DISTRIBUTED PARALLEL PROCESSING TECHNIQUES FOR ADAPTIVE SONAR BEAMFORMING

ALAN D. GEORGE, JESUS GARCIA, KEONWOOK KIM, and PRIYABRATA SINHA

High-performance Computing and Simulation (HCS) Research Laboratory
Department of Electrical and Computer Engineering, University of Florida
P.O. Box 116200, Gainesville, FL 32611-6200

Received 27 July 1999
Revised 29 March 2000

Quiet submarine threats and high clutter in the littoral environment increase computation and communication demands on beamforming arrays, particularly for applications that require in-array autonomous operation. By coupling each transducer node in a distributed array with a microprocessor, and networking them together, embedded parallel processing for adaptive beamformers can glean advantages in execution speed, fault tolerance, scalability, power, and cost. In this paper, a novel set of techniques for the parallelization of adaptive beamforming algorithms is introduced for in-array sonar signal processing. A narrowband, unconstrained, Minimum Variance Distortionless Response (MVDR) beamformer is used as a baseline to investigate the efficiency and effectiveness of this method in an experimental fashion. Performance results are also included, among them execution times, parallel efficiencies, and memory requirements, using a distributed system testbed comprised of a cluster of workstations connected by a conventional network.

1. Introduction

Beamforming is a method of spatial filtering and is the basis for all array signal processing. Beamforming attempts to extract a desired signal from a signal space cluttered by interference and noise. Specifically, sonar beamforming uses an array of spatially separated sensors, or hydrophones, to acquire multichannel data from an undersea environment. The data is filtered spatially to find the direction of arrival of target signals and to improve the signal-to-noise (SNR) ratio. An array can be configured arbitrarily in three dimensions, but this paper assumes a linear array with equispaced nodes designed to process narrowband signals.

Adaptive beamforming (ABF) optimizes a collection of weight vectors to localize targets, via correlation with the data, in a noisy environment. These weight vectors generate a beampattern that places nulls in the direction of unwanted noise (i.e. signals, called interferers, from directions other than the direction of interest). As opposed to conventional beamforming (CBF) techniques that calculate these weight vectors independently of the data, an adaptive beamformer uses information about the cross-spectral density matrix (CSDM) to compute the weights in such a way as to improve the beamforming output. Numerous ABF algorithms have been proposed in the literature[1-4]. This paper uses as its baseline the basic unconstrained Minimum Variance Distortionless Response (MVDR) algorithm as presented by Cox *et al.*[5] MVDR is an optimum beamformer that chooses weights to minimize output power subject to a unity gain constraint in the steering direction. The steering direction is the bearing that the array is "steered" toward to look for a particular incoming signal. A mono-frequency, unconstrained MVDR algorithm was selected as the baseline for this study since it is a well-known problem of sufficient complexity to demonstrate the effectiveness of the distributed parallel techniques developed for adaptive beamforming.

This paper introduces a novel method for the parallel processing of adaptive beamformers in a distributed fashion that scales with problem and system size on a linear array. This technique is ultimately

targeted for in-array processing, where each node in the array consists of a hydrophone coupled with a low-power DSP microprocessor. The microprocessors are connected via a point-to-point communications network. The parallel nature of such a sonar array eliminates a single point of failure inherent in arrays with a single processor, making it more reliable. By harnessing the aggregate computational performance, parallel processing on a distributed sonar array holds the potential to realize a variety of conventional, adaptive, and match-field algorithms for real-time processing.

Several studies in the literature on computational algorithms for sonar signal processing have exploited the increased computational power attained by a parallel system that scales with the problem size. George *et al.*[6] developed three parallel algorithms for frequency-domain CBF. George and Kim[7] developed a coarse-grained and a medium-grained parallel algorithm for split-aperture CBF (SA-CBF). Both of these sets of algorithms were designed for distributed systems composed of networked processors each with local memory. Banerjee and Chau[8] proposed a fine-grained MVDR algorithm for a network of general-purpose DSP processors using a QR-based matrix inversion algorithm. Their results, obtained through the use of an Interprocessor Communication (IPC) cost function, show that when the number of processors matches the problem size, the interprocessor communication costs exceeds the computational savings.

Most of the work conducted to date on parallel adaptive algorithms has been targeted for systolic array implementations[9-13]. These parallel algorithms exhibit real-time processing capabilities but are designed for implementation on a single VLSI processor and therefore cannot easily scale with varying problem and array size. Moreover, these systolic methods are inherently fine-grained algorithms that are not generally appropriate for implementation on distributed systems because of the communication latencies inherent in such architectures. One of the major contributors to execution time in MVDR is a matrix inversion that must be performed upon the CSDM with each iteration of beamforming. Many fine-grained algorithms exist for parallel matrix inversion[14-19]. However, like systolic array algorithms, these algorithms generally require too much communication to be feasible on a distributed system.

The distributed method for parallel adaptive beamforming presented in this paper addresses both the computation and communication issues that are critical for achieving scalable performance. Beamforming iterations are decomposed into stages and executed across the distributed system via a loosely coupled pipeline. Within each pipelined iteration, the tasks that exhibit the highest degree of computational complexity are further pipelined. Moreover, data packing is employed to amortize the overhead and thereby reduce the latency of communication, and multithreading is used to hide much of the remaining latency.

A theoretical overview of MVDR is given in Section 2. In Section 3, the baseline sequential algorithm is examined in terms of its individual tasks and their performance. In Section 4, the novel set of distributed processing techniques is presented in terms of a parallel MVDR algorithm. A performance analysis comparing the parallel algorithm and the sequential algorithm in terms of execution time, speedup and efficiency is given in Section 5. Finally, Section 6 concludes with a summary of the advantages and disadvantages of the parallel algorithm as well as a synopsis of the results and directions for future research.

2. MVDR Overview

The objective of beamforming is to resolve the direction of arrival of spatially separated signals within the same frequency band. The sources of the signals are located far from the sensor array. Therefore, the propagating wave is a plane wave and the direction of propagation is approximately equal at each sensor. We also neglect the effect of multipath propagation caused by reflection. Multiple sensors in an array are used to capture multichannel data that is spatially filtered by weights assigned to each sensor output. The weights, also called the aperture function, form a beampattern response, which is a function of the number of sensors, sensor spacing, and wave number given by $k = \omega / c$, where ω is the center frequency of the

signal and c is the propagation velocity for the signal. The sensor outputs are multiplied by the weights and summed to produce the beamformer output. The beamformer output is a spatially filtered signal with an improved SNR over that acquired from a single sensor.

Frequency-domain beamforming offers finer beam-steering resolution and is more memory efficient than time-domain beamforming. George *et al.*[6] showed that the execution time of the Fast Fourier Transform (FFT) is negligible compared to the execution times of other stages in the beamforming process. Hence, for this research, we assume that all time-domain data has been Fourier transformed. As a baseline, the data is processed using a single frequency bin since, as will become more clear later, the parallel method presented in this paper is readily extensible to multiple frequency bins.

Fig. 1 shows the structure of a basic adaptive beamformer with a signal arriving from angle θ. There are n sensors with frequency-domain outputs $x_0, \ldots, x_{n-1}$. The weights, w_i, in the figure are intersected by an arrow to illustrate their adaptive nature. Denoting the column vector of frequency-domain sensor outputs as $\boldsymbol{x}$ and the column vector of weights as $\boldsymbol{w}$ with entries w_i, the scalar output, y, can be expressed as

$$y = \boldsymbol{w}^* \boldsymbol{x} \tag{2.1}$$

where the * denotes complex-conjugate transposition.

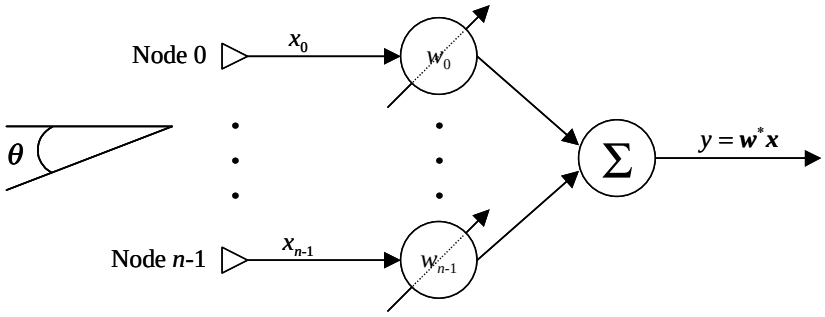


Fig. 1. Narrowband ABF structure.

CBF uses a delay and sum technique to steer the array in a desired direction independently of the data. The array is steered by properly phasing the incoming signal and summing the delayed outputs to produce the result. The steering vector, $\boldsymbol{s}$, is calculated from the frequency and steering direction, which when multiplied by the incoming signal will properly phase the array. The equation for $\boldsymbol{s}$ is given by

$$s = [1, \quad e^{-jkd\sin(\theta)}, \quad e^{-j2kd\sin(\theta)}, \cdots, \quad e^{-j(n-1)kd\sin(\theta)}]^T \tag{2.2}$$

where k is the aforementioned wave number, n is the number of nodes, and d is the distance between the nodes in the array. For a detailed discussion of CBF and the formation of the steering vectors, the reader is referred to Clarkson[21].

The cross-spectral density matrix, R, is the autocorrelation of the vector of frequency-domain sensor outputs

$$R = E\{\boldsymbol{x} \cdot \boldsymbol{x}^*\}. \tag{2.3}$$

The matrix R is also referred to as the covariance or correlation matrix in time-domain algorithms. The output power per steering direction is defined as the expected value of the squared magnitude of the beamformer output:

$$P = E\{|y|^2\} = \boldsymbol{w}^* E\{\boldsymbol{x} \cdot \boldsymbol{x}^*\} \boldsymbol{w} = \boldsymbol{w}^* R \boldsymbol{w}. \tag{2.4}$$

In CBF, the weight vector $\boldsymbol{w}$ is equal to $\boldsymbol{s}$, the steering vector.

MVDR falls under the class of linearly constrained beamformers where the goal is to choose a set of weights, $\boldsymbol{w}$, that satisfy

$$\boldsymbol{w}^{*}\boldsymbol{s} = g \tag{2.5}$$

which passes signals from the steering direction with gain g while minimizing the output power contributed by signals, or interferers, from other directions. In MVDR, the gain constant g equals one. Thus, by combining Eqs. (2.4) and (2.5) and assuming that $\boldsymbol{s}$ is normalized (i.e. $\boldsymbol{s}^{*}\boldsymbol{s} = n$), we solve for the weights from

$$\underset{\boldsymbol{w}}{Min}\, P = \boldsymbol{w}^{*}\, R\, \boldsymbol{w} \text{ constrained to } \boldsymbol{w}^{*}\boldsymbol{s} = 1. \tag{2.6}$$

Using the method of Lagrange multipliers, it is found that the solution for the weight vector in Eq. (2.6) is

$$\boldsymbol{w} = \frac{R^{-1}\boldsymbol{s}}{\boldsymbol{s}^{*}R^{-1}\boldsymbol{s}}. \tag{2.7}$$

By substituting Eq. (2.7) into Eq. (2.6), we obtain the scalar output power for a single steering direction as:

$$P = \frac{1}{\boldsymbol{s}^{*}R^{-1}\boldsymbol{s}}. \tag{2.8}$$

Thus, the MVDR algorithm optimally computes the weight vectors depending on the sampled data. The result is a beampattern that places nulls in the direction of strong interferers.

As an example, Fig. 2 shows the beampatterns and beamforming power plots from a conventional beamformer and an MVDR beamformer. The data in this example was created for one frequency bin, f =30 Hz, where the output of each sensor has a signal plus noise component. The amplitude of the noise is normally distributed with zero mean and unit variance, with an SNR of 25 dB. Isotropic noise is modeled as a large number of statistically independent stochastic signals arriving at the array from all directions.

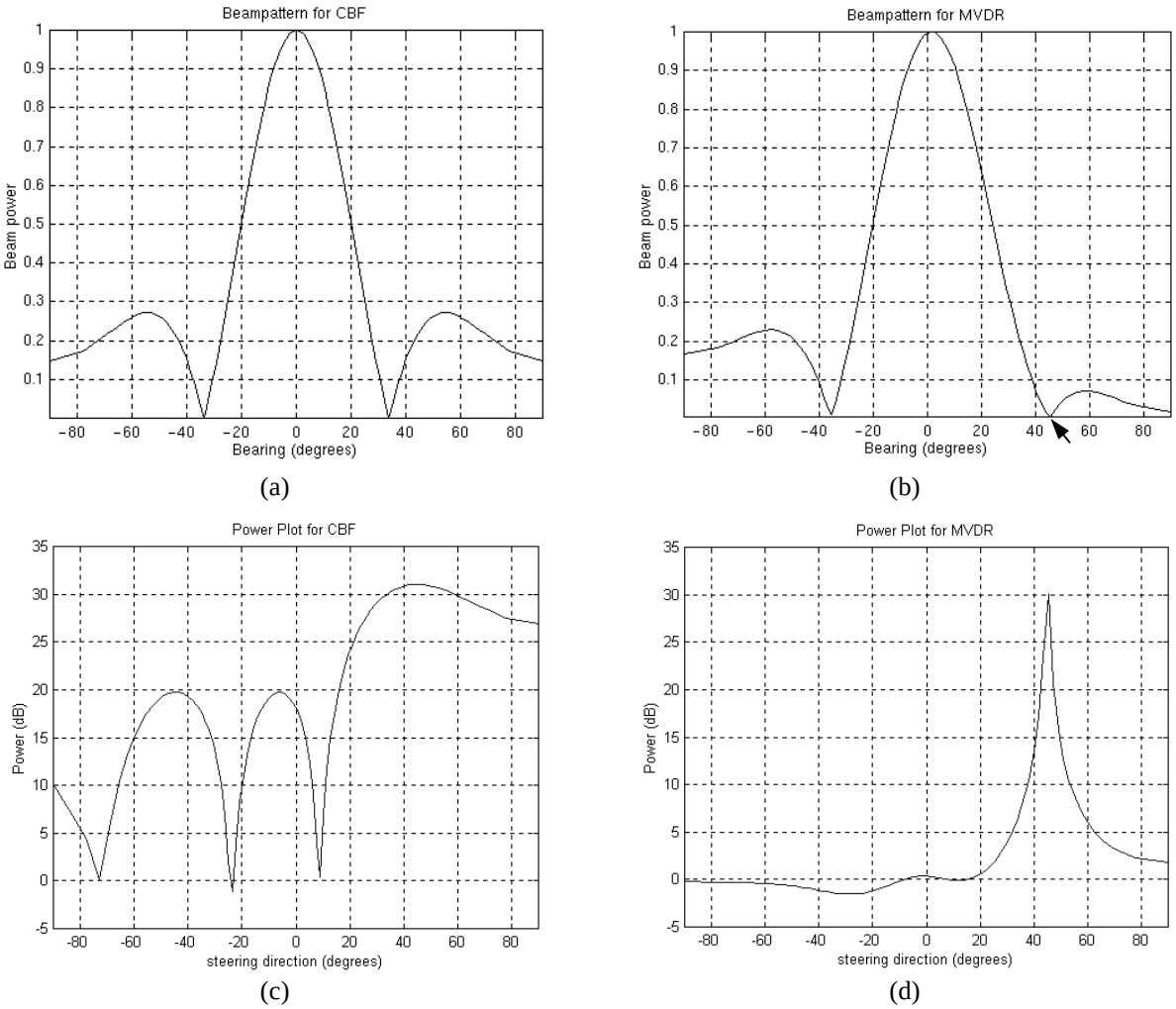


Fig. 2. Array response and power plots for 4 nodes, 91 steering directions, and incoming signal at $\theta = 45°$. CBF *(a)* and MVDR *(b)* plots of beampattern for the array when steered toward broadside. Arrow in *(b)* points to null steered at interference arriving at 45°. Resulting log-scale power plots for CBF *(c)* and MVDR *(d)*.

Since the weight vectors for CBF are calculated independently from the sampled data and the only *a priori* knowledge of the desired signal is the bearing, CBF ignores possible interferers from other directions. Fig. 2(a) shows the beampattern created by a CBF algorithm looking for signals at broadside (i.e. $\theta = 0°$) with a source of interference at $\theta = 45°$. The result is a beampattern that passes signals at broadside with unity gain while attenuating signals from other angles of incidence. At $\theta = 45°$, the beampattern exhibits a side lobe that does not completely attenuate the interfering signal, which will result in a "leakage" of energy from the signal at 45° into broadside. Conversely, the weight vectors in MVDR depend explicitly on the sampled data and can therefore steer nulls in the direction of interferers, as indicated by the arrow in Fig. 2(b). In this figure, the main lobe is at broadside (i.e. the steering direction), and at $\theta = 45°$ there is a null that minimizes the energy contributed from the interference in that direction, thereby preventing leakage.

Changing the number of nodes and the distance between the nodes can change the beampattern of the array. Clarkson[21] gives a detailed analysis of how changing these array parameters affects the spatial bandwidth of the main lobe, the side lobe level, and roll-off of the side lobes. Figs. 2(c) and 2(d) show the

final beamforming results, for CBF and MVDR respectively, where the power found in each steering direction is plotted on a logarithmic scale to emphasize the artificial energy found in directions where there was no signal source. The relatively poor results in Fig. 2(c) are evidence of the energy contributed by the interference to other steering directions as this energy was passed, although attenuated, by the CBF beampattern. By contrast, the MVDR results in Fig. 2(d) exhibits a dramatic improvement due to the ability of MVDR to place nulls in the direction of strong signal interferences when not steering toward the direction of the interference.

3. Sequential MVDR Algorithm and Performance

Adaptive beamforming with MVDR is divided into three tasks. The first task is the *Averaging* of the CSDM. The second task is *Matrix Inversion*, which inverts the exponential average of the CSDM. The third task is *Steering*, which steers the array and calculates the power for each steering direction. Fig. 3 is a block diagram of the sequential algorithm.

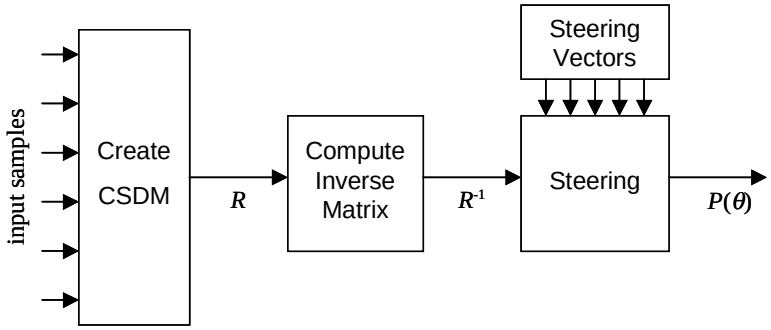


Fig. 3. Block diagram of the sequential MVDR algorithm.

Subsection 3.1 presents an overview of each task in the sequential algorithm, followed by performance results given in Subsection 3.2. The performance results will identify the computational bottlenecks of the sequential MVDR algorithm.

3.1. Sequential MVDR tasks

As defined in Eq. (2.3), the creation of the CSDM requires computations involving an infinite series with an infinite number of sample values, which is of course not feasible in a real application. The CSDM is estimated from the streaming input data and updated at each iteration using the exponential averaging method given by

$$\hat{R}(k+1) = \alpha\hat{R}(k) + (1-\alpha)\{x(k+1)\cdot x(k+1)^H\} \tag{3.1}$$

where k+1 denotes the current sample, k denotes the previous sample, and α is the exponential averaging parameter. This average provides an estimate of the CSDM, assuming that the signal data is stationary.

The *averaging* task consists of n^2 complex multiplications and additions followed by n^2 divisions, to compute the estimate of the CSDM, which results in a computational complexity of $O(n^2)$. The number of operations required in estimating the CSDM could be reduced by making use of the Hermitian property of the CSDM. However, the computational complexity of this task would still remain at $O(n^2)$.

The inversion algorithm we use for the *matrix inversion* task is the Gauss-Jordan elimination algorithm with full pivoting, adapted from Press *et al*.[20] Gauss-Jordan elimination for matrix inversion assumes that we are trying to solve the equation

$$A \cdot X = I \tag{3.2}$$

where A is the matrix to be inverted, X is the matrix of unknown coefficients which form the inverse of A, and I is the identity matrix. This inversion technique uses elementary row operations to reduce the matrix A to the identity matrix, thereby transforming the identity matrix on the right-hand side into the inverse of A. The method used here employs full pivoting, which involves doing row *and* column operations in order to place the most desirable (i.e. usually the largest available element) pivot element on the diagonal. This algorithm has a main loop over the columns to be reduced. The main loop houses the search for a pivot element, the necessary row and column interchanges to put the pivot element on the diagonal, division of the pivot row by the pivot element, and the reduction of the rows. The columns are not physically interchanged but are relabeled to reflect the interchange. This operation scrambles the resultant inverse matrix so a small loop at the end of the algorithm is needed to unscramble the solution. The pivoting operation has been used to make the inversion algorithm numerically stable.

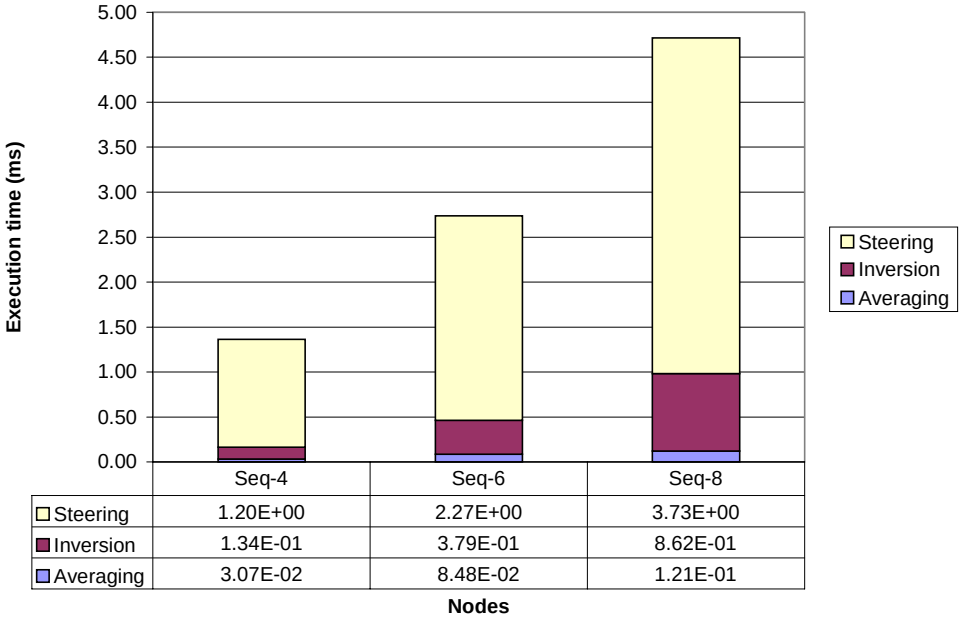
Gauss-Jordan elimination was selected because it is computationally deterministic in that the number of operations required to complete the inversion is fixed and known *a priori* independent of the data, and thus the overhead of dynamic task scheduling is avoided. An inversion algorithm without such characteristics would require run-time decisions to be made regarding the assignment of tasks to processors, thereby adding to the overhead of the parallel processing and imposing limits on attainable speedup. For details on the structure of the Gauss-Jordan elimination algorithm and its numerical stability, the reader is referred to Golub and Van Loan[22]. The complexity of the algorithm is $O(n^3)$. Gauss-Jordan elimination is particularly attractive for distributed parallel systems because the outer loop that iterates over the n columns of the matrix is easily decomposed over n stages and is therefore ideal for pipelining in a coarse-grained algorithm where the iterations of the loop are decomposed into separate stages. The memory requirements are minimized by building the inverse matrix in the place of A as it is being destroyed.

The *steering* task is responsible for steering the array and finding the output power for each of the steering directions. The main operation in the *steering* task is the product of a row vector by a matrix then by a column vector, which results in $O(n^2)$ operations. This operation must be performed once for every steering direction, which increases the execution time linearly as the number of steering directions increases. Although with a fixed number of steering angles the computational complexity of the *steering* task is lower than that of the *matrix inversion* task as $n \rightarrow \infty$, it will dominate the execution time of the algorithm for sonar arrays where the number of nodes is less than the number of steering angles. The performance analysis in the next subsection will illustrate the impact of each of the tasks in the sequential algorithm.

3.2. Sequential MVDR performance

Experiments were conducted to examine the execution time of the sequential model and its three inherent tasks. The sequential algorithm was coded in C, compiled under Solaris 2.5, and then executed on an Ultra-1 workstation with a 167MHz UltraSPARC-I processor containing 32kB of on-chip cache memory, 512kB of secondary cache memory, and 128MB of main memory. This platform was selected from the many types of workstations in the testbed as one that is reasonably representative of the performance envelope that can be expected from embedded DSP microprocessors that will be deployed in the next several years in terms of processor performance, on-chip and off-chip memory capacity, DMA support, etc. To study the effects of problem size, the program was executed for 4, 6, and 8 sensor nodes, each using 45 steering directions. The execution time for each task was measured separately and averaged over many

beamforming iterations. The results are shown in Fig. 4. Each stacked bar is partitioned by the tasks of MVDR beamforming discussed above. The height of each stacked bar represents the average amount of time to execute one complete iteration of beamforming. As predicted, the *steering* task is the most computationally intensive since, in this case, the number of nodes is always less than the number of steering directions. The *matrix inversion* task follows while *averaging*, seen at the bottom of each stacked bar, is almost negligible.



	Seq-4	Seq-6	Seq-8
Steering	1.20E+00	2.27E+00	3.73E+00
Inversion	1.34E-01	3.79E-01	8.62E-01
Averaging	3.07E-02	8.48E-02	1.21E-01

Fig. 4. Average execution time per iteration as a function of array size for the baseline sequential MVDR algorithm with 45 steering directions and 576 iterations on an Ultra-1 workstation.

The array architecture assumed for the implementation of the sequential algorithm is one in which the sensor nodes collect data and send it to a front-end processor for beamforming. Communication latency is ignored in the performance analysis for the sequential algorithm, and the total execution times are based completely on the sum of the execution times of the three computational tasks.

The sequential algorithm was optimized for computational speed by pre-calculating the steering vectors and storing them in memory to be retrieved only when needed in the *steering* task. George and Kim[7] analyze the trade-offs between a minimum-calculation model and a minimum-memory model for a sequential split-aperture CBF algorithm. In their minimum-calculation model, the steering vectors and the inverse-FFT basis are calculated in an initial phase of execution and saved in memory to access when needed in the *steering* and *iFFT* processing tasks, respectively. The minimum-memory model conserves memory by computing these vectors on the fly as needed. Performance analyses indicate that the minimum-calculation model is five times faster than the minimum-memory model but requires twice as much memory capacity. In MVDR, there is no *iFFT* task and therefore no need to calculate and store an inverse-FFT basis. However, the SA-CBF and MVDR algorithms both require steering vectors in their *steering* tasks. Since, like MVDR, the *steering* task was found to be the dominant factor in the performance of the SA-CBF algorithm, we expect similar computation-vs-memory tradeoffs with the MVDR algorithm. As execution time is the primary focus of the performance analyses in this paper, we

chose to compromise memory space for execution speed by pre-calculating the steering vectors and storing them in memory to access when needed in the *steering* task.

The following section describes the novel techniques and the resulting distributed-parallel MVDR algorithm that achieves promising levels of speedup by decomposing, partitioning, and scheduling in pipeline stages the two most computationally intensive tasks, *steering* and *matrix inversion*. Section 5 analyzes the performance of the parallel algorithm and compares it with the purely sequential algorithm presented above.

4. Distributed-Parallel MVDR Processing

The level at which a computational task is partitioned among processors defines the granularity of the parallel algorithm. In a coarse-grained decomposition the computational tasks are relatively large with less communication among the processors. In a fine-grained decomposition the tasks are relatively small and require more communication. A medium-grained decomposition is a compromise between the two.

The lower communication overhead inherent in coarse- and medium-grained decompositions makes them the preferred methods when developing parallel algorithms for distributed systems. A coarse-grained decomposition for parallel beamforming is one that would assign independent beamforming iterations (i.e. the outermost loop in the sequential basis) to each processing node and then pipeline the beamforming tasks for overlapping concurrency of execution across the nodes. For instance, in a 4-node array, node 0 would be assigned iterations 0, 4, 8, 12, etc., node 1 would be assigned iterations 1, 5, 9, 13, etc., and so forth. By contrast, a medium-grained decomposition is one that would assign the same beamforming iteration to all nodes for simultaneous concurrency of execution, where each node would be responsible for computing the power results for a subset of the steering angles of interest. For instance, in a 4-node array that is beamforming with 180 steering angles, node 0 would be assigned the first 45 steering angles of all iterations, node 1 the second 45 steering angles, etc.

Coarse- and medium-grained decomposition algorithms have been developed for parallel CBF and SA-CBF by George *et al.*[6] and George and Kim[7], respectively. In both papers, the coarse-grained algorithm is called *iteration decomposition* and the medium-grained algorithm is called *angle decomposition*. The angle-decomposition method requires all-to-all communication to distribute the data among the nodes while the iteration decomposition method requires all-to-one communication to send the data to the scheduled node. The performance of iteration decomposition is generally superior to angle decomposition on distributed systems because it requires less communication among the nodes. However, since angle decomposition is not pipelined, it has a shorter result latency (i.e. the delay from the sampling of input data until the respective beamforming output is rendered) and makes more efficient use of memory by distributing the steering vectors across the nodes. Moreover, when the network in the distributed system is capable of performing a true hardware broadcast (i.e. when a sender need only transmit a single packet onto the network for reception by all other nodes), performance approaches that of iteration decomposition.

The smaller frequency of communication inherent in a coarse-grained decomposition makes it the most practical basis for a parallel MVDR algorithm designed for in-array processing on a distributed sonar array. A medium-grained algorithm for MVDR would require all nodes to invert the same matrix independently, causing undue computational redundancy. A coarse-grained algorithm avoids redundant computations by scheduling beamforming iterations for different data sets. Of course, the high frequency of communication inherent to fine-grained algorithms would make them impractical for implementation on a distributed array.

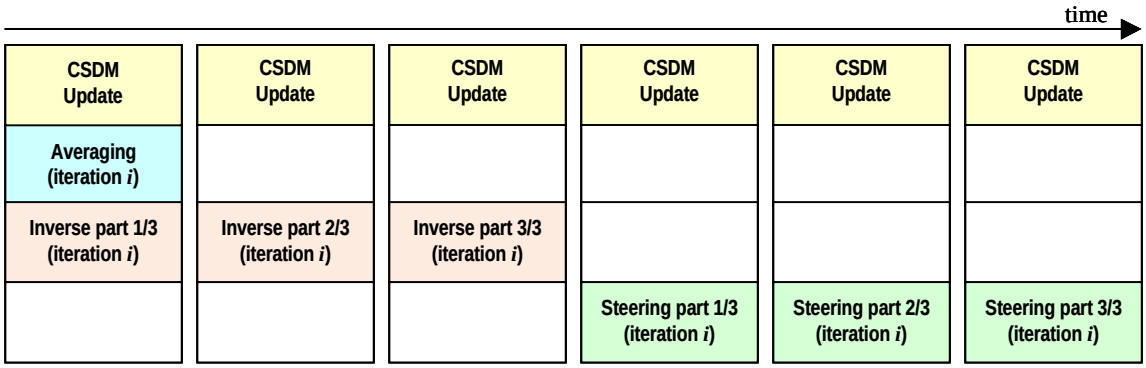
The techniques for distributed-parallel MVDR (DP-MVDR) beamforming that are presented in this section include two main components, a computation component and a communication component. The computation component uses round-robin scheduling of beamforming iterations to successive nodes in the array, and both staggers and overlaps execution of each iteration within and across nodes by pipelining. The communication component of the algorithm reduces the communication latency of all-to-all

communication with small amounts of data by spooling multiple data samples into a single packet. In so doing, a separate thread is employed to hide communication latency by performing the all-to-all communication while the main computational thread is processing the data received during the previous communication cycle. The next two subsections discuss the two algorithmic components in detail, followed by performance results in Section 5.

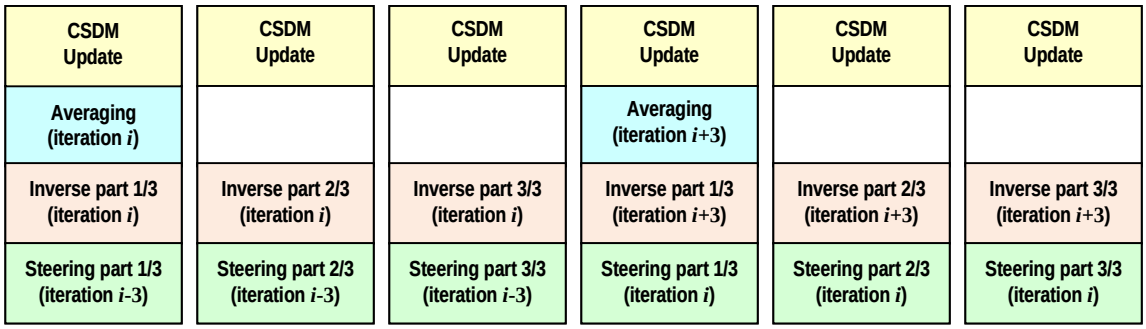
4.1 Computation component of the DP-MVDR algorithm

As reviewed in Section 3, each beamforming iteration of the sequential MVDR algorithm is composed of an *averaging* task, a *matrix inversion* task, and a *steering* task. Before the task of *matrix inversion* can begin, the exponential average of the CSDM ensemble must be computed. The *steering* task then uses the inverse of the CSDM average to steer for every angle of interest. In the DP-MVDR algorithm, each node in succession is scheduled to beamform a different data set in a round-robin fashion. For instance, node 0 would be assigned to process the first set of data samples collected by the array, node 1 the second set, and so forth, causing each node to beamform using every n^{th} data set, where n is the number of nodes.

Furthermore, the DP-MVDR algorithm also pipelines the *matrix inversion* and *steering* tasks within each of the beamforming iterations by decomposing each task into n stages. When a node is scheduled a new beamforming iteration, it computes the exponential average of the CSDM and executes the first stage of the *matrix inversion* task. The next n-1 stages are spent completing the computation of the matrix inversion and the following n stages perform the steering, thereby imposing an overall result latency of $2n$ pipeline stages. Meanwhile, the running ensemble sum of CSDMs is updated at the beginning of every pipeline stage. Fig. 5(a) illustrates the basic pipeline structure in terms of the stages of execution associated with a single iteration executing on a single node in a system of three nodes (i.e. $n = 3$). Since each node is scheduled a new beamforming iteration every n stages, but the result latency is $2n$, it follows that the computations for two iterations can be overlapped within each node by computing a *matrix inversion* stage of the current iteration i and the respective *steering* stage of the previously scheduled iteration i-n in the same pipeline stage. Fig. 5(b) illustrates the pipeline structure for all activities assigned to each node in the system.



(a) Stages of execution for a single node on a single beamforming iteration (n = 3)



(b) Stages of execution for a single node on multiple beamforming iterations (n = 3)

Fig. 5. Block diagram of the computation component of DP-MVDR in terms of a single beamforming iteration (a) and multiple iterations (b) executing on one of the nodes in the system (where n = 3).

As an example, consider the six stages of the pipeline in Fig. 5(b). In the first pipeline stage, the CSDM ensemble is updated with sample data from all nodes, the exponential average is computed for iteration i, and the output is used by the first phase of the matrix inversion. Meanwhile, in that same node, the first phase of the steering task is computed based on the results from the matrix inversion previously computed for iteration i-n. In the second pipeline stage, new data from all nodes is again used to update the CSDM ensemble while the second phase of the inversion for iteration i continues, along with the second phase of the steering task for iteration i-n. In the third pipeline stage, the next update takes place and the inversion for iteration i is completed. The steering for iteration i-n is also completed, thereby ending all processing for iteration i-n by producing the beamforming result. Finally, in the fourth, fifth and sixth pipeline stages on this node, the steering is performed for iteration i, while the first three stages are executed for the next assigned iteration, iteration i+n.

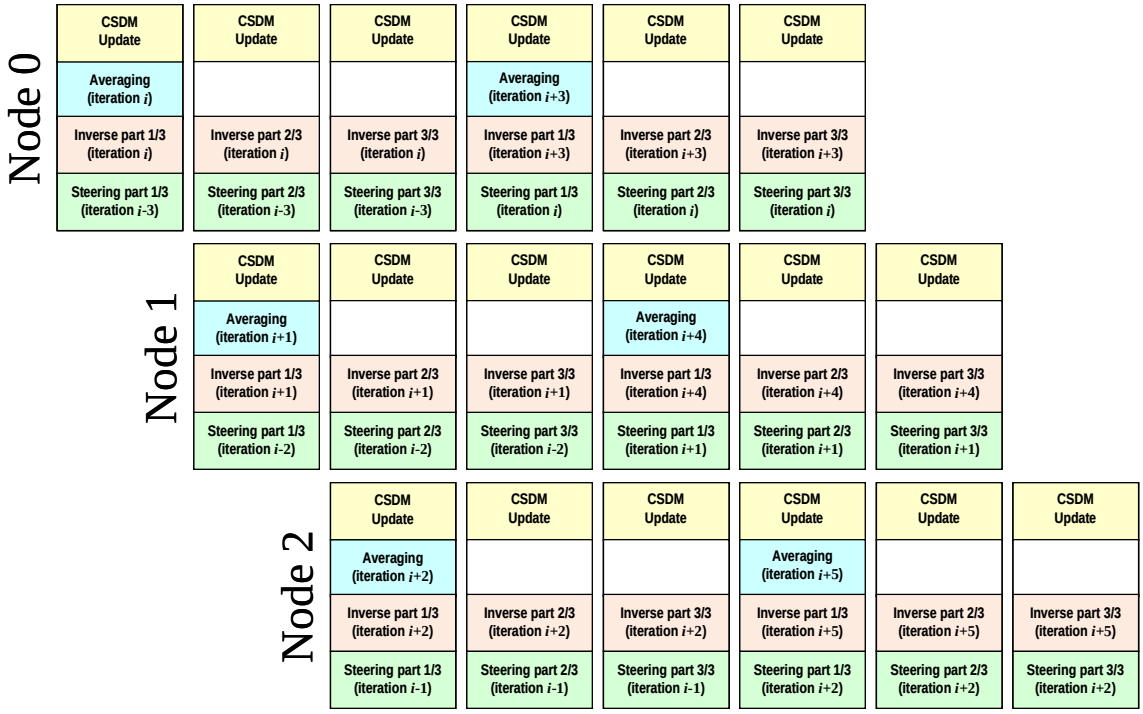


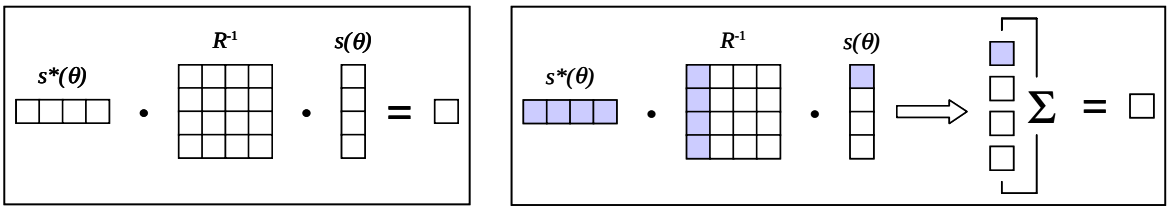
Fig. 6. Block diagram of the stages in the computation component of DP-MVDR in terms of all nodes in the system (where $n = 3$).

Once the pipeline has been initialized and has filled, the overlapping of iterations between consecutive nodes in the array allows for the output from one complete beamforming iteration to be available from the system at the end of every pipeline stage. This overlap makes the effective throughput of the system equal to the reciprocal of the execution time associated with a single stage in the pipeline. As an example, Fig. 6 shows the overlapping execution of all nodes in a system with three nodes. In this case, node 0 is responsible for beamforming iterations 0, 3, 6, …, i-3, i, i+3, etc. Similarly, node 1 is assigned iterations 1, 4, 7, …, i-2, i+1, i+4, etc. and node 2 is assigned iterations 2, 5, 8,…, i-1, i+2, i+5, etc. In general, for a system of n nodes, an arbitrary node j is assigned iterations j, n+j, 2n+j, 3n+j, etc. in a round-robin fashion. The decomposition schemes employed for the various tasks are discussed below.

In every pipeline stage the CSDM estimate is updated from the present data set collected from all the nodes in the system. Therefore, the computational complexity of the *averaging* task in the parallel algorithm remains the same as in the sequential algorithm, $O(n^2)$.

The Gauss-Jordan elimination algorithm loops for every column of the CSDM. Since the CSDMs always have n columns, a single loop of the algorithm is executed per pipeline stage. This form of decomposition decreases computational complexity from $O(n^3)$ in the sequential algorithm to $O(n^2)$ in the parallel algorithm for the *matrix inversion* task over n nodes.

Fig. 7 shows the flow of processing associated with the *steering* task in the sequential algorithm and its decomposed counterpart in the parallel algorithm. The product shown is the denominator of Eq. (2.8) and θ is the steering direction.



(a) (b)

Fig. 7. Sequential *steering* task (a) and decomposed *steering* task (b).

Fig. 7(b) shows the decomposition of the *steering* task in the DP-MVDR algorithm. The shaded elements denote the elements that are multiplied in one pipeline stage. The conjugate-transpose of the steering vector is multiplied by one column of the CSDM inverse, then by one element of the steering vector, for every steering direction, which results in $O(n)$ complexity. This process is repeated n times, one per stage, for each column of R^{-1} and each corresponding element of $\boldsymbol{s}(\theta)$. The results from each stage are accumulated until the multiplication is completed in the n^{th} stage. As evidenced in the computational performance experiments previously conducted with the sequential algorithm, the *steering* task in the DP-MVDR algorithm, while less computationally complex than matrix inversion as $n \to \infty$ for a fixed number of steering angles, dominates the execution time when n is less than the number of steering directions.

4.2 Communication component of the DP-MVDR algorithm

The CSDM update operation of Eq. (3.1) at the beginning of every iteration of beamforming requires that each node must receive a periodically updated local copy of every other node's data sample, where a data sample is the complex value of one frequency bin. In so doing, the nodes must perform an all-to-all communication of a single sample per node between each pipeline stage in order to meet the data requirements imposed by the MVDR algorithm. In a distributed array this communication pattern incurs a high degree of network contention and latency that might render the computation component of the parallel algorithm ineffectual.

To lower the impact of all-to-all communication, we use a technique called "data packing" where the nodes pack data (e.g. d samples) that would be needed in future stages and send that data as one packet. Data packing eliminates the need to perform an all-to-all communication cycle between each pipeline stage and instead it is performed every d stages. Data packing does not reduce the communication complexity of all-to-all communication, which is $O(n^2)$ in a point-to-point network where n is the number of nodes. However, due to the overhead in setup and encapsulation for communication, sending small amounts of data results in a high overhead to payload ratio, making it desirable to pack multiple data together to amortize the overhead and reduce the effective latency of communication.

In addition to the use of data packing to reduce latency, we use multithreaded code in order to hide latency by overlapping communication with computation. Using a separate thread for communication is comparable to the use of a DMA controller capable of manipulating data between the node's memory and the network interface in that, after setup, communication occurs in the background. In this scheme, while the nodes are processing their current data set, the communication thread is exchanging the data samples for the next set of iterations. Fig. 8 shows a block diagram of this method.

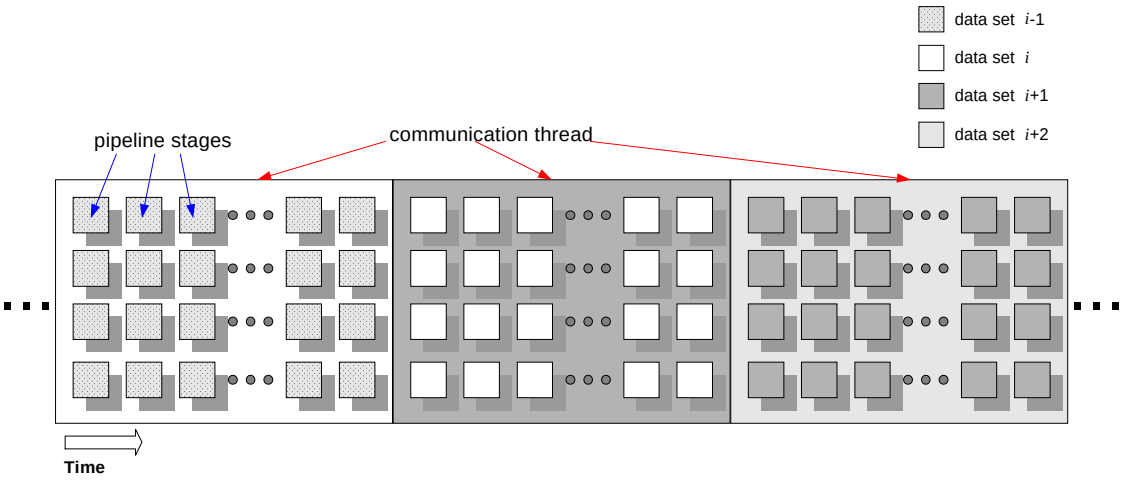


Fig. 8. Overlap of computation and communication threads.

In Fig. 8, the squares in the foreground represent pipeline stages of computation that overlap in time with the all-to-all communication cycles represented by the rectangles in the background. The different types of shading depict the sets of data that are being operated upon by each thread (e.g. the pattern in the background with the first cycle is repeated in the pipeline stages of the second cycle). Fig. 8 illustrates how the data that is currently being processed was transmitted and received during the previous communication cycle.

Several experiments were conducted on a testbed to determine the appropriate number of data elements per packet to transmit per communication cycle so that communication and computation can be overlapped, with minimum overhead in the computation thread. The testbed consisted of a cluster of Ultra-1 workstations (i.e. the same platform used in Section 3) connected by a 155Mb/s (OC-3c) Asynchronous Transfer Mode (ATM) network via TCP/IP. The parallel code was written in the C version of MPI[23]. MPI is a standard specification that defines a core set of functions for message-passing communication and parallel program coordination. All-to-all communication was achieved by calling multiple *send* and *receive* primitives. Each complex data sample consists of two, 8-byte floating-point values. Fig. 9 compares the amount of time each thread dedicates to each component in a full communication and processing cycle for different payload sizes per packet in an 8-node system.

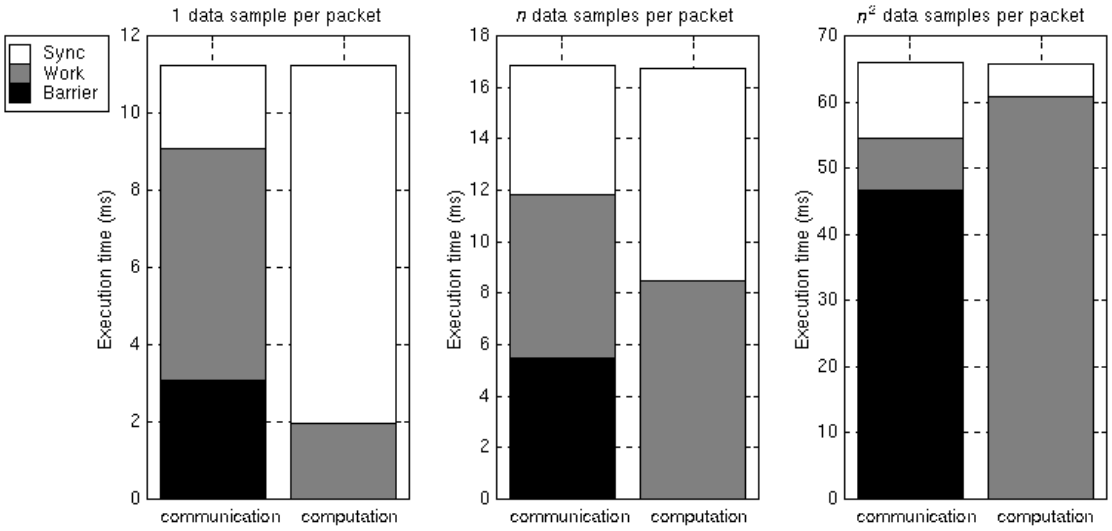


Fig. 9. Thread performance as a function of data samples per packet (n = 8).

A cycle in the communication thread consists of a barrier synchronization to control the all-to-all transaction between nodes, communication of d data samples that is the actual work of the thread, and thread synchronization within the node. A cycle in the computation thread consists of the processing of d data samples and thread synchronization. The synchronization time in the communication thread is the amount of time that it takes to tell the computation thread that it has completed the communication of the data, plus the amount of time it waits for the computation thread to tell it to begin a new communication cycle. The synchronization time in the computation thread is the amount of time it waits for the communication thread to finish, plus, the amount of time it takes to tell the communication thread to begin a new communication cycle. Thus, the thread that finishes its work first must block to wait for the other thread to finish. To achieve an optimal overlap of computation over communication, the goal is for the communication thread to finish its communication cycle while the computation thread is still processing data. Fig. 9 shows that for payload sizes of 1 and n data samples per packet, the computation thread finishes its work cycle before the communication thread finishes its work cycle, and thus maximum overlap is not achieved to completely hide the communication latency. When the payload size is n^2 data samples per packet, the amount of time the computation thread spends processing the data is greater than the amount of time the communication thread spends in sending the next set of data, thereby achieving complete overlap. The amount of data to pack for computation to overlap communication depends on the inherent performance of the processors and network in the system. Relative to processor speed, slower networks with a higher setup penalty will require larger amounts of packing while faster networks will require less.

There are several side effects with the use of multithreading and data packing in the DP-MVDR algorithm. First, multithreading requires thread synchronization, which is overhead not present in a single-threaded solution. Also, the more data that is packed the longer the result latency. Therefore, the number of data samples to pack should be sufficient to overlap the communication completely while maintaining the least possible result latency. In an n-node system, where d samples per packet are transmitted per communication cycle, the result latency with data packing is increased, from $2n$ pipeline stages to $2n + d$ stages, since the algorithm allots d stages for communication.

An increase in memory requirements is another side effect of data packing and multithreading. Fig. 10 shows the data memory requirements for the sequential algorithm and the parallel algorithm versus payload

size of the packet and number of nodes in the system. As stated earlier, each sample is a complex scalar that consists of two, 8-byte floating-point values. The sequential algorithm needs to store only one copy of the data from each node and thus requires the least amount of memory, which is $16n$ bytes. In the parallel algorithm, even without data packing the double-buffering associated with multithreading raises the memory requirement to twice that of the sequential algorithm – one set that is being communicated, and another set that is being processed. In general, for an n-node system where each sample is 16 bytes, the memory required per node to store d samples per packet from every node is $2d \times 16n$ bytes. Consider a single front-end processor executing the sequential algorithm for an 8-node array. It requires $16n = 128$ bytes to store the vector of data samples. Each node in an 8-node array executing DP-MVDR and sending only one sample per packet requires 256 bytes. For n data samples per packet each node requires 2kB, and for n^2 samples per packet each node requires 16kB.

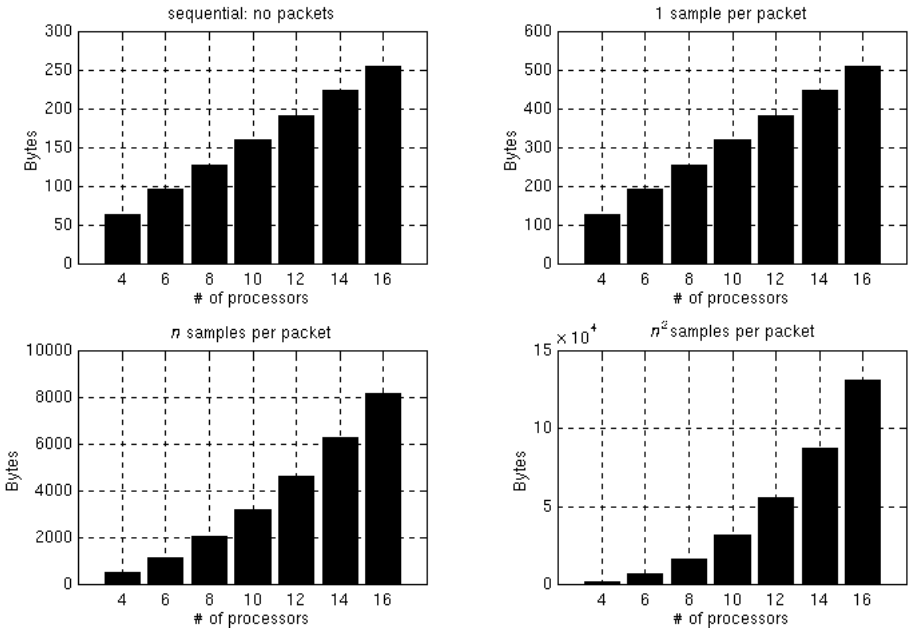


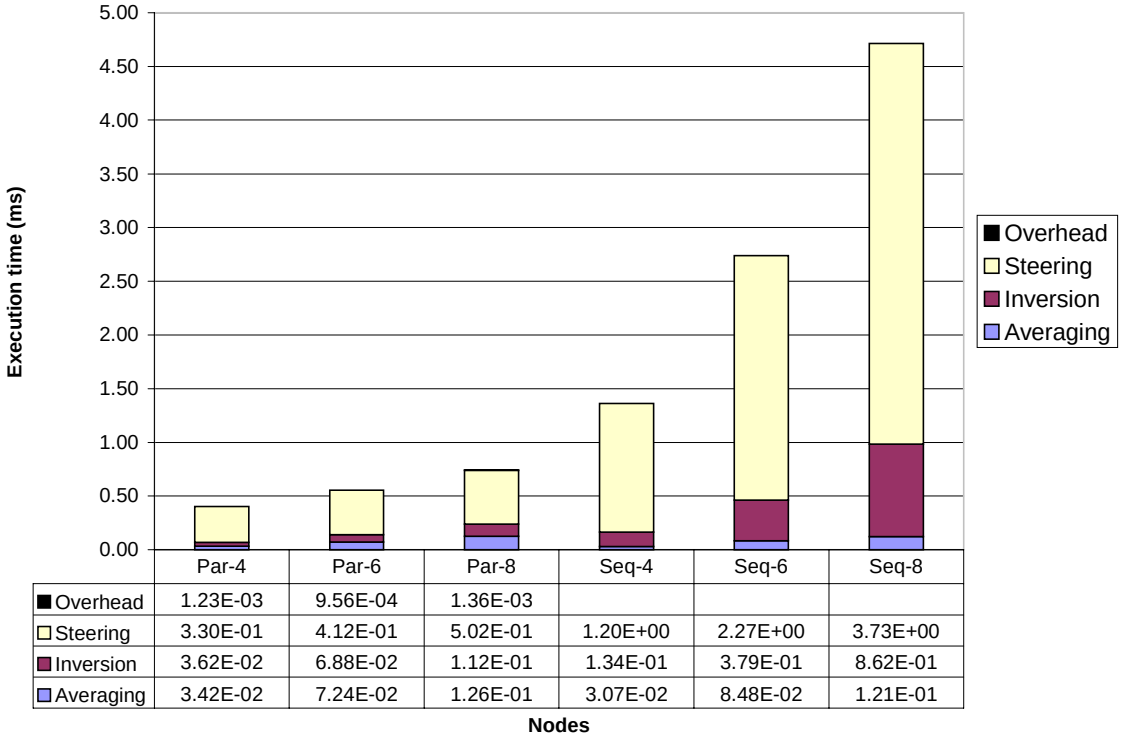
Fig. 10. Data memory requirements for communication as a function of the number of processors for various payload sizes.

The need for data packing and multithreading in the DP-MVDR algorithm is most critical when the nodes in the array are connected by a unicast point-to-point communications network that is slow relative to processor speed. If the network is capable of inherently supporting hardware broadcast (e.g. a ring), where a message destined for all other nodes in the system is transmitted only once and yet received multiple times, then the communication complexity of the all-to-all communication reduces from $O(n^2)$ to $O(n)$. This reduction in communication complexity may permit the use of smaller degrees of data packing while still achieving the overlap in computation over communication, and in so doing reduce the memory requirements and result latency. Moreover, a broadcast network that is fast relative to processor speed may even mitigate the need for multithreading. Thus, the communication component of the DP-MVDR algorithm may be tuned in its use depending on the processor and network speed and functionality.

The next section presents performance results with the DP-MVDR algorithm for several array sizes. Since communication latency is completely hidden by packing data samples, the execution times measured are a function of the computation thread alone.

5. Parallel MVDR Performance

To ascertain overall performance attributes and relative performance of inherent tasks using a distributed systems testbed, the algorithm was implemented via several message-passing parallel programs (i.e. one per array configuration) coded in C-MPI and executed on a cluster of UltraSPARC workstations with an ATM communications network. Each node measures the execution times for its tasks independently and final measurements were calculated by averaging the individual measurements across iterations and nodes. The degree of data packing employed is $d = n^2$ data samples per packet. Fig. 11 shows the execution times for both the parallel and the sequential algorithms for 45 steering directions. The number of nodes is varied to study the effects of problem and system size.



	Par-4	Par-6	Par-8	Seq-4	Seq-6	Seq-8
Overhead	1.23E-03	9.56E-04	1.36E-03			
Steering	3.30E-01	4.12E-01	5.02E-01	1.20E+00	2.27E+00	3.73E+00
Inversion	3.62E-02	6.88E-02	1.12E-01	1.34E-01	3.79E-01	8.62E-01
Averaging	3.42E-02	7.24E-02	1.26E-01	3.07E-02	8.48E-02	1.21E-01

Fig. 11. Average execution time per iteration as a function of array size for the parallel and sequential MVDR algorithms with 45 steering angles and 576 iterations on the Ultra-1/ATM cluster.

The parallel times show that the *steering* task remains the most computationally intensive task for both the parallel and sequential programs, since in all cases the number of nodes is less than the number of steering directions. As the *matrix inversion* task in the parallel algorithm has the same parallel complexity as the *averaging* task, $O(n^2)$, their execution times are approximately equal. The sequential algorithm does not include an *overhead* component because it is not pipelined nor multithreaded, and communication latencies are ignored. However, the *overhead* time of the parallel algorithm is negligible and therefore does not significantly affect the total execution time. The amount of time to perform the *averaging* task is comparable for both algorithms since it is not affected by the pipelining in DP-MVDR.

The two decomposed tasks in DP-MVDR, *matrix inversion* and *steering*, show significant improvement over their sequential counterparts. Fig. 12 is a comparison of DP-MVDR and the sequential algorithm in terms of scaled speedup (i.e. speedup where the problem size grows linearly with the number

of nodes in the system) and parallel efficiency for each of these two tasks. Parallel efficiency is defined as the ratio of speedup versus the number of processing nodes employed in the system.

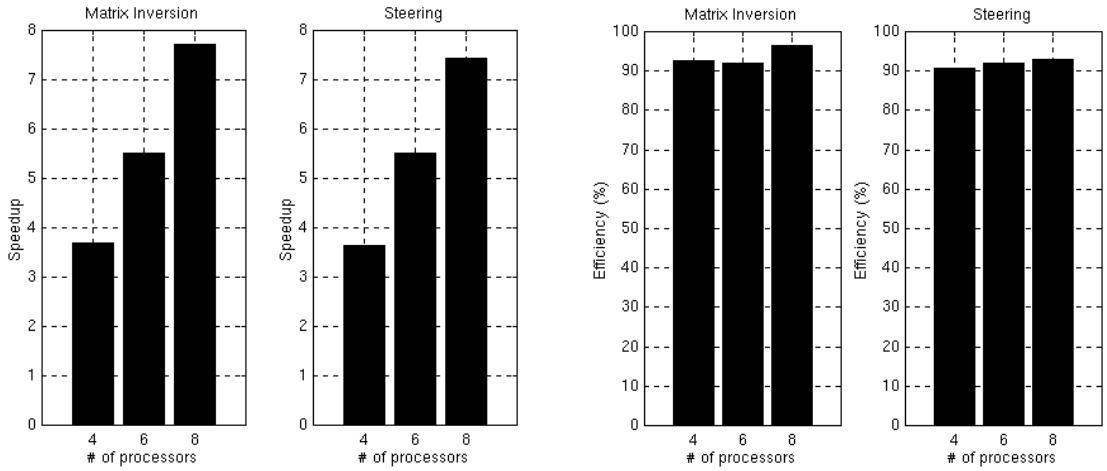


Fig. 12. Scaled speedup per critical task (a) and parallel efficiency per critical task (b) as a function of the number of processors.

The *averaging* task is not included in the figure since it is performed identically in both the sequential and parallel algorithms and thus exhibits no speedup. The *matrix inversion* and *steering* tasks achieve near-linear speedup with average parallel efficiencies of 94% and 92%, respectively. The parallel efficiency of the *inversion* task does not demonstrate a clear monotonic increasing or decreasing trend. Rather, it shows a slight variation about an average efficiency of 94%. This characteristic is due to the pivoting operation, which involves data-dependent interchanges of elements. Since these two tasks were each decomposed into n stages, the ideal case would of course be a speedup of n and efficiency of 100%. However, much as with any pipeline, the ideal is not realized due to the overhead incurred from multithreading and managing the multistage computations within the pipeline. Unlike the *inversion* task, the *steering* task shows a slightly increasing trend in its parallel efficiency, due to the adverse effect of pipelining and multithreading overheads declining with respect to the increasing problem size. This characteristic is a clear demonstration of Amdahl's Law, as the steering efficiency grows with an increase in the parallelizable fraction of the algorithm, while the overhead remains constant.

The previous figures show the results for each individual beamforming task. When comparing the overall system performance of the parallel algorithm with the sequential algorithm, we are primarily concerned with the effective execution time of each algorithm. The effective execution time in the parallel algorithm is the amount of time between outputs from successive iterations once the pipeline has filled (i.e. one pipeline stage). In the sequential algorithm, the effective execution time is the same as the result latency (i.e. the length of one beamforming iteration). For the parallel program, time-stamp functions are called at the beginning and end of each pipeline stage to obtain the effective execution time. Each node performs its own independent measurements, which are then averaged across iterations and nodes.

Fig. 13 shows the overall system performance in terms of scaled speedup and parallel efficiency. Despite the long result latency of DP-MVDR, the effective execution time is low, resulting in scaled speedups of approximately 3.4, 4.9, and 6.2 for 4, 6, and 8 nodes, respectively. The parallel efficiencies are approximately 85%, 82%, and 78%, respectively, with an average efficiency of approximately 82%. This moderate deterioration of efficiency that occurs with increase in the number of nodes is a result of the rapidly increasing time taken by the *averaging* task, which is the only task in the sequential algorithm that

is not parallelized. Thus, as the number of nodes is increased, there is a proportionately larger increase in the sequential part of the algorithm compared to the part that has been parallelized, causing an overall decrease in parallel efficiency. Of course, given the limited number of steering angles and frequency bins involved, these results can be considered as a lower bound on parallel performance, since an increase in the number of angles and/or bins employed in an array would naturally place far more pressure on those tasks with the highest computational complexity (i.e. the *matrix inversion* and *steering* tasks). Thus, parallel efficiencies for systems that use more angles and bins can be expected to attain even higher levels of parallel efficiency.

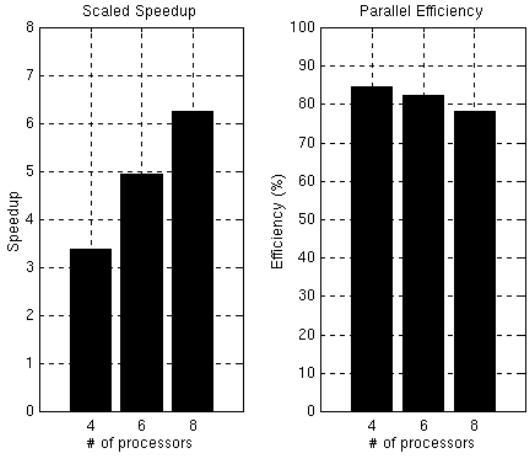


Fig. 13. Overall scaled speedup and parallel efficiency as a function of the number of processors.

6. Conclusions

This paper introduces a novel set of techniques for the parallelization of adaptive beamforming on distributed systems for in-array sonar signal processing, such as a sonar array whose nodes are comprised of DSP processors with local memory and are interconnected by a point-to-point communications network. The techniques employed focus upon a challenging baseline consisting of a narrowband, unconstrained MVDR beamformer, where a high degree of parallel efficiency is obtained through several forms of pipelined processing coupled with techniques to both reduce and hide communication overhead. The results also provide a basis for parallel in-array processing for other forms of MVDR beamforming algorithms involving a number of frequency bins.

The parallel algorithm decomposes the most complex tasks of sequential MVDR in a coarse-grained fashion using both a computation component and a communication component. In the computation component, successive iterations of beamforming are distributed across nodes in the system with round-robin scheduling and executed in an overlapping, concurrent manner via pipelining. Within each pipeline, the matrix inversion and steering tasks are themselves pipelined and overlapped in execution across and within the nodes. The computational complexity of the parallel algorithm is thus reduced to $O(n^2)$ for a fixed number of steering directions, where n is the number of nodes and sensors in the system.

In the communication component, provisions are made to support point-to-point networks of moderate performance with the ability to only perform unicast communications. Data packing techniques are employed to reduce the latency of the all-to-all communication needed with MVDR by amortizing the setup and encapsulation overhead, and multithreading is employed to hide the remaining latency with

concurrency in computations and communications. For a cluster of ATM workstations, the packing of n^2 data samples per packet was empirically found to provide complete overlap of computation versus communication while maintaining the minimum result latency. However, the optimal choice of this parameter will depend upon the particular processors, network, and interfaces present in the distributed architecture of interest.

There are several side effects associated with the use of pipelining, multithreading, and data packing in this (or any) parallel algorithm. Synchronization between stages in the loosely coupled pipeline brings with it increased overhead, as does the multithreading of computation and communication threads. Pipelining increases the result latency, as does multithreading and data packing, and memory requirements increase linearly with increases in the degree of data packing.

The results of performance experiments on a distributed system testbed indicate that the parallel algorithm is able to provide a near-linear level of scaled speedup with parallel efficiencies that average more than 80%. This level of efficiency is particularly promising given that the network used in the experiments (i.e. ATM) does not support hardware broadcast. When hardware broadcast is available in a given network, the communication complexity drops from $O(n^2)$ to $O(n)$, making it likely that even higher efficiencies can be obtained. Moreover, on distributed systems with a fast network relative to processor speed and capable of broadcast, the need for data packing and multithreading may be mitigated thereby minimizing computation and communication overhead.

Although the baseline algorithm employed to study distributed processing techniques for adaptive beamforming is defined in terms of only a single frequency bin of interest, the techniques are readily extensible for multiple frequency bins since beamforming cycles are performed independently for each bin. Thus, for each stage of each task in each pipelined processor in the system, the computations would be repeated for each of the b bins of interest. For example, in terms of the computation component of the DP-MVDR algorithm, the processor assigned to compute the output for beamforming iteration i would proceed by first performing the task of CSDM averaging for each of the b bins in a single stage. Next, in that same stage plus n–1 additional stages, the processor would perform the task of matrix inversion for each of the b matrices that result from the averaging. Finally, via n more stages, the processor would perform the task of steering using each of the b inverted matrices that result from the inversion task. The narrowband beamformer used as a baseline can be considered to be a worst-case scenario for MVDR processing, and thus the results obtained represent a lower bound with respect to parallel performance. Such is the case because a greater number of frequency bins being processed leads to an increase in potential concurrency without an increase in data dependencies. This change implies that a relatively larger portion of the sequential algorithm can be parallelized with a lower degree of relative overhead, thereby leading to the potential for improvements in speedup and parallel efficiency. The effect of this trend is a subject for future study.

Several directions for future research are anticipated. Building on the techniques presented in this paper, enhanced forms of the baseline MVDR beamformer with broadband processing, robustness constraints, and iterative updates of the inverse CSDM matrix will be parallelized and analyzed. An implementation of the resulting parallel algorithm for practical adaptive beamforming on an embedded, distributed system comprised of low-power DSP devices is anticipated. Moreover, to complement the performance-oriented emphasis in this paper, further work is needed to ascertain strengths and weaknesses of sequential and parallel MVDR algorithms in terms of fault tolerance and system reliability. By taking advantage of the distributed nature of these parallel techniques for in-array sonar processing, the potential exists for avoiding single points of failure and overcoming the loss of one or more nodes or links with graceful degradation in performance. Finally, techniques described in this paper can be applied to more advanced beamforming algorithms such as MUSIC and matched-field processing, and future work will focus on the study of opportunities for parallelization in these algorithms.

Acknowledgements

The support provided by the Office of Naval Research on grant N00014-99-1-0278 is acknowledged and appreciated. Special thanks go to Thomas Phipps from the Applied Research Lab at the University of Texas at Austin, and the anonymous reviewers at the JCA, for their many useful suggestions.

References

1. A. O. Steinhardt and B. D. Van Veen, "Adaptive Beamforming," *International J. of Adaptive Control and Signal Processing* **3**(3) (1989), 253-281.
2. L. Castedo and A. R. Figueiras-Vidal, "An Adaptive Beamforming Technique Based on Cyclostationary Signal Properties," *IEEE Trans. on Signal Processing* **43**(7) (1995), 1637-1650.
3. M. Zhang and M. H. Er, "Robust Adaptive Beamforming for Broadband Arrays," *Circuits, Systems, and Signal Processing* **16**(2) (1997), 207-216.
4. J. Krolik and D. Swingler, "Multiple Broad-Band Source Location using Steered Covariance Matrices," *IEEE Trans. on Acoustics, Speech, and Signal Processing* **37**(10) (1989), 1481-1494.
5. H. Cox, R. M. Zeskind, and M. M. Owen, "Robust Adaptive Beamforming," *IEEE Trans. On Acoustics, Speech, and Signal Processing* **35**(10) (1987), 1365-1377.
6. A. D. George, J. Markwell, and R. Fogarty, "Real-time Sonar Beamforming on High-performance Distributed Computers," *Parallel Computing*, to appear.
7. A. D. George and K. Kim, "Parallel Algorithms for Split-Aperture Conventional Beamforming," *J. Computational Acoustics* **7**(4) (1999), 225-244.
8. S. Banerjee and P. M. Chau, "Implementation of Adaptive Beamforming Algorithms on Parallel Processing DSP Networks," Proc. SPIE, Vol. 1770 (1992), 86-97.
9. M. Moonen, "Systolic MVDR Beamforming with Inverse Updating," *IEE Proc. Part-F, Radar and Signal Processing* **140**(3) (1993), 175-178.
10. J. G. McWhirter and T. J. Shepherd, "A Systolic Array for Linearly Constrained Least Squares Problems," Proc. SPIE, Vol. 696 (1986), 80-87.
11. F. Vanpoucke and M. Moonen, "Systolic Robust Adaptive Beamforming with an Adjustable Constraint," *IEEE Trans. On Aerospace and Electronic Systems* **31**(2) (1995), 658-669.
12. C. E. T. Tang, K. J. R. Liu, and S. A. Tretter, "Optimal Weight Extraction for Adaptive Beamforming Using Systolic Arrays," *IEEE Trans. On Aerospace and Electronic Systems* **30**(2) (1994), 367-384.
13. S. Chang and C. Huang, "An Application of Systolic Spatial Processing Techniques in Adaptive Beamforming," *J. Acoust. Soc. Am.* **97**(2) (1995), 1113-1118.
14. M. Y. Chern and T. Murata, "A Fast Algorithm for Concurrent LU Decomposition and Matrix Inversion," Proc. International Conf. on Parallel Processing (1983), 79-86.
15. D. H. Bailey and H. R. P. Ferguson, "A Strassen-Newton Algorithm for High-Speed Parallelizable Matrix Inversion," Proc. Supercomputing (1988), 419-424.
16. D. S. Wise, "Parallel Decomposition of Matrix Inversion using Quadtrees," Proc. International Conf. on Parallel Processing (1986), 92-99.
17. V. Pan and J. Reif, "Fast and Efficient Parallel Solution of Dense Linear Systems," *Computers Math. Applic*. **17**(11) (1989), 1481-1491.
18. B. Codenotti and M. Leoncini, "Parallelism and Fast Solution of Linear Systems," *Computers Math. Applic*. **19**(10) (1990), 1-18.

19. K. K. Lau, M. J. Kumar, and S. Venkatesh, “Parallel Matrix Inversion Techniques,” Proc. IEEE International Conf. on Algorithms and Architectures for Parallel Processing (1996), 515-521.
20. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*, 2nd ed., Cambridge University Press, 1992.
21. P. M. Clarkson, *Optimal and Adaptive Signal Processing*, CRC Press, 1993.
22. G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed., Johns Hopkins University Press, 1996.
23. Message Passing Interface Forum, “MPI: A Message-Passing Interface Standard,” Technical Report CS-94-230, Computer Science Dept., Univ. of Tennessee, April 1994.