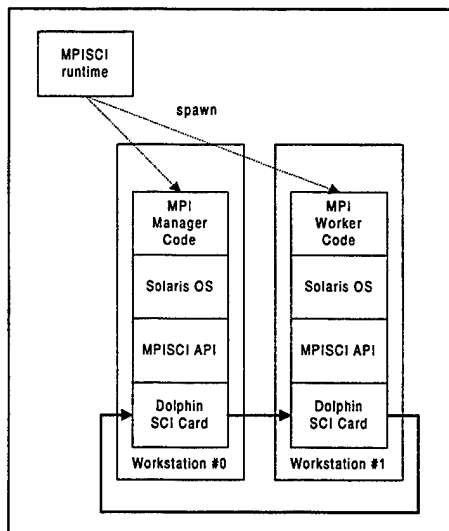


Fig. 4.9a SCI cluster testbed composed of UltraSPARC workstations connected by SCI adapter cards (2-node case)

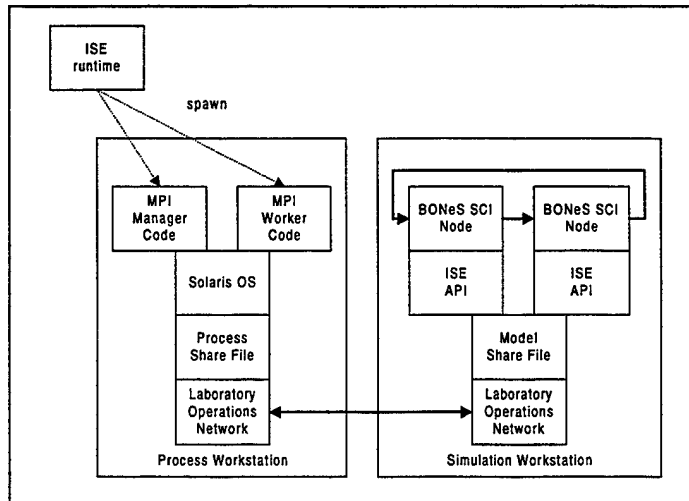


Fig. 4.9b Structure of ISE in the simulation of the virtual prototype of the SCI cluster (2-node case)

The first validation experiment, *Acknowledged Send*, uses MPI calls to send simple, acknowledged packet transfers between computers across the network. This test is used to measure the application-to-application communication latency through the communication software stack and the network itself. The second validation experiment is a *Matrix Multiply* parallel program written in MPI that produces both communication and computation delays. The manager node in this program distributes parts of two matrices to be multiplied by the worker nodes. The intermediate results from each node are collected by the manager after the calculations.

4.5.2. Description of Case Study

There are many situations in which it is desirable to first validate a parallel system model by ramping down the size or speed of the components to match a testbed in the laboratory. Once the accuracy of the models has been validated and optimized to meet the needs of the design, the virtual prototype can then be ramped back up to the sizes and speeds required of the new system. For example, we may validate the virtual prototype for a candidate parallel system with SCI operating at 1.6 Gb/s using existing hardware. Once the validation is complete, the virtual prototype can then be modified to operate with an 8.0 Gb/s SCI data rate with a high degree of confidence in the results. In this fashion, we can use small testbed prototypes to garner more accuracy for the models, and then move forward with network data rates, processor speeds, degrees of parallelism, and other enhancements that are not easily attained in a testbed prototype due to limitations in technology, cost, component availability, etc. In effect, we leverage the best features of the experimental world with the best of the simulative world.

Of course, whether we validate with a small testbed prototype or not, the eventual goal is to design, develop, and evaluate a new parallel and distributed computing system. In so doing, we may wish to hold the architecture fixed, vary the algorithms, and study the effects. In other cases, we may wish to hold the algorithms fixed, vary the architectural features, and study these effects. Or, in still other cases, our intent may be to study a wide array of permutations in both algorithm and architecture. To illustrate the latter, a case study is presented in which two parallel algorithms for sonar signal processing are compared while changing features of the distributed computer architecture model. This case study is intended to demonstrate that ISE is useful as an

interactive and iterative design environment for architectures which may not have as yet a real-world implementation.

The case study involves the development of a virtual prototype for a special-purpose parallel and distributed system targeted for high-speed and low-power execution of sonar beamforming algorithms. The architecture of interest is a multicomputer comprised of eight independent processors interconnected by a bidirectional, linear array network and communicating via message passing. The link speed between the nodes is varied between 2.5 Mb/s and 10 Mb/s, and the processor speed on each network node is varied between 25% and 100% of the performance of an UltraSPARC microprocessor. Two parallel algorithms coded and implemented in MPI, called *Beamformer1* and *Beamformer2*, are compared to evaluate their relative performance and sensitivity to network and processor speed. The two programs differ in that *Beamformer1* focuses on parallel beamforming whereas *Beamformer2* goes one step further by adding overlap of computation and communication wherever possible. As such, the processor and network speeds will affect the two programs in different ways, making ISE the ideal environment to prototype the system and evaluate the design alternatives.

4.6. Results

The results of the networking and parallel processing experiments for validation purposes are provided in the following subsection. Afterwards, the case study on performance issues with embedded parallel architectures and algorithms for sonar signal processing using variation in algorithms and architectural features is presented.

4.6.1. Results of Validation Experiments

The results of the first set of validation tests, *Acknowledged Send*, over a 2-node SCI network connecting UltraSPARC workstations are shown in Fig. 4.10a. The data points of the two curves represent values collected from the same MPI program run over the actual testbed and the ISE-simulated virtual prototype. The values of parameters in the network and communication stack models have been calibrated, and this tuning coupled with the inherent accuracy of the high-fidelity model developed for SCI allows us to achieve performance curves that are nearly identical.

Since the model was adjusted for two nodes, it can reasonably be expected that the results will be somewhat dissimilar for different network sizes. To preserve the validity of the simulations, all model parameters adjusted in the 2-node simulations were held constant for the 4-node simulations. The 4-node test results shown in Fig. 4.10b begin to deviate as the packet size increases. At 65536-integer packets, there is a maximum of 20% deviation between real and simulated *Acknowledged Send* times.

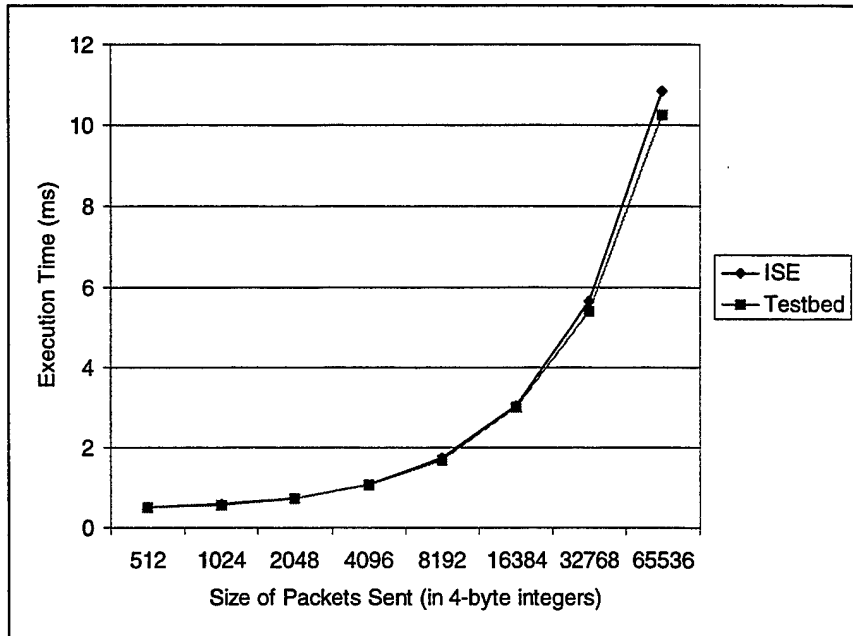


Fig 4.10a Testbed vs. ISE: *Acknowledged Send* experiments for 2-node SCI clusters

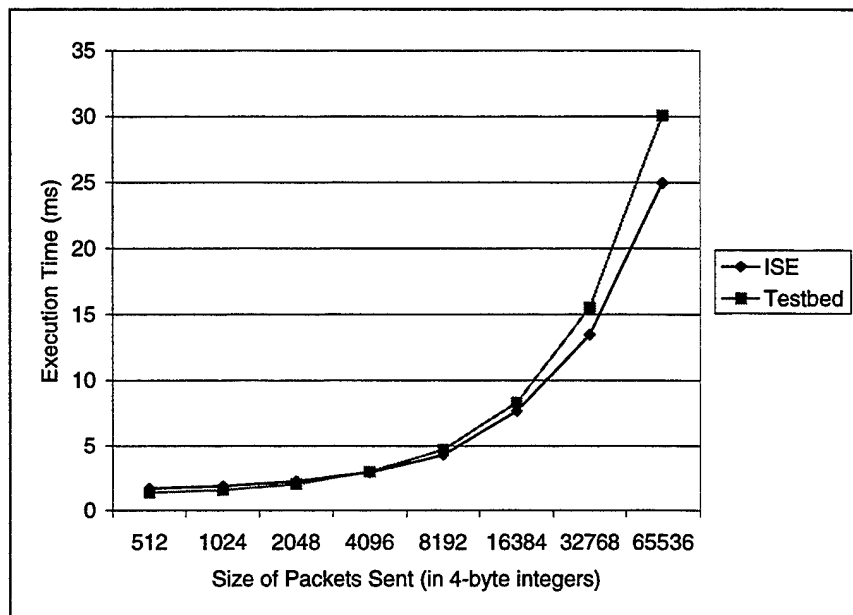


Fig. 4.10b Testbed vs. ISE: *Acknowledged Send* experiments for 4-node SCI clusters

The results of the execution times from the second set of validation experiments are shown in Fig. 4.11. Fig. 4.11a presents the results of the 2-node parallel *Matrix Multiply* over both the testbed and the ISE-simulated virtual prototype, while Fig. 4.11b shows the results for a 4-node run. Simulation parameters for these experiments were unchanged from the validated parameters set for the *Acknowledge Send* tests.

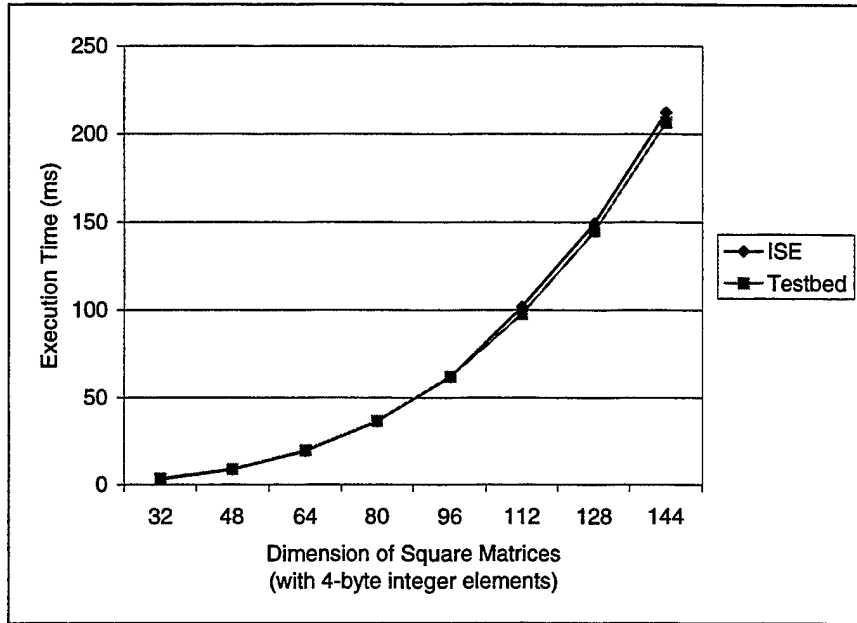


Fig. 4.11a Testbed vs. ISE: *Matrix Multiply* completion time for 2-node SCI clusters

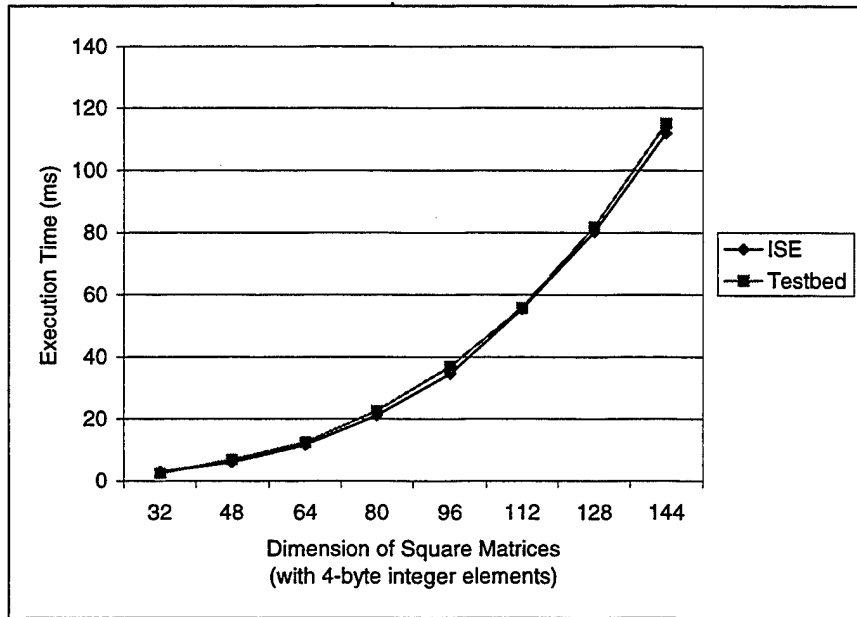


Fig. 4.11b Testbed vs. ISE: *Matrix Multiply* completion time for 4-node SCI clusters

As these results illustrate, ISE provides performance estimation that closely matches the performance of the SCI-connected UltraSPARC testbed, thus giving the user a high degree of confidence in the validity of the virtual prototype. However, when no validation testbed is available, ISE is still strong at allowing comparative tradeoffs to be made with respect to algorithm and architecture design parameters.

4.6.2. Results of Case Study

The results from the comparative case study of the parallel beamformer algorithms over simulated architectures of varying network and processor speeds are presented next. The graph in Fig. 4.12a shows the behavior of the *Beamformer1* algorithm running over the virtual sonar system prototype. To vary the speed of the eight processors in the virtual prototype of the sonar computer system, variations in processor speed range from 25% to 100% of the performance of an UltraSPARC processor running at 170 MHz. Similarly, variations in network speed range from 2.5 to 10 Mb/s.

The results of *Beamformer2*, shown in Fig. 4.12b, provide the comparative information that makes this case sensitivity study most useful. At each point on the two contour graphs, a comparison of the overall execution time for each algorithm can be reliably made because the network model assumptions and processor behavior have been kept constant between algorithms. The only differences between the two figures are the algorithms themselves. The advantages of the overlapping communication in *Beamformer2* are reflected in the fact that the performance is relatively insensitive to network speed in most cases, whereas in *Beamformer1* the slower network speeds have a marked effect on performance. With both algorithms the processor speed was found to contribute significantly to overall performance in a non-linear fashion.

The sort of sensitivity and comparative analysis shown here is a major advantage when using ISE for rapid virtual prototyping of complex systems. Tradeoff analyses can be made to better understand the contributions of each potential bottleneck in the architecture or software. ISE supports an interactive and iterative process in which the designer can quickly make changes to address key performance issues. Such a methodology leads to more efficient and effective design solutions. In addition, virtual prototyping with ISE allows the designer to keep multiple candidates (e.g. the two versions of the beamformer above or two candidate networks) on hand for further evaluation at a moment's notice.

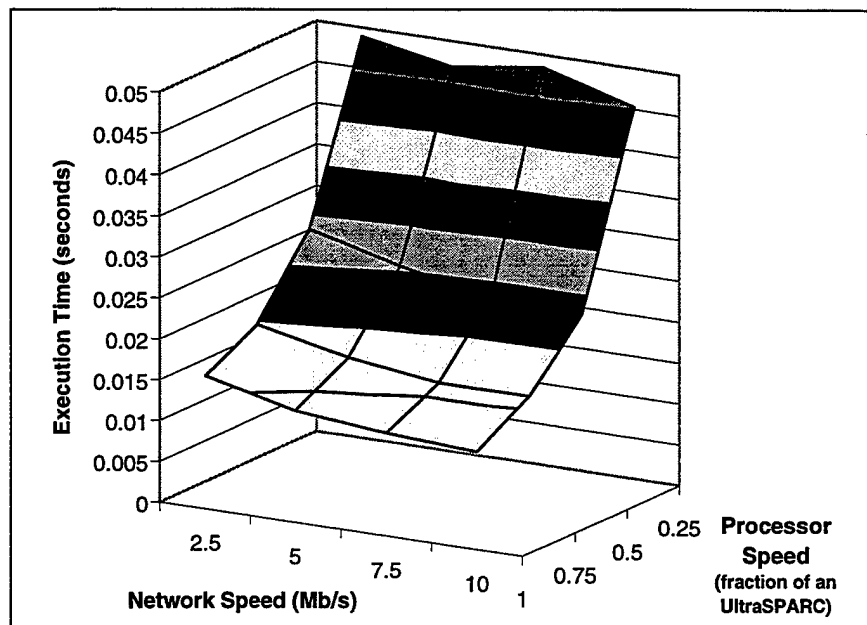


Fig. 4.12a Simulated execution time for *Beamformer1* with variance in network and processor speeds

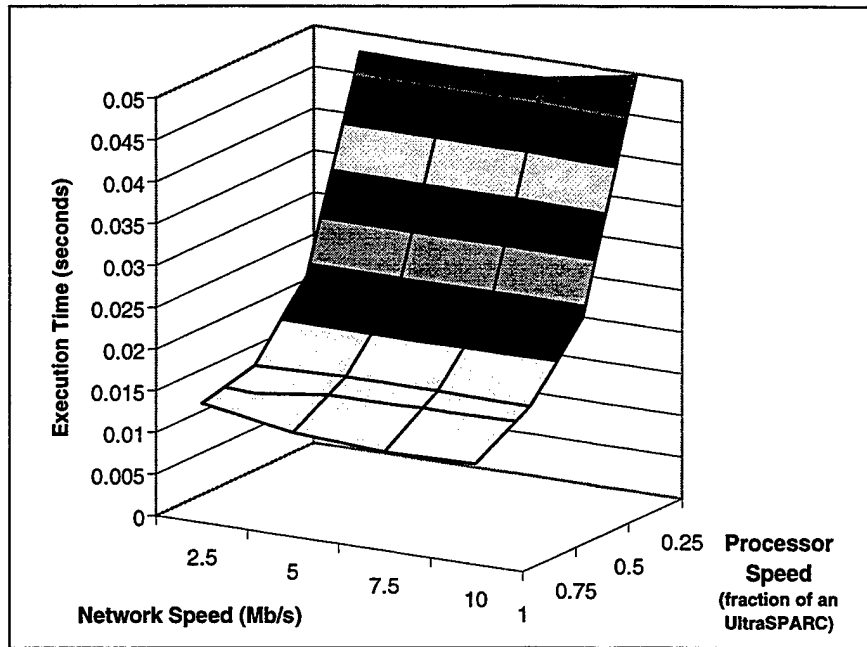


Fig. 4.12b Simulated execution time for *Beamformer2* with variance in network and processor speeds

4.7. Conclusions

This paper has given an overview of the structure, features, and intended uses of the Integrated Simulation Environment. This environment allows the user to execute real parallel programs running on virtual prototypes of parallel and distributed computing architectures and systems, by leveraging HWIL processors and high-fidelity models, so as to evaluate overall system performance. While high fidelity brings with it an increase in computational complexity for simulation, methods have been presented by which the problem of simulation explosion can be contained. Validation results for both networking and parallel processing behavior have illustrated how a virtual prototype in ISE can achieve virtually the same performance as the actual target system. The usefulness and flexibility of ISE for developing and comparing notional network architectures, with real parallel code instead of simple traffic generators, has also been demonstrated with a case study taken from a real-world application in sonar signal processing.

ISE provides the designers of parallel and distributed architectures and algorithms a method to combine high-fidelity network models and real parallel code running over real processors for execution-driven simulation. Moreover, the models may be tuned with small-scale prototypes from the testbed and then ramped up to larger data rates, processor speeds, degrees of parallelism, etc. so that results from ISE can nearly emulate the behavior of the target system. ISE is equally useful in comparing notional architecture models with consistent assumptions and performance trends that mimic the expected behavior of the real systems. Models that have been created and validated in BONEs by hardware architects may be shared with software designers who can better analyze how their programs will perform on the architecture. Likewise, software can be provided to the hardware designers, who can tailor the architectures for optimum performance.

Several future extensions to ISE are under development in order to make it an even more powerful approach for the design and analysis of parallel and distributed systems. For example, future work will strive to extend the capability of simulating fully heterogeneous systems

composed of different kinds of processors (e.g. RISC, VLIW, DSP), connected by a variety of networks in complex topologies, with an assortment of interfaces, memory hierarchies, etc. In addition, work will be pursued toward the goal of making the interface to processor models as seamless as is the current interface to the HWIL processors. With these enhancements, the usefulness of ISE will be expanded to cover a wider range of system designs, from scalable multicomputer systems with multiple network connections to tightly coupled, shared-memory multiprocessors.

4.8. References

- [1] J.K. Adams, D.E. Thomas, "Design Automation for Mixed Hardware-Software Systems," *Electronic Design*, vol 45, n 5, Mar. 3, 1997, pp. 64-72.
- [2] Alta Group, *BONeS DESIGNER User's Guide*, Foster City, CA: Alta Group, 1994.
- [3] T.E. Anderson, D.E. Culler, D.A. Patterson, "A Case for NOW (Networks of Workstations)," *IEEE Micro*, vol 15, n 1, Feb. 1995, pp. 54-64.
- [4] D. Becker, R.K. Singh, S.G. Tell, "An Engineering Environment for Hardware/Software Co-Simulation," *29th ACM/IEEE Design Automation Conference*, 1992, pp. 129-134.
- [5] W.E. Biles, C.M. Daniels, T.J. O'Donnell, "Statistical Considerations in Simulation on a Network of Microcomputers," *Proceedings of the 1985 Winter Simulation Conference*, 1985, pp. 388-393.
- [6] W.D. Bishop, W.M. Loucks, "A Heterogeneous Environment for Hardware/Software Cosimulation," *Proceedings of the IEEE Annual Simulation Symposium*, 1997, pp. 14-22.
- [7] M. Borgatti, R. Rambaldi, G. Gori, R. Guerrieri, "A Smoothly Upgradable Approach to Virtual Emulation of HW/SW Systems," *Proceedings of the International Workshop on Rapid System Prototyping*, 1996, pp. 83-88.
- [8] E.A. Brewer, C.N. Dellarocas, "PROTEUS: A High-Performance Parallel-Architecture Simulator," *Performance Evaluation Review*, vol 20, n 1, Jun. 1992, pp. 247-248.
- [9] J. Chen, M. Lee, T. Saadawi, "A Block-based Simulation Tool for the Enhancement of TCP/UDP Protocols," *Simulation*, vol 64, n 1, Jan. 1995, pp. 61-63.
- [10] K. Cosic, I. Kopriva, I. Miller, "Workstation for Integrated System Design and Development," *Simulation*, vol 58, n 3, Mar. 1992, pp. 152-162.
- [11] F. Dahlgren, "A Program-Driven Simulation Model of an MIMD Multiprocessor," *Proceedings of the 24th Annual Simulation Symposium*, 1991, pp. 40-49.
- [12] H. Davis, S.R. Goldschmidt, J. Hennessy, "Multiprocessor Simulation and Tracing Using Tango," *Proceedings of the International Conference on Parallel Processing*, vol 2, 1991, pp. II99-II107.
- [13] T. Delaitre, G.R. Ribeiro-Justo, F. Spies, S.C. Winter, "A Graphical Toolset for Simulation Modeling of Parallel Systems," *Parallel Computing*, vol 22, no 3, Feb. 28, 1997, pp. 1823-1836.
- [14] Dolphin Interconnect Solutions, AS, *Dolphin SBUS-2 Cluster Adapter Card Overview*, Oslo, Norway: Dolphin Interconnect Solutions, Sep. 1996.
- [15] D. Dunlop, K. McKinley, "OMI—A Standard Model Interface for IP Delivery," *Proceedings of the International Verilog HDL Conference*, Mar. 1997, pp. 83-90.
- [16] G. Edwards, R. Sankar, "Modeling and Simulation of Networks Using CSIM," *Simulation*, vol 58, n 2, Feb. 1992, pp. 131-136.
- [17] A. George, "Simulating Microprocessor-Based Parallel Computers Using Processor Libraries," *Simulation*, vol 60, n 2, Feb. 1993, pp. 129-134.
- [18] A. George, S. Cook, "Distributed Simulation of Parallel DSP Architectures on Workstation Clusters," *Simulation*, vol 67, n 2, Aug. 1996, pp. 94-105.
- [19] W. Gropp, E. Lusk, N. Doss, A. Skjellum, "A High-performance, Portable Implementation of the MPI Message Passing Interface Standard," *Parallel Computing*, vol 22, n 6, Sep. 1996, pp. 789-828.
- [20] R.M. Fujimoto, "Parallel Discrete Event Simulation," *Communications of the ACM*, vol 33, n 10, Oct. 1990, pp. 30-53.

- [21] T.V. Huynh, H.G. Tirud, "An Approach to End-to-End System Simulation," *1997 IEEE Aerospace Conference Proceedings*, 1997, pp. 447-461.
- [22] IEEE, 1499-1998 IEEE Draft Standard D4.0: Interface for Hardware Description Models of Electronic Components, Piscataway, NJ: IEEE Service Center, 1998.
- [23] IEEE, 1596-1992 IEEE Standard for Scalable Coherent Interface (SCI), Piscataway, NJ: IEEE Service Center, 1993.
- [24] M. Kasbekar, S. Nagar, A. Sivasubramaniam, "pSNOW: A Tool to Evaluate Architectural Issues for NOW Environments," *Proceedings of the International Conference on Supercomputing*, 1997, pp. 100-107.
- [25] K. Kim, Y. Kim, Y. Shin, K. Choi, "An Integrated Hardware-Software Cosimulation Environment with Automated Interface Generation," *Proceedings of the International Workshop on Rapid System Prototyping*, 1996, pp. 66-71.
- [26] T. Le, F.-M. Renner, M. Glesner, "Hardware in-the-loop Simulation—a Rapid Prototyping Approach for Designing Mechatronics Systems," *Proceedings of the International Workshop on Rapid System Prototyping*, 1997, pp. 116-121.
- [27] C. Liem, F. Naçabal, C. Valderrama, P. Paulin, A. Jerraya, "System-on-a-Chip Cosimulation and Compilation," *IEEE Design & Test of Computers*, vol 14, n 2, Apr.-Jun. 1997, pp. 16-25.
- [28] Message-Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, Knoxville, TN: U. of Tennessee, May 1994.
- [29] C.M. Pancake, M.L. Simmons, J.C. Yan, "Performance Evaluation Tools for Parallel and Distributed Systems," *IEEE Computer*, vol 28, n 11, Nov. 1995, pp. 16-19.
- [30] D.K. Panda, D. Basak, D. Dai, R. Kesavan, R. Sivaram, M. Banikazemi, V. Moorthy, "Simulation of Modern Parallel Systems," *Proceedings of the 1997 Winter Simulation Conference*, 1997, pp. 1013-1020.
- [31] Parallab, "Programmer's Guide to MPI for Dolphin's SBus-to-SCI Adapters," Bergen, Norway: Parallab, U. of Bergen, 1995.
- [32] D. Park, R.H. Saavedra, "Trojan: A High-Performance Simulator for Shared Memory Architectures," *Proceedings of the 29th Annual Simulation Symposium*, 1996, pp. 44-53.
- [33] S.K. Reinhardt, M.D. Hill, J.R. Larus, A.R. Lebeck, J.C. Lewis, D.A. Wood, "The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers," *Performance Evaluation Review*, vol 21, n 1, Jun. 1993, pp. 48-60.
- [34] R. Richter, J.C. Walrand, "Distributed Simulation of Discrete Event Systems," *Proceedings of the IEEE*, vol 77, n 1, Jan. 1989, pp. 99-113.
- [35] K.S. Shanmugan, V. Frost, W. LaRue, "A Block-Oriented Network Simulator (BONeS)," *Simulation*, vol 58, n 2, Feb. 1992, pp. 83-94.
- [36] A. Sivasubramaniam, "Execution-Driven Simulators for Parallel Systems Design," *Proceedings of the 1997 Winter Simulation Conference*, 1997, pp. 1021-1028.
- [37] J.-P. Soininen, T. Huttunen, K. Tiensyrjä, H. Heusala, "Cosimulation of Real-Time Control Systems," *Proceedings, European Design Automation Conference with EURO-VHDL*, 1995, pp. 170-175.
- [38] C.A. Valderrama, F. Naçabal, P. Paulin, A.A. Jerraya, "Automatic Generation of Interfaces for Distributed C-VHDL Cosimulation of Embedded Systems: An Industrial Experience," *Proceedings of the International Workshop on Rapid System Prototyping*, 1996, pp. 72-77.
- [39] J. Wilson, "A New Methodology for Co-Design: Virtual System Integration," *Electronic Engineering*, vol 67, n 825, Sep. 1995, pp. S55-S59.

5. PRELIMINARY DESIGN AND MEASUREMENTS WITH THE DISTRIBUTED PARALLEL SONAR ARRAY PROTOTYPE

This chapter presents the prototype distributed and parallel array developed for this project, configured as eight or fewer nodes tied together via a time-division multiplexed bus. Also included is a description of the software communications implementation and experimental results of the split-aperture parallel beamformer executing on the prototype.

5.1. The TMS320C542 DSKplus Development Board

The DSKplus is a development board readily available from TI which integrates a low-power TMS320C542, a Host Port Interface (HPI), Analog Inputs and Outputs operating through the TLC320AC01 CODEC, socketed crystal oscillator, and I/O expansion headers for external designs. In addition it is supported by a suite of software tools that are used to debug and run software on the DSKplus. Fig. 5.1 below shows a layout of the DSK board.

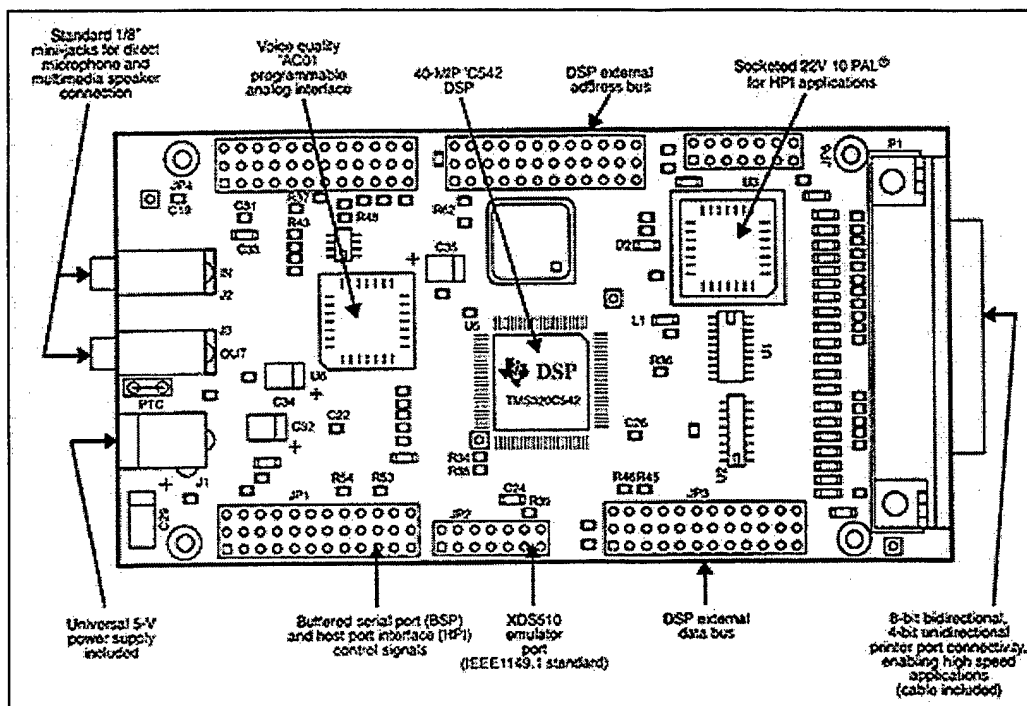


Fig. 5.1 The DSKplus Development Board

The TMS320C542 is a low-power DSP architecture with 10KW (i.e. 20KB) of on-board memory (which may be mapped to program or data space), 2KW of onboard ROM, one on-chip timer, and a phase-locked-loop (PLL) clock generator (which may be used to multiply the off-chip oscillator), an asynchronous buffered serial port (BSP). The processor also incorporates a time-division multiplexed (TDM) serial bus (which may be configured for synchronous serial operation or for coordinating between multiple processors).

The *DPSA Daughter Card* designed and fabricated at the University of Florida for this project is an expansion board that sits atop the DSKplus card and offers four additional functions. First, a set of jumpers were added to control the TMS320's onchip PLL so that the clock could easily be divided or multiplied. Second, two 128KB SRAM blocks were added to expand the program and data memory. Each of the two Integrated Device's IDT71V016 devices is addressable in 64K 16-bit words. Third, a microphone preamplifier/antialiasing circuit and a Linear Technology LTC1605 16-bit sampling ADC was added to compensate for the AC01 CODEC's on the DSKplus (more discussion on this below). Last, a Xilinx FPGA was outfitted on the board to allow experimentation with alternate interconnection networks besides the TDM. In addition, the FPGA could include a custom Direct Memory Access (DMA) controller to write samples from the A/D straight into a circular buffer in memory. The DPSA daughter card is shown in Fig. 5.2, and the primary and daughter card coupled together in Fig. 5.3.

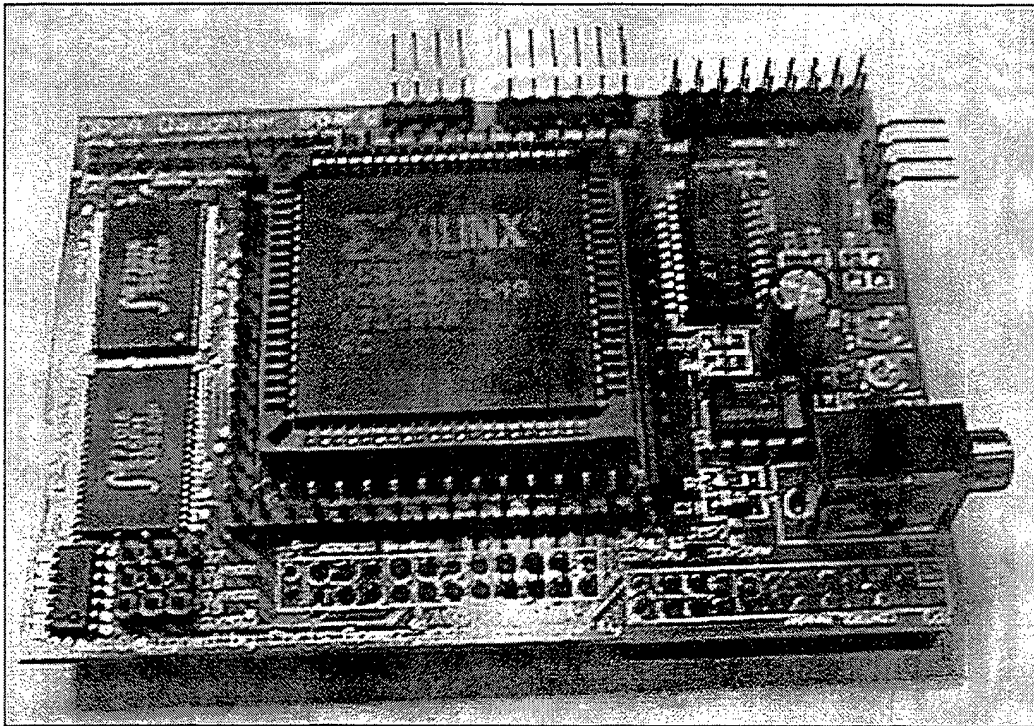


Fig. 5.2 The DPSA Daughter Card

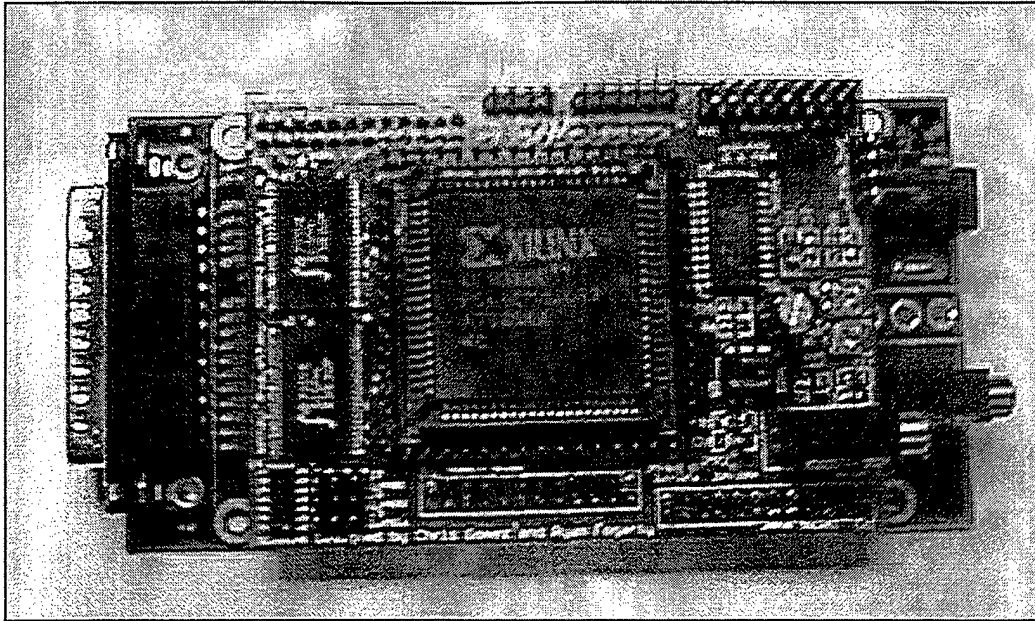


Fig. 5.3 A Complete DPSA Prototype Node

5.2. Communications Implementation

The Message Passing Interface (MPI) is a standardized set of communication primitives that may be used to build an application program interface (API) to coordinate parallel processes. All parallel beamforming algorithms were coded using MPI for the communication and coordination between intelligent array nodes. In order to run these applications on the TMS320C542 multicomputer prototype, an MPI implementation was created to interface to the TDM ports. The MPI implementation is designed to correctly transmit high-level MPI packets in the format needed for the TDM port. On the receiving end, the implementation includes the interrupt service routine to handle incoming raw data from the TDM and a receiving routine for the beamformer program to find and read the original high-level MPI data.

The implementation for *MPI_Send* takes as input the data the beamformer wants to transmit, the destination node, a tag used later for matching purposes, and the length of the data. Using the TDM, each node in the array is allocated a time slot for transmission. Each slot holds a single 16-bit word. The MPI implementation transmits a word of the data at a time, prepended by the size and tag values, waiting for its slot time between words.

On the receiving end, the implementation provides an interrupt service routine to handle data incoming from the TDM. When a word is received in any slot, the service routine interrupts the beamformer process. It determines the source of the data by noting the slot in which the data was received. The implementation also keeps a variable for maintaining the status of communication with each of the other nodes. This status will indicate to the interrupt service routine whether the word received from the TDM is the start of a new MPI packet or is a continuation of previous communication from that source node. If the received word is a new packet, the service routine will search for an empty location in the implementation's buffer space. Via the status variable for the source node, the implementation will indicate that all remaining words later received from that node within the same MPI packet are to be placed in the discovered buffer location. After completing this process for the received word, the service routine will return execution to the beamformer process.

When the beamformer needs to retrieve data from remote nodes to its memory space, it calls the *MPI_Recv* function. This function will search the MPI implementation's buffer space for packets matching the beamformer's search criteria. If it finds a completed MPI packet which matches, the data is copied into the beamformer's memory space. If no such packet is found or if the packet has words that have not yet arrived, the function will stay in a loop until the receive can be completed.

With respect to the detailed implementation of the interrupt service routine for receiving, three methods were studied. The first method organizes the implementation's buffer space into several records, each of which includes bits indicating whether the record is empty, contains an incomplete packet, or contains a complete packet. Such an organization simplified the *MPI_Recv* function but imposed demands onto the interrupt service routine. For example, the service routine needed to search the buffer space for free records. Doing so made execution of the service routine take longer than the period between successive words needing to be retrieved from the TDM bus.

The second method, which is the method implemented thus far, simplifies the interrupt service routine by using circular buffers, as shown in Fig. 5.4. The implementation at a receiver will have a circular buffer for each possible source, an index to the beginning of the circular buffer, and an index to the end. When a word is received on the TDM from a particular source, the interrupt service routine will automatically place the data at the logical top of the circular buffer for that source node and increment the end pointer. Thus, the service routine saves time by not searching for free slots. When the beamformer wishes to receive an MPI packet, the *MPI_Recv* function will linearly search through the circular buffer for the desired source. It will determine the location, size, and matching criteria for each packet as it reads the circular buffer. When the function matches the first packet to the desired criteria, it will copy the data to the beamformer's memory space and change the logical beginning of the circular buffer to point past the packet. If the matching packet is in the interior of the circular buffer, that section of the buffer will remain empty after the data has been copied. Eventually, the beamformer will request the first packet in the buffer, and the beginning pointer will be incremented. The entire process consists of the interrupt service routine augmenting the logical top of the circular buffer with incoming data and the *MPI_Recv* function playing catch-up by copying messages out of the logical bottom of the circular buffer.

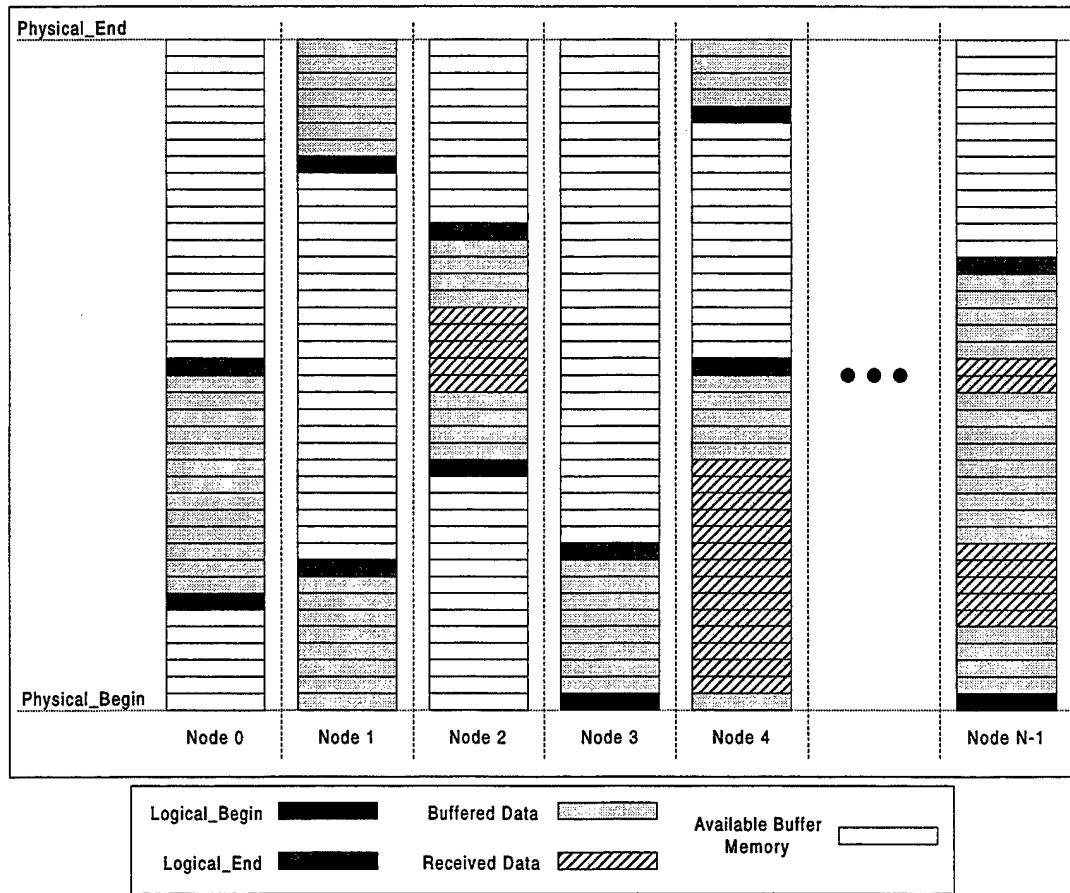


Fig. 5.4 Circular Buffer Structure at Each Node for TDM Communications

A third method, in which all data from all incoming nodes are placed into the same circular buffer, is left for future work. In this method, a second, more robust buffer would be implemented at a higher layer. Raw data from the unified circular buffer would be organized into the high-level buffer whenever convenient for the beamformer application. This method would further decrease the complexity of the interrupt service routine. As the current implementation stands, the interrupt service routine takes the majority of the available time between successive words arriving on the TDM. As such, there is no time for useful computations between receives, resulting in poor overlap between communication and computation at a receiving node.

5.3. Performance Results

In order to gauge the performance of the above described parallel and distributed array prototype, parallel beamforming applications were executed on it and timed. The algorithms selected were the coarse-grain and medium-grain split-aperture parallel beamformers, which were run on up to eight prototype nodes. In addition, the sequential split-aperture algorithm executing on a single TMS320C542 was used as the baseline for comparison. All algorithms used the minimum-memory model for split-aperture beamforming in order to alleviate the stringent memory requirements of the prototype. The nodes were clocked at 10MHz and each node was allocated one-eighth of the TDM slots for transmission, yielding an outgoing bit rate of 312.5 kbps. Furthermore, each node can receive information at 312.5 kbps from each other node, yielding a maximum of just under 2.2 Mbps for the incoming bit rate for concurrent

communication with all seven other nodes. The development boards could have been run at up to 40MHz, though little significant change would be seen in the relative speedup results since the TDM ports scale with processor speed.

Fig. 5.5 shows the scaled speedup of the two versions of the parallel beamformer over the sequential baseline. By scaled speedup, we refer to the fact that the sequential baseline for a 4-node parallel algorithm is the sequential program operating on input data from 4 array nodes. Similarly, the 6-node parallel algorithms are compared to the sequential program for a system with 6 input data streams. Due to the memory requirements of the sequential program, the baseline for a system with 8 array nodes will not fit in the memory space of a single TMS320C542 development platform; therefore, the execution time used as the baseline for 8 nodes is extrapolated from the 4-node and 6-node sequential execution times.

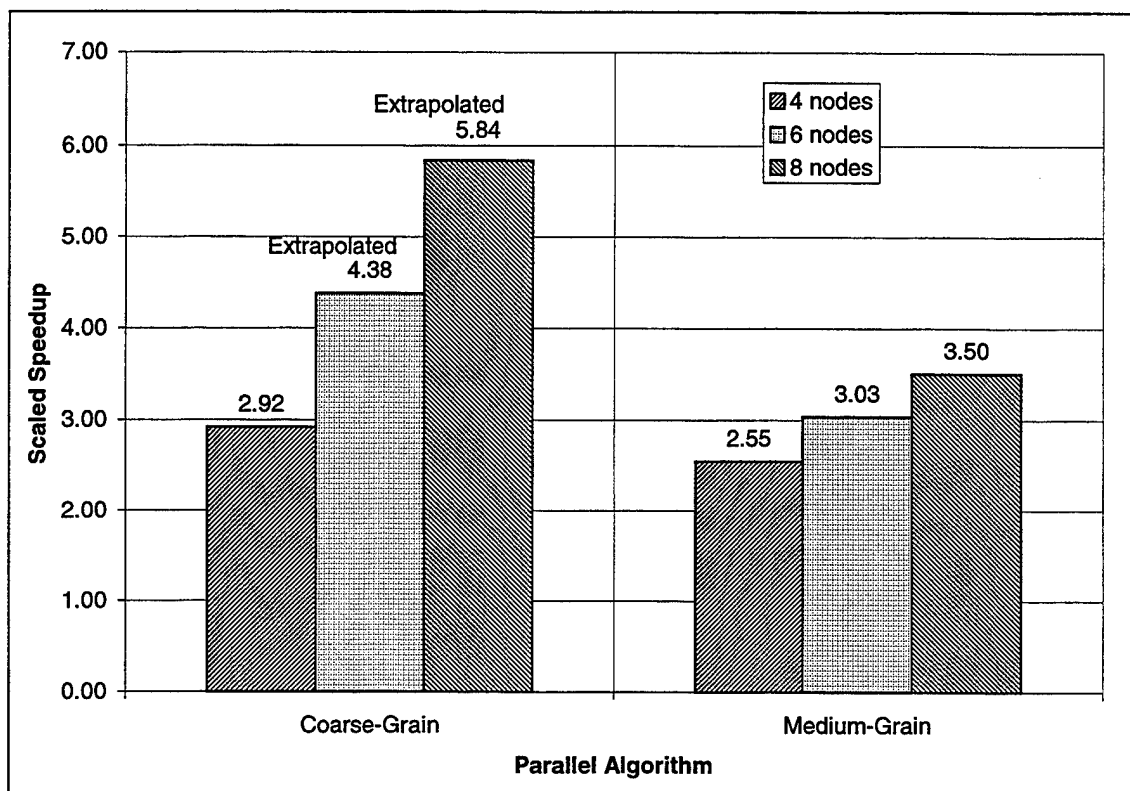


Fig. 5.5 Parallel Split-aperture Performance Results for the DPSA Prototype

The results from the experiments with the coarse-grain algorithm are shown on the left of the figure. Because the algorithm implements a full iteration of the split-aperture beamformer at each node, pipelining them to achieve speedup, the memory requirements are as large as that of the sequential program. Thus, the 8-node coarse-grain split-aperture algorithm could not be run on the development prototype. In addition, since the MPI implementation requires significant memory space for its buffers, the 6-node configuration of this algorithm was also not capable of fitting in the limited memory space. Previous analytical work completed for this project has shown that one can expect constant parallel efficiency from the coarse-grain algorithm if implemented well. Therefore, the above figure includes the experimental speedup observed for the 4-node coarse-grain split-aperture algorithm. The 73-percent parallel efficiency of this configuration was used to extrapolate the 6-node and 8-node speedups also shown in the figure.

Despite the fact that the prototype uses a limited 16-bit fixed-point processor and a bus-based communications network, the coarse-grain algorithm shows speedup rivaling that previously accomplished on clusters of high-performance RISC workstations.

On the right half of the figure, the results from the experiments with the medium-grain algorithm are shown. In each of these cases, the algorithm fit into the memory of the development boards. Indeed, one of the major advantages of the medium-grained algorithm is its efficient use of memory. The algorithm supplies speedups ranging from 2.5 to 3.5 on prototypes between 4 nodes and 8 nodes in size. Coinciding with expectations that the medium-grain algorithm is less efficient than the coarse-grain algorithm, the experimental parallel efficiencies for the medium-grain split-aperture beamformer fall below 50 percent for 8 nodes.

5.4. Conclusions

The DPSA prototype system is comprised of eight commercially available fixed-point digital signal processing development boards. The prototype is intended to illustrate the feasibility of distributing algorithm work across multiple array nodes for low-power, low-cost implementation of sonar beamforming. To extend the utility of the development boards, a daughter card was created for each node in the prototype for this project to provide increased memory and support future improvements in the interconnection network. Parallel processing experiments were conducted on the prototype using the split-aperture parallel algorithms developed for this project. The results proved to be encouraging for the possibility of using such a distributed parallel sonar array to provide speedup on the order of the expected speedup and the speedup previously achieved on high-performance networks of workstations.

The completion of this prototype opens doors to new opportunities in prototyping of distributed parallel sonar arrays. First, with the use of the FPGA on the daughter card, an interconnection network more appropriate for a sonar array architecture may be implemented, yielding a better understanding of the performance of an autonomous array. Such a network could not only provide more realistic prototyping of a ring or bidirectional-array network, but could also provide improved performance for the parallel beamformers. Second, more advanced beamforming algorithms, such as adaptive methods and matched-field processing, are slated for future implementation on the prototype.

6. FUTURE RESEARCH

As before, quiet submarine threats and high clutter in the littoral undersea environment demand that higher-gain acoustic sensors be deployed for undersea surveillance. The effect of this trend is high-element-count sonar arrays with increasing data rates and associated signal processing. These autonomous passive sonar array technologies face limitations which include low fault-tolerance due to single points of failure and computational complexity that cannot be supported in real-time by conventional means. Moreover, these limitations are especially evident with the continuing development of higher-fidelity acoustics algorithms such as adaptive and matched-field processing. These demands are further complicated by the limits on processor performance, memory capacity, etc. associated with low-power, autonomous sonar arrays.

Therefore, future research directions in this area will focus on fault-tolerant distributed and parallel processing techniques to decrease cost and improve performance and reliability of large, autonomous, battery-powered, disposable sonar arrays for these more advanced forms of beamforming. Particular goals include the development of new algorithms and supportive architectures for parallel in-array processing of adaptive and match-field processing on low-power, autonomous sonar arrays forming embedded distributed systems. The three-year approach identified for this future research program accentuates hardware-software interaction, architecture-algorithm mapping, CAD-based rapid virtual prototyping via high-fidelity network, architecture and interface modeling, and prototype development using an existing experimental array testbed.

In the first year of a new three-year program beginning in FY99, the emphasis will be on the analysis, selection, parallelization, and evaluation of adaptive beamforming (ABF) algorithms and supportive distributed system architectures. Building on previous success in FY98 with parallelization of SA-CBF algorithms, a host of decomposition, partitioning, and mapping activities will be undertaken. The goal will be to study, evaluate, and demonstrate the processing and memory characteristics of ABF algorithms and their constituent algorithmic elements (e.g. matrix inversion) in terms of baseline sequential forms and new parallel forms.

In the second year, work will continue on the parallelization of advanced ASW algorithms and supportive architectures in terms of more advanced ABF examples and initial basic examples with matched-field tracking (MFT). As in the first year of the program, new parallel algorithms will be developed, analyzed, and evaluated in terms of their realization on low-power autonomous sonar arrays. Similarly, in the third and final year of the new program, more advanced examples of MFT will be parallelized and evaluated along with initial basic examples with algorithms for matched-field processing (MFP).

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE 29 Jan 99	3. REPORT TYPE AND DATES COVERED Annual (1 Jan 98 - 31 Dec 98)	
4. TITLE AND SUBTITLE Parallel and Distributed Computing Architectures and Algorithms for Fault-Tolerant Sonar Arrays (Annual Report #3)		5. FUNDING NUMBERS N00014-98-1-0188	
6. AUTHORS Alan D. George			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) HCS Research Laboratory Dept. of Electrical and Computer Engineering, University of Florida PO Box 116200, 320 Larsen Hall Gainesville, FL 32611-6200		8. PERFORMING ORGANIZATION REPORT NUMBER HCS-TR-99-1	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Office of Naval Research Ballston Centre Tower One 800 N Quincy Street Arlington, VA 22217-5660		10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES			
12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE		12b. DISTRIBUTION CODE	
13. ABSTRACT (<i>Maximum 200 words</i>) This report summarizes the progress and results of the third year of a three-year study whose goal is the use of fault-tolerant distributed and parallel processing techniques to decrease the cost and improve the performance and reliability of large, battery-powered, autonomous sonar arrays. In FY98, work has been completed for single-aperture conventional parallel beamforming (CBF), recently completed for split-aperture parallel beamforming (SA-CBF), and initiated for more advanced adaptive or optimal processing algorithms. A comprehensive set of performance experiments has been conducted with the new CBF and SA-CBF parallel algorithms on both clustered workstation testbeds and simulated array architectures, and the results have illuminated the strengths and weaknesses of the algorithms. Several of the new CBF and SA-CBF parallel algorithms developed in this project have been shown to have promising performance and scalability characteristics in terms of processing and memory capacity. In addition, to support sensitivity studies for the mapping of parallel algorithms to array architectures, a new rapid virtual prototyping tool for the design and analysis of distributed parallel sonar arrays has been completed and demonstrated. Finally, an 8-node distributed parallel array prototype has been designed, developed, fabricated, and tested, and promising preliminary results have been attained.			
14. SUBJECT TERMS distributed computing; parallel computing; computer networks; sonar arrays; beamforming algorithms; fault-tolerant computing		15. NUMBER OF PAGES	
		16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT