A PARALLEL ALGORITHM FOR DISTRIBUTED MINIMUM VARIANCE DISTORTIONLESS RESPONSE BEAMFORMING

By

JESUS LUIS GARCIA

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

1999

For my family

ACKNOWLEDGMENTS

I would like to thank Dr. Alan George for allowing me to pursue my research interests in the HCS Lab and for encouraging me and believing in me during unfortunate circumstances. I especially want to thank Ken Kim without whose guidance and advice this work would not be possible.

I owe much to my family in Z-Systems, which has helped me grow intellectually and musically. I specifically want to thank Glenn Zelniker for showing me the wonders of DSP, Rainier De Varona for giving me the opportunity to hear him vent, and Dale Ramcunis for enriching my palate. I must also thank Z-Systems for those extra lines on my resume.

My fellow lab members deserve many thanks for hours of wasted time and entertainment. I thank the Trilogy for many dinners and classic rock quizzes, the CGs for a great team effort, the Beefcakes for letting me train their muscles and their minds, and the ITMSBs for letting me teach them how to speak proper English. I especially want to thank BBB, Mr. Bigglesworth, Gomez Christopher, and Jefferson for allowing me to call them by any ridiculous name I could think of, and many other things.

Last but definitely not least, I want to thank my family for their infinite love and support. I would also like to thank my many friends for years of good memories and unremembered nights.

TABLE OF CONTENTS

page

ACKNOWLEDGMENTS iii

ABSTRACT vi

CHAPTERS

1 INTRODUCTION 7

2 BACKGROUND 11

2.1 MPI 12
2.2 Threads 14
2.3 Performance measurements 16

3 OVERVIEW OF MVDR ALGORITHM 19

4 SEQUENTIAL MVDR ALGORITHM AND PERFORMANCE 25

4.1 Sequential MVDR tasks 25
4.2 Performance results for sequential MVDR algorithm 27

5 PARALLEL MVDR ALGORITHM 31

5.1 Computation component of DP-MVDR 33
5.2 Communication component of DP-MVDR 37

6 PARALLEL MVDR PERFORMANCE 45

7 CONCLUSIONS 50

APPENDICES

A DATA MODEL 54

B VERIFICATION OF RESULTS 56

C PARALLEL AND SEQUENTIAL C++ CODE 58

D MATLAB SOURCE CODE 94

REFERENCES 99

BIOGRAPHICAL SKETCH 101

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

A PARALLEL ALGORITHM FOR DISTRIBUTED MINIMUM VARIANCE DISTORTIONLESS RESPONSE BEAMFORMING

By

Jesus Luis Garcia

December 1999

Chairman: Alan D. George
Major Department: Electrical and Computer Engineering

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 thesis, a new parallel algorithm for Minimum Variance Distortionless Response (MVDR) adaptive beamforming on distributed systems is introduced for in-array sonar signal processing. 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 high-speed network.

CHAPTER 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 ratio (SNR). An array can be configured arbitrarily in three dimensions, but this thesis 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 the cross-spectral density matrix (CSDM) to extract the desired signal without distortion and a minimum of variance at the output. Numerous ABF algorithms have been proposed in the literature[1-4], but this thesis focuses on the 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.

This thesis introduces a novel parallel algorithm for MVDR beamforming in a distributed fashion that scales with problem and system size on a linear array. This new algorithm is designed for in-array processing where the nodes are hydrophones coupled with low-power DSP microprocessors 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 on 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 parallel algorithm MVDR for adaptive beamforming presented in this thesis 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.

Background information necessary to understand some of the implementation details of the parallel algorithm is given in Chapter 2. A theoretical overview of MVDR is given in Chapter 3. In Chapter 4, the sequential algorithm is examined in terms of its individual tasks and their performance. In Chapter 5, a new distributed-parallel MVDR

algorithm is presented. A performance analysis comparing the parallel algorithm and the sequential algorithm in terms of execution time, speedup and efficiency is given in Chapter 6. Finally, Chapter 7 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.

CHAPTER 2
BACKGROUND

The parallel programs employed in this thesis use a paradigm for communication among distributed-memory multicomputers known as message passing. Message passing allows multiple processes in a system to communicate through messages. The Message Passing Interface (MPI) is a standard developed by the Message Passing Interface Forum[20] that defines the syntax and semantics of a core of library functions, for communication in a distributed-memory multicomputer, that is highly portable and easy to use. The MPI standard is implementation independent, meaning that it can be implemented differently depending on the target system. Thus, there are several proprietary as well as public domain implementations of MPI.

For this research, a public-domain implementation called MPICH was used. Since MPICH is freely distributed and is meant for high portability, it makes no assumptions about the underlying network and therefore does not take advantage of features such as physical broadcast capabilities. Instead, MPICH uses point-to-point communications running over a robust TCP/IP protocol stack to simulate broadcast communications. Communicating over TCP/IP increases the communication overhead substantially, especially for short messages. In the parallel MVDR algorithm presented in this thesis, one technique used to hide the communication latency is to use a separate thread to handle communications. Even though MPI does not offer explicit support for

threads it is, nevertheless, thread-safe. Thread-safe refers to the fact that only the calling thread is affected by calls to MPI functions.

The parallel programs in this thesis use threads and MPI to achieve communication and synchronization among the nodes. The following two sections in this chapter will explain the basics of programming with MPI and threads in order to provide a clearer understanding of the mechanisms used to hide the communication latency. Section 2.3 will give an explanation of the performance measurements used to evaluate the performance of the parallel algorithm.

2.1 MPI

Included in the MPI standard is a set of point-to-point and collective communications operations. MPI provides 125 functions but most parallel programs require only six basic functions, which are:

1. *MPI_Init* – initialization,
2. *MPI_Comm_rank* – get process ID,
3. *MPI_Comm_size* – get total number of processes,
4. *MPI_Send* – send message,
5. *MPI_Recv* – receive message,
6. *MPI_Finalize* – clean up all MPI states and exit MPI.

These are the six MPI functions used in the code for the parallel MVDR algorithm. The prototypes of the *MPI_Send* and *MPI_Recv* functions are as follows:

- *int MPI_Send(&message, count, datatype, dest, tag, communicator)*
- *int MPI_Recv(&message, count, datatype, source, tag, communicator, &status).*

The *message* parameter is the location of the data that is to be sent or received. *Count* specifies how many data elements of type *datatype* are to be sent or received. MPI defines a set of data types that are analogous to the C data types (i.e., MPI_INT, MPI_DOUBLE, MPI_BYTE, etc.) MPI defines the concept of a *communicator*, which

is the collection of processes that can communicate with one another. Most MPI function calls must specify the communicator. The *rank* of the process is a unique integer assigned to each process in a communicator to distinguish between them. It is this *rank* that serves as the source and destination in point-to-point functions. The *tag* parameter serves to distinguish messages from a single process. The *status* parameter in the *MPI_Recv* function specifies the tag and source of the received message.

MPI also provides a set of collective communication primitives. There are three types of collective operations: (1) Synchronization, (2) Data movement, and (3) Collective computation. The synchronization primitive is *MPI_Barrier(comm)*, which blocks until all processes in the communicator *comm* call it. Data movement functions include *MPI_Bcast, MPI_Gather, MPI_Allgather, MPI_Scatter, MPI_Alltoall*, and many others. The collective computation functions (e.g., *MPI_Reduce, MPI_Scan*, etc.) allow computations to be performed on a set of collective data and distribute the results in a specified fashion among the nodes.

The parallel code uses multiple point-to-point communication primitives, *MPI_Send* and *MPI_Recv*, to perform all-to-all communication. It was found empirically that using multiple *MPI_Send* and *MPI_Recv* function calls is actually faster than using MPI collective communications primitives such as *MPI_Bcast* and *MPI_Allgather*. However, results may differ for a different testbed and MPI implementation. The high-level pseudo-code for how *MPI_Send* and *MPI_Recv* were used in the parallel code to perform all-to-all communication is given in Figure 2.1.

```
MPI_Barrier(MPI_COMM_WORLD);            //synchronize nodes
/* Begin communication */
FOR i = 0 TO number of nodes            //send to all nodes
  IF (i!=my_rank)                       //do not send to yourself
    MPI_Send(&my_data, num_of_data_elements, MPI_DOUBLE, i, tag(i),
             MPI_COMM_WORLD);
  ENDIF
ENDFOR
FOR j = 0 TO number of nodes            //receive from all nodes
  IF (j!=my_rank)                       //do not receive from yourself
    MPI_Recv(&data_buffer, num_of_data_elements, MPI_DOUBLE, j, tag(j),
             MPI_COMM_WORLD, &status);
  ENDIF
ENDFOR
```

Figure 2.1. Pseudo-code of all-to-all communication using point-to-point primitives.

2.2 Threads

A thread of execution is the stream of instructions in a program executed by the CPU. Each thread of execution is associated with a *thread*, an abstract data type representing flow of control within a process. A thread has its own execution stack, program counter value, register set, and state. Threads give the illusion of concurrent execution by context switching in which the processor dedicates a certain amount of time to the execution of one thread then switches context and executes another thread. The thread package used to code the parallel MVDR algorithm is the POSIX.1c[21] standard. This standard defines function calls for thread management, mutual exclusion, and condition variables.

In the parallel code, six different functions are used for thread handling. The six functions handle thread creation and destruction, and thread synchronization. The six functions are the following:

1. *pthread_create* – creates a new thread to execute a specified function,
2. *pthread_join* – blocks the calling function until the specified thread exits,
3. *pthread_mutex_lock* – acquire a lock to protect a critical section,
4. *pthread_mutex_unlock* – release the lock,
5. *pthread_cond_wait* – blocks the calling thread until a signal is received,
6. *pthread_cond_signal* – sends a signal to awaken a blocked thread.

The remainder of this section will describe how thread synchronization is achieved by the use of functions 3, 4, 5, and 6.

Multiple threads within a process must share resources. Thread synchronization is used to ensure that threads get consistent results when they share resources. The POSIX.1c standard provides mutexes (i.e., mutual exclusion variables) and condition variables to support the two types of synchronization – locking and waiting. A thread *locks* a resource when it has exclusive access to it. Locking is typically of short duration. A thread is *waiting* when it blocks until the occurrence of some event.

The parallel code uses a combination of mutexes and condition variables to achieve synchronization among the threads. Mutexes are used to protect critical sections and obtain exclusive access to resources. A thread calls the function *pthread_mutex_lock(&my_lock)* to acquire a lock at the beginning of the critical section. The same thread calls *pthread_mutex_unlock(&my_lock)* to release the lock at the end of the critical section. Condition variables allow a thread to block until a certain event or combination of events occurs. A thread tests a predicate (i.e., a condition) and calls *pthread_cond_wait* if the predicate is false. When another thread changes variables that might make the predicate true, it awakens the waiting thread by executing p*thread_cond_signal*. To ensure that the test of the predicate and the wait are atomic, the calling thread must obtain a mutex before it tests the predicate. If the thread blocks on a condition variable, *pthread_cond_wait* atomically releases the mutex and blocks. After

returning from *pthread_cond_wait* the predicate must be tested again since the return may have been caused by *pthread_cond_signal* that did not signify that the predicate had become true.

The parallel MVDR algorithm was coded using one thread for communication and another thread for computation. The threads must synchronize at the beginning and end of each communication cycle. The thread that finishes its cycle first calls *pthread_cond_wait* and blocks until the other thread calls *pthread_cond_signal* to awaken the blocking thread. Figure 2.2 gives the high-level pseudo code for the thread synchronization sections of the parallel code.

2.3 Performance measurements

The performance of the parallel MVDR algorithm was gauged by measuring its execution time, scaled speedup, and parallel efficiency and compared to the sequential algorithm.

Execution time is a measure of the amount of time for an application to complete execution. The execution time was measured by calling a high-resolution timing function in UNIX (i.e., *gethrtime*) that the returns the time expressed in nanoseconds since some arbitrary time in the past. The function was called at the beginning of the section of interest and the return value saved. It was called again at the end of the section and the two times subtracted to get the total execution time of the section of interest.

Speedup is the ratio of sequential execution time to parallel execution time. Scaled speedup is attained when the problem size scales with the number of processors.

```
pthread_mutex_lock(&start_lock);         //acquire mutex start_lock
WHILE (!startnow)                       //wait for startnow to be true
  pthread_cond_wait(&start_cond, &start_lock);
startnow = 0;                            //set startnow back to false
pthread_mutex_unlock(&start_lock);       //release mutex start_lock
/* Communication section (see Figure 2.1)*/
/* Tell computation thread you are done communicating*/
pthread_mutex_lock(&my_lock);            //acquire mutex my_lock
done=1;                                  //set done flag to true
pthread_cond_signal(&cond);              //signal computation thread
pthread_mutex_unlock(&my_lock);          //release mutex my_lock
```

(a)

```
// Wait for communication thread to tell you it is done communicating
pthread_mutex_lock(&my_lock);            //acquire mutex my_lock
WHILE (!(done))                          //wait for done to be true
  pthread_cond_wait(&cond, &my_lock);
done = 0;                                //set done flag to false
pthread_mutex_unlock(&my_lock);          //release mutex my_lock
/* Tell communication thread to begin new communication cycle */
pthread_mutex_lock(&start_lock);         //acquire mutex start_lock
startnow=1;                              //set startnow flag to true
pthread_cond_signal(&start_cond);        //signal communication thread
pthread_mutex_unlock(&start_lock);       //release mutex start_lock
```

(b)

Figure 2.2. Pseudo-code for thread synchronization.
(a) Code executed by communication thread; (b) Code executed by computation thread.

In the parallel MVDR algorithm, the number of processors is tied to the problem size (i.e., the number of sensors in the sonar array) so the performance results in Chapter 6 are in terms of scaled speedup. The ideal speedup, or the degree of parallelism (DOP), is the number of processes executing in parallel. If speedup is less than 1, then the

sequential program executes faster than the parallel program and the parallel program is futile.

Parallel efficiency is the ratio of the speedup attained versus the ideal speedup and is expressed as a percentage. The ideal parallel efficiency is 100%.

CHAPTER 3
OVERVIEW OF MVDR ALGORITHM

The objective of beamforming is to resolve the direction of arrival of spatially separated signals within the same frequency band. 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 $k = w/c$, where w 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.*[8] 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 and we only process the data in a single frequency bin since, as will become clear later, the parallel algorithm presented in this thesis is easily extensible to multiple frequency bins.

Figure 3.1 shows the structure of a basic adaptive beamformer with a signal arriving from angle q. There are n sensors with frequency-domain outputs $x_o, \ldots, x_{n-1}$. 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

$$y = \boldsymbol{w}^{*}\boldsymbol{x} \tag{3.1}$$

where the * denotes complex-conjugate transposition.

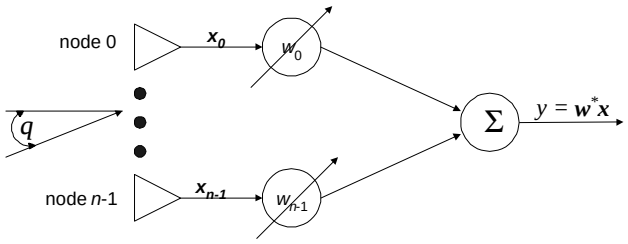


Figure 3.1. Narrowband ABF structure.

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

$$\boldsymbol{s} = [1, \quad e^{-jkd\sin(q)}, \quad e^{-j2kd\sin(q)}, \cdots, \quad e^{-j(n-1)kd\sin(q)}] \tag{3.2}$$

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[22].

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

$$R = E\{\boldsymbol{x}\cdot\boldsymbol{x}^{*}\}. \tag{3.3}$$

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:

$$P = E\{|y|^2\} = \boldsymbol{w}^* E\{\boldsymbol{x}\cdot\boldsymbol{x}^*\}\boldsymbol{w} = \boldsymbol{w}^* R\, \boldsymbol{w}. \tag{3.4}$$

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

$$\boldsymbol{w}^* \boldsymbol{s} = g \tag{3.5}$$

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. (3.4) and (3.5) and assuming that $\boldsymbol{s}$ is normalized (i.e., $\boldsymbol{s}^*\boldsymbol{s} = 1$), we solve for the weights from

$$\underset{w}{Min}\, P = \boldsymbol{w}^*\, R\, \boldsymbol{w} \text{ constrained to } \boldsymbol{w}^* \boldsymbol{s} = 1. \tag{3.6}$$

Using the method of Lagrange multipliers, it is found that the solution for the weight vector in Eq. (3.6) is

$$\boldsymbol{w} = \frac{R^{-1}\boldsymbol{s}}{\boldsymbol{s}^* R^{-1}\boldsymbol{s}}. \tag{3.7}$$

By substituting Eq. (3.7) into Eq. (3.6), we obtain the scalar output power for a single steering direction as:

$$P = \frac{1}{\boldsymbol{s}^* R^{-1}\boldsymbol{s}}. \tag{3.8}$$

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, Figure 3.2 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 amplitudes of both the signal and noise are normally distributed with zero mean and unit variance, with an SNR of 25 dB.

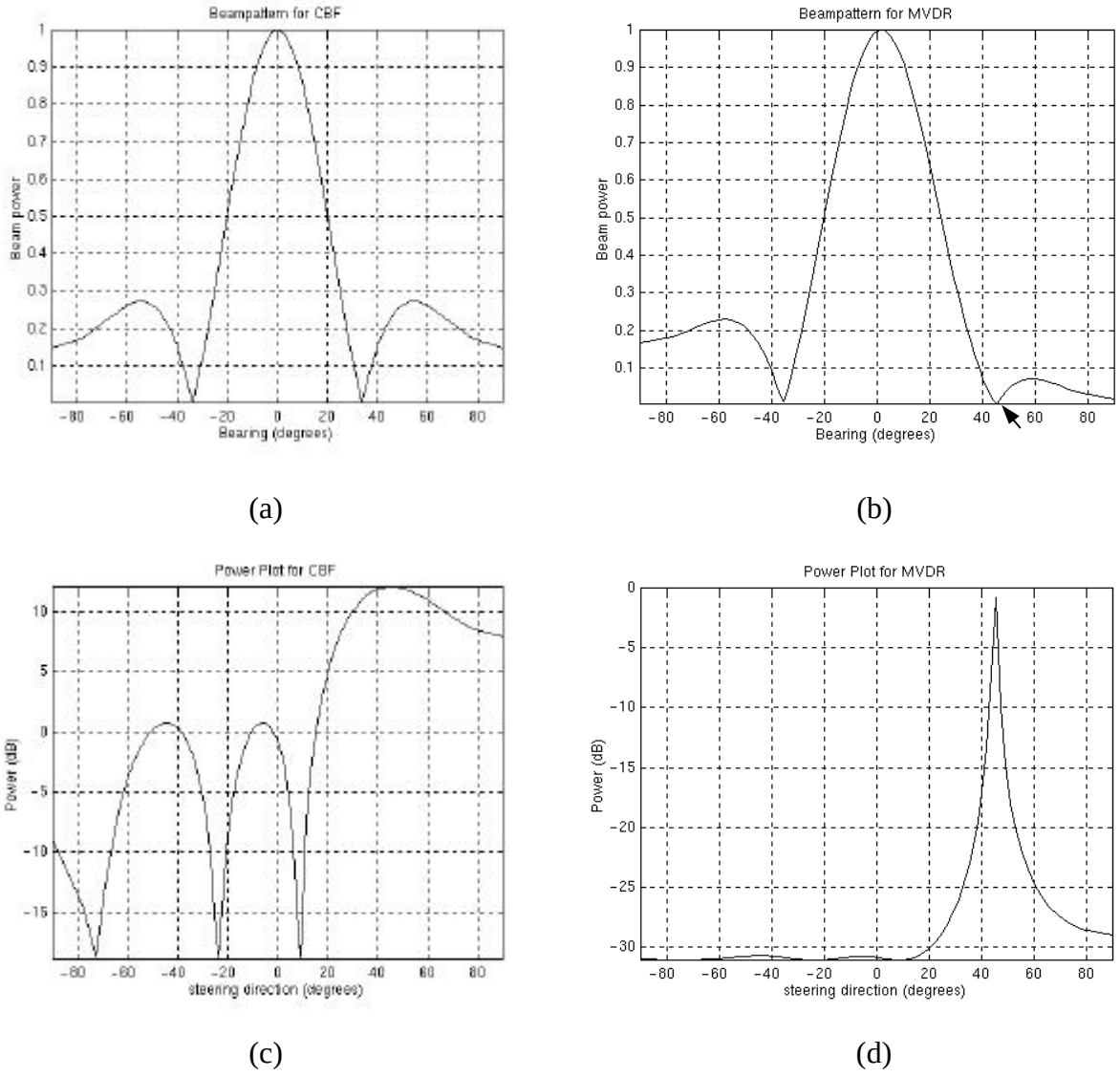


Figure 3.2. Array response and power plots for 4 nodes, 91 steering directions, and incoming signal at q = 45°. (a) CBF plot of beampattern for the array when steered toward broadside; (b) MVDR plot of beampattern for the array when steered toward broadside. Arrow points to null steered at interference arriving at 45°; (c) Resulting log-scale power plot for CBF; (d) Resulting log-scale power plot for MVDR.

Isotropic noise is modeled as a large number of statistically independent stochastic signals arriving at the array from all directions. 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. Figure 3.2(a) shows the beampattern created by a CBF algorithm looking for signals at broadside (i.e., $q = 0°$) with an interferer at $q = 45°$. The result is a beampattern that passes signals at broadside with unity gain while attenuating signals from other angles of incidence. At $q = 45°$, 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 Figure 3.2(b). In this figure, the main lobe is at broadside (i.e., the steering direction), and at $q = 45°$ there is a null that minimizes the energy contributed from the interferer 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[22] 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. Figures 3.2(c) and 3.2(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 Figure 3.2(c) are evidence of the energy contributed by the interferer to other steering directions as this energy was passed, although attenuated, by the CBF beam pattern. By contrast, the MVDR results in Figure 3.2(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.

CHAPTER 4
SEQUENTIAL MVDR ALGORITHM AND PERFORMANCE

Adaptive beamforming with MVDR is divided into three tasks. The first task is the *Ensemble Averaging* of the CSDM. The second task is *Matrix Inversion*, which inverts the ensemble average of the CSDM. The third task is *Steering*, which steers the array and calculates the power for each steering direction. Figure 4.1 is a block diagram of the sequential algorithm, where *n* is the number of sensor nodes in the array.

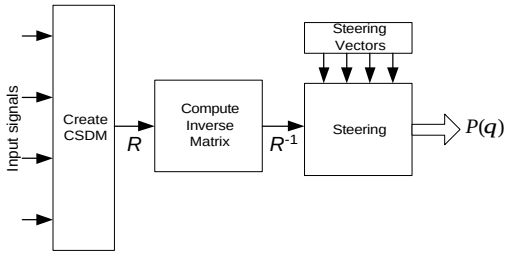


Figure 4.1. Block diagram of the sequential MVDR algorithm.

Section 4.1 presents an overview of each task in the sequential algorithm, followed by performance results given in Section 4.2. The performance results will identify the computational bottlenecks of the sequential MVDR algorithm.

4.1 Sequential MVDR tasks

As defined in Eq. (3.3), the formation of the CSDM requires an expectation operation. The expectation is computed by multiplying the column vector of sensor data,

x, by its complex-conjugate transpose, resulting in an instantaneous estimate of the CSDM. This instantaneous estimate is then added to the ensemble, and the result is divided by the number of estimates accumulated to compute the average. This *ensemble averaging* task consists of n^2 complex multiplications and additions to compute the CSDM and update the ensemble, followed by n^2 divisions to perform the averaging, which results in a computational complexity of $O(n^2)$.

The inversion algorithm we use for the *matrix inversion* task is the Gauss-Jordan elimination algorithm with full pivoting, adapted from Press *et al*.[23] Gauss-Jordan elimination for matrix inversion assumes that we are trying to solve the equation

$$A \cdot X = I \tag{4.1}$$

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.

Gauss-Jordan elimination with full pivoting was chosen because it is numerically stable and efficient in number of computations and memory usage. For details on the algorithm structure and numerical stability of Gauss-Jordan elimination the reader is referred to Golub and Van Loan[24]. The complexity of the algorithm is $O(n^3)$, which is as efficient as any other method for matrix inversion. Gauss-Jordan elimination is particularly attractive for parallel systems because the outer loop that executes n times 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 section will illustrate the impact of each of the tasks in the sequential algorithm.

4.2 Performance results for sequential MVDR algorithm

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 and 128MB of memory. 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 Figure 4.2. 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 *ensemble averaging*, seen at the bottom of each stacked bar, is almost negligible.

The array architecture assumed for 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.

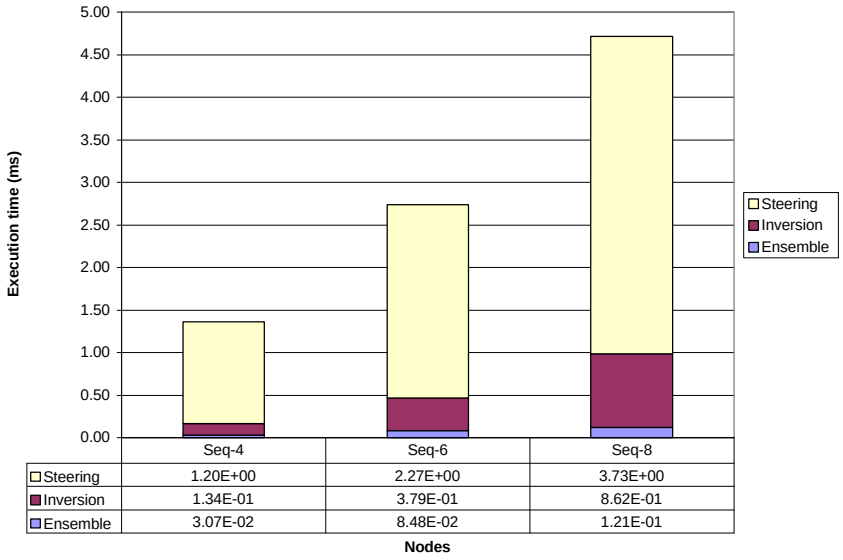


Figure 4.2. Average execution time per iteration as a function of array size for the sequential MVDR algorithm with 45 steering directions and 576 iterations on an Ultra-1 workstation.

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 thesis, 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 chapter describes a 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*. Chapter 6 analyzes the performance of the parallel algorithm and compares it with the purely sequential algorithm presented above.

CHAPTER 5
PARALLEL MVDR ALGORITHM

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 node 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 the 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 distributed-parallel MVDR (DP-MVDR) beamforming algorithm presented here has 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 sections discuss the two algorithmic components in detail, followed by performance results in Chapter 6.

5.1 Computation component of DP-MVDR

As reviewed in Chapter 4, each beamforming iteration of the sequential MVDR algorithm is composed of an *ensemble averaging* task, a *matrix inversion* task, and a *steering* task. Before the task of *matrix inversion* can begin, the ensemble average of the CSDMs 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 beamforming iteration by decomposing them into n stages

each. When a node is scheduled a new beamforming iteration, it computes the ensemble 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 a result latency of 2*n* pipeline stages. Since a node is scheduled a new beamforming iteration every *n* stages, iterations are overlapped within a node by computing a *matrix inversion* stage of the current iteration and the respective *steering* stage of the previously scheduled iteration in the same pipeline stage. Meanwhile, the running sum of CSDMs is updated at the beginning of every pipeline stage. The pipeline structure of the computation component of the DP-MVDR is shown in Figure 5.1 for $n = 3$.

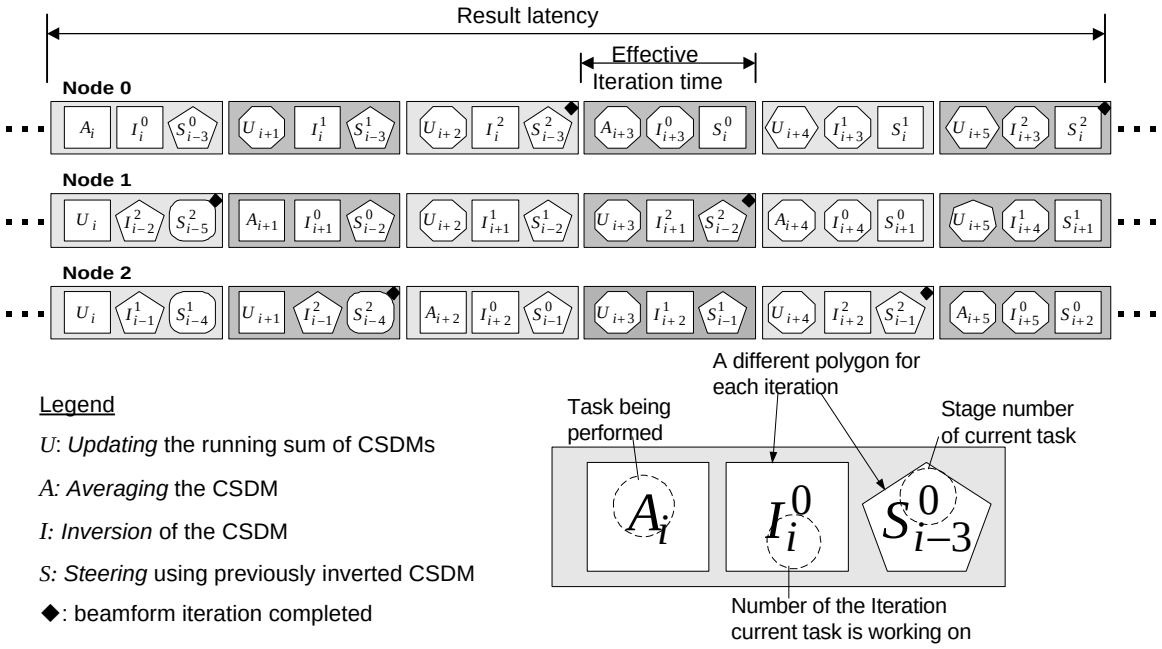


Figure 5.1. Block diagram of the computation component of DP-MVDR (for $n = 3$).

As shown in the legend of the figure, the symbols in the polygons each represent the task being performed in a given stage of the pipeline. The subscripts indicate the

beamforming iteration for which the task is associated, and the superscript (if present) indicates the stage of the task being computed. To further aid in distinguishing between beamforming iterations scheduled across successive nodes, a different polygon is used to depict each iteration of beamforming. For example, the *U* stages have the same polygon as the following *A* task in the same pipe, since the *U* stages update with data the ensemble that will be averaged by the subsequent *A* task. Each pipeline stage is depicted by a shaded box and the completion of a beamforming iteration by a black diamond (i.e., a milestone).

As an example, consider the first four stages of the pipeline shown in the first row of the diagram for node 0. In the first pipeline stage, the ensemble is updated and the average is computed for iteration *i*, and the output is used by the first stage of the matrix inversion. Meanwhile, in that same pipeline stage, the first stage of the steering task is underway based on the results from the matrix inversion previously computed for iteration *i*-3. In the second pipeline stage, new data from all nodes for iteration *i*+1 is used to update the ensemble while the second stage of the inversion for iteration *i* and the second stage of the steering for iteration *i*-3 continue. In the third pipeline stage, the update for iteration *i*+2 takes place, the inversion for iteration *i* is completed, and the steering for iteration *i*-3 is also completed thereby ending iteration *i*-3 by producing the beamforming result. Finally, in the fourth pipeline stage of node 0, the process starts all over with the update and computation of the ensemble average for iteration *i*+3, the first stage of inversion for that same iteration, and the first stage of steering for iteration *i*. Once the pipeline has initialized and filled, the overlapping of iterations between consecutive nodes in the array allows for one beamforming result to be available from the

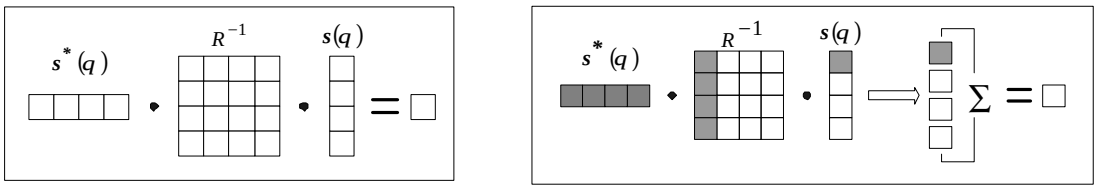
system at the end of every pipeline stage. The decomposition scheme employed for each task is discussed below.

In every pipeline stage a new CSDM is created from the present data set collected from all the nodes in the system and added to the running sum. When a new iteration of beamforming is scheduled to begin on a given node, the running sum is then divided by the total number of sets accumulated thus far and thus ensemble averaging for that iteration is completed. Therefore, the computational complexity of the *ensemble 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.

Figure 5.2 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. (3.8) and q is the steering direction.

Figure 5.2(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 $\boldsymbol{s}(q)$. The results from each stage



(a) (b)

Figure 5.2. Steering task multiplication
(a)Sequential *steering* task ; (b) Decomposed *steering* task .

are accumulated until the multiplication is completed in the n^{th} stage. As evidenced in the performance experiments previously conducted with the sequential algorithm, the *steering* task in the DP-MVDR algorithm, while less computationally complex than matrix inversion as $n \rightarrow \infty$ for a fixed number of steering angles, dominates the execution time when n is less than the number of steering directions.

For further clarification of the DP-MVDR algorithm, high-level pseudo-code is provided in Figure 5.3. The outer loop in the code repeats successive iterations of adaptive beamforming. The inner loop is responsible for coordinating the activities of each of the n processing nodes in the array. The purpose behind the communication-oriented instructions in the first `IF/ENDIF` block of the code is explained in the next section.

5.2 Communication component of DP-MVDR

The expectation operation of Eq. (3.3) at the beginning of every iteration of beamforming requires that each node have a continually 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

```
FOR i = 1 TO number of beamforming iterations
   FOR k = 0 TO number of nodes – 1
      Create R matrix and add to ensemble;
      count++;                        // Count number of stages per comm. cycle
      IF (count == d)                 // Barrier reached
         Wait for communication to finish;
         count = 0;
         Start communicating next d data samples;
      ENDIF
      IF (k == my_rank)               // my_rank is the node number
        CSDM = ensemble_average();
        stage = 0;                   // stage is the current stage of inversion
                                     // and steering
      ENDIF
      Invert(CSDM, stage);                 // CSDM is the ensemble average
      Steer(inverse_csdm(k-n), stage++); // steer using inverse found in
                                           // previous iteration, and increment
                                           // stage
   ENDFOR
ENDFOR
```

Figure 5.3. Pseudo-code for the DP-MVDR algorithm.

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. Figure 5.4 shows a block diagram of this method.

In Figure 5.4, 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). Figure 5.4 illustrates

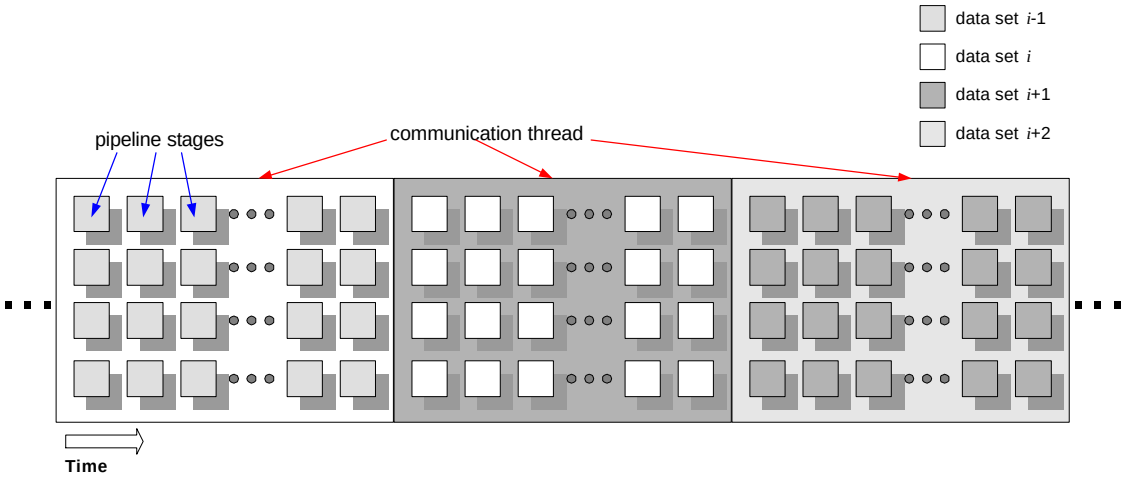


Figure 5.4. Overlap of computation and communication threads.

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 optimal 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 Chapter 4) 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[20]. MPI is a standard that defines a core of functions for message-passing communication and synchronization. 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. Figure 5.5 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.

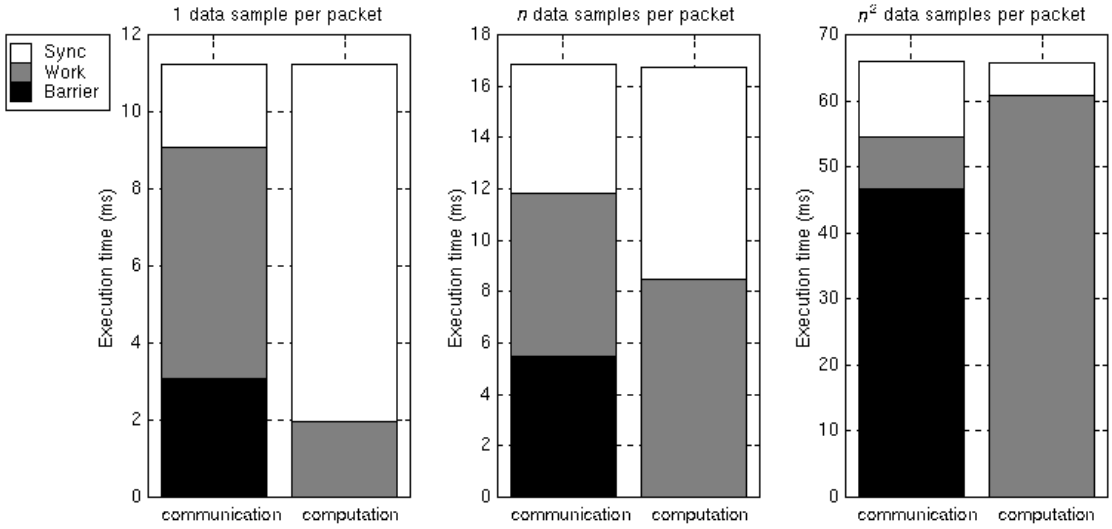


Figure 5.5. Thread performance as a function of data samples per packet (n = 8).

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. Figure 5.5 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 $2n + d$ since the algorithm allots d stages for communication.

An increase in memory requirements is another side effect of data packing and multithreading. Figure 5.6 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 2*d* * 16*n* bytes. Consider a single front-end processor executing the sequential algorithm for an 8-node array. It requires 16*n* = 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.

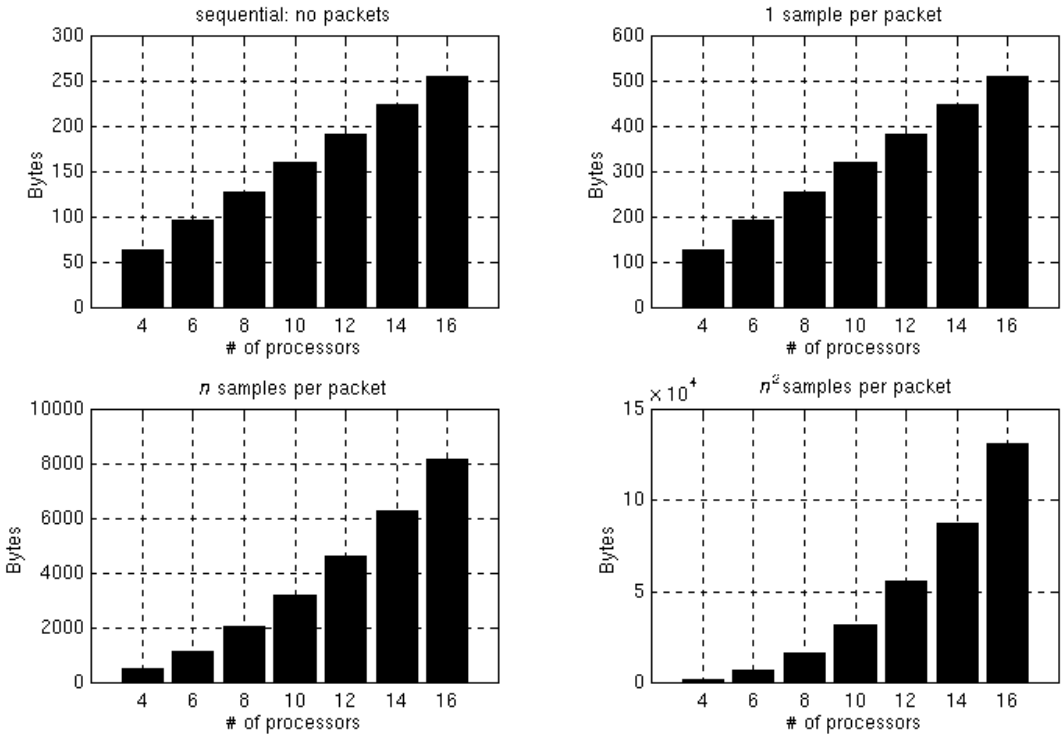


Figure 5.6. Data memory requirements for communication as a function of the number of processors for various payload sizes.

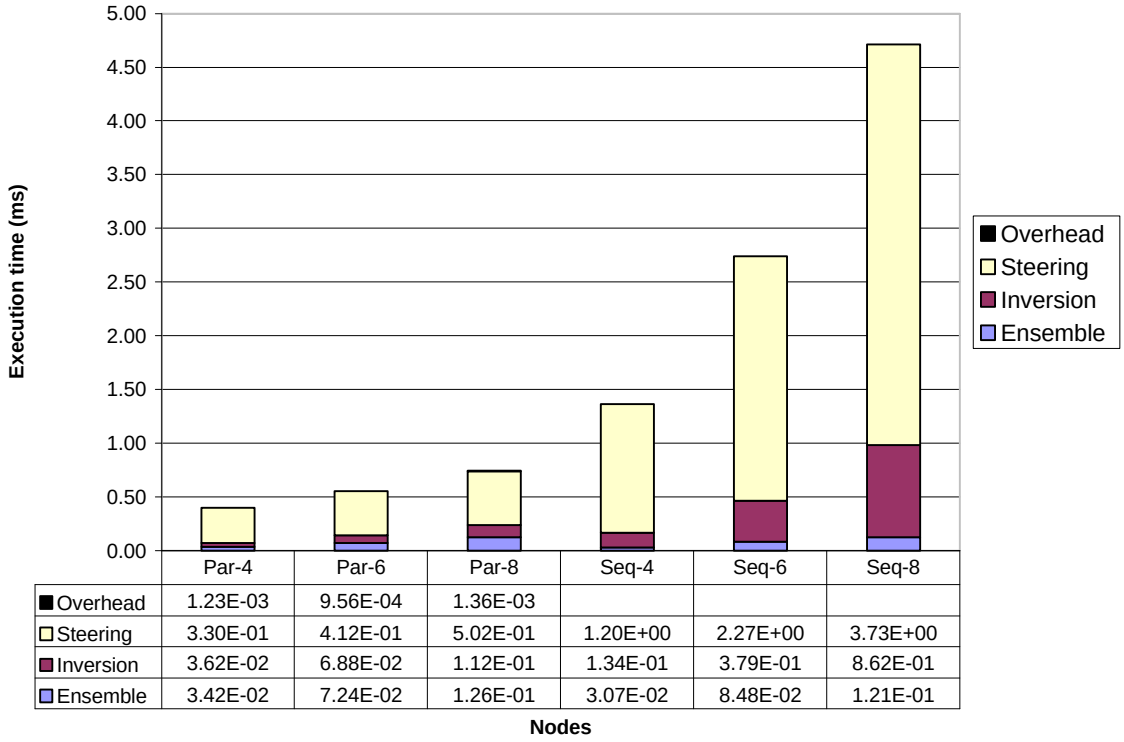
The need for data packing and multithreading in the DP-MVDR algorithm is most critical when the nodes in the array are connected by a unicast point-to-point communications network that is slow relative to processor speed. If the network is capable of inherently supporting hardware broadcast (e.g., a 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 chapter 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.

CHAPTER 6
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. Figure 6.1 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.

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 *ensemble average* 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.



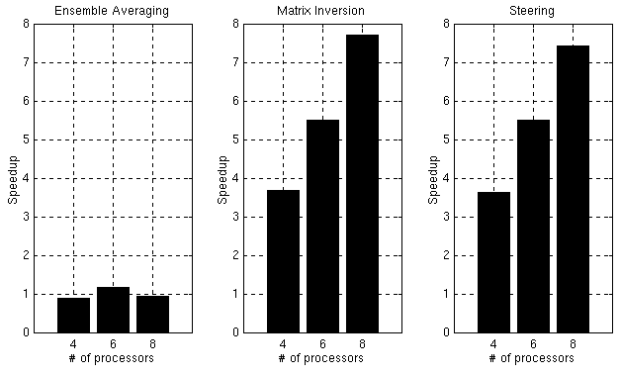
	Par-4	Par-6	Par-8	Seq-4	Seq-6	Seq-8
Overhead	1.23E-03	9.56E-04	1.36E-03			
Steering	3.30E-01	4.12E-01	5.02E-01	1.20E+00	2.27E+00	3.73E+00
Inversion	3.62E-02	6.88E-02	1.12E-01	1.34E-01	3.79E-01	8.62E-01
Ensemble	3.42E-02	7.24E-02	1.26E-01	3.07E-02	8.48E-02	1.21E-01

Figure 6.1. 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.

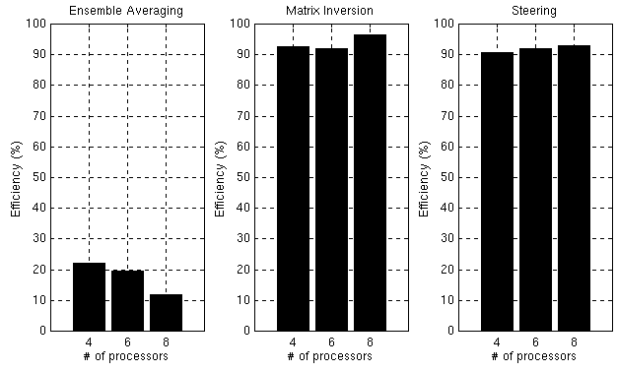
However, the *overhead* time of the parallel algorithm is negligible and therefore does not significantly affect the total execution time. The amount of time to perform the *ensemble averaging* task is comparable for both algorithms since it is not affected by the pipelining in DP-MVDR.

The two decomposed tasks in DP-MVDR, *matrix inversion* and *steering*, show significant improvement over their sequential counterparts. Figure 6.2 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 per task. Parallel efficiency is defined as the ratio of speedup versus the number of processing nodes employed in the system.



(a)



(b)

Figure 6.2. Individual task performance as a function of the number of processors. (a) Scaled speedup per task; (b) Parallel efficiency per task.

As expected, the *ensemble averaging* task exhibits no speedup since it is performed identically in both the sequential and parallel algorithms. The *matrix*

inversion and *steering* tasks achieve near-linear speedup with average parallel efficiencies of 94% and 92%, respectively. Since these two tasks were each decomposed into *n* stages, the idea 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.

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.

Figure 6.3 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%.

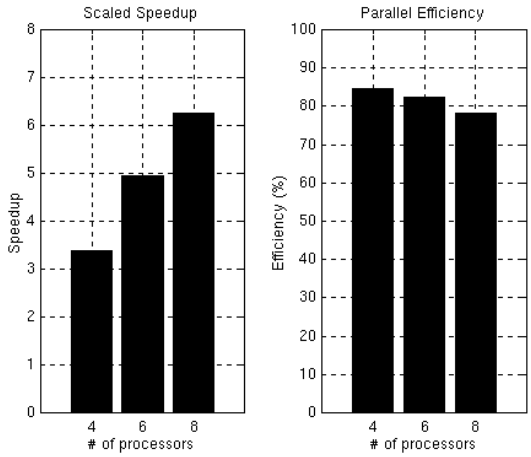


Figure 6.3. Overall scaled speedup and parallel efficiency as a function of the number of processors.

CHAPTER 7
CONCLUSIONS

This thesis introduces a new algorithm for parallel MVDR beamforming on distributed systems for in-array sonar signal processing, such as a sonar whose nodes are comprised of DSP processors with local memory and are interconnected by a point-to-point communications network. The methods employed provide considerable speedup with multiple nodes, thus enabling previously impractical algorithms to be implemented in real time. Furthermore, the fault tolerance of the sonar system can be increased by taking advantage of the distributed nature of this parallel algorithm and avoiding single points of failure.

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 on 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 DP-MVDR algorithm is defined in terms of a single frequency bin of interest, it is easily extensible to 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 ensemble 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 ensemble 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.

Several directions for future research are anticipated. An implementation of this new parallel algorithm for MVDR beamforming targeted toward an embedded, distributed system based on low-power DSP devices is anticipated. Moreover, to complement the performance-oriented emphasis in this thesis, further work is needed to ascertain strengths and weaknesses of the sequential and parallel MVDR algorithms in terms of fault tolerance and system reliability. Finally, techniques described in this thesis 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.

APPENDIX A
DATA MODEL

The data model used in the experiments with the DP-MVDR algorithm is based on a model described by Schmidt[25] for use with the MUSIC algorithm, but it is also compatible with the MVDR algorithm. Using Schmidt's notation, the wavefronts received at the *M* array elements are linear combinations of the *D* incident wavefronts and noise. Thus, the model for the received data vector *X* is

$$\begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_M \end{bmatrix} = \begin{bmatrix} a(q_1) & a(q_2) & \cdots & a(q_D) \end{bmatrix} \begin{bmatrix} F_1 \\ F_2 \\ \vdots \\ F_D \end{bmatrix} + \begin{bmatrix} W_1 \\ W_2 \\ \vdots \\ W_M \end{bmatrix} \tag{A.1}$$

or

$$X = AF + W\,. \tag{A.2}$$

The complex vector, *F*, represents the *D* incident signals by amplitude and phase. The complex vector, *W*, represents the noise at each sensor, whether it is isotropic noise (i.e., noise arriving from all directions with random amplitude) or measurement noise (i.e., instrumentation noise). The elements of the *A* matrix, $a(q_k)$, are column vectors of the sensor response as a function of the signal of arrival angles and the sensor location. In other words, a_{jk} depends on the position of the j^{th} sensor and its response to a signal incident from the direction of the k^{th} signal. The equation for $a(q_k)$ is:

$$a(q_k) = \exp(-i * d_j * t_{jk} * 2p * f) \quad \text{(A.3)}$$

where $i = \sqrt{-1}$, d_j is the relative position of the j^{th} sensor from the center of the array, t_{jk} is the time delay from the k^{th} signal sensed at the j^{th} sensor, and f is the frequency of the signal. In this thesis, the signal is generated for only one frequency bin.

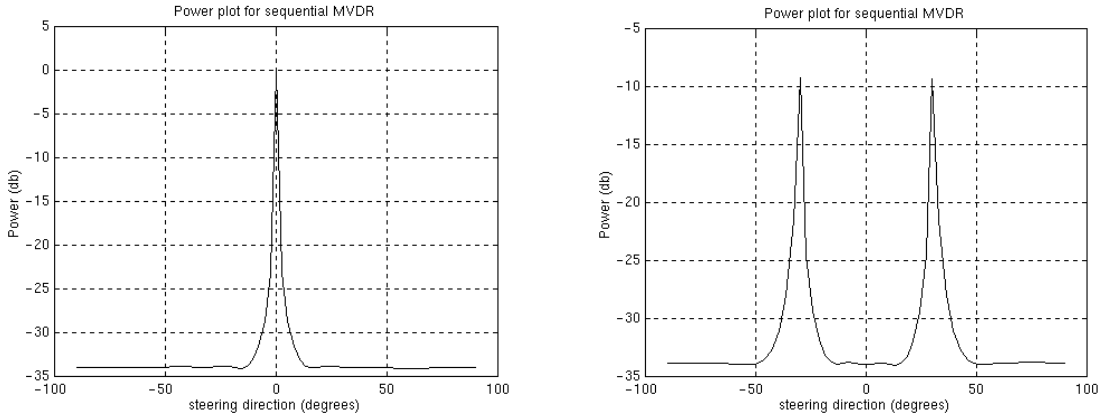
APPENDIX B
VERIFICATION OF RESULTS

The following plots confirm the ability of the parallel and sequential code in their ability to locate signals in the presence of incoherent, isotropic noise, as a correct MVDR algorithm should. The programs do not calculate the weight vectors that determine the beampattern explicitly, so no beampattern plots were generated. The plots shown are for the 8-node configuration. The test data had the following characteristics:

Table B.1. Signal characteristics

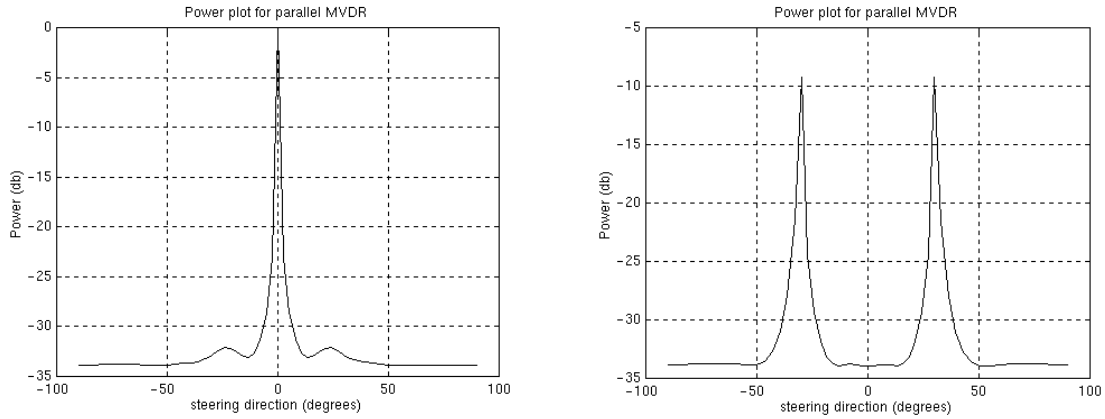
Signal Frequency	30 Hz
Signal-to-noise ratio	25 dB
Direction of arrival	0°, -30° and 30°
Noise component	Isotropic noise

Figures B.1a and B.1b show how the sequential algorithm resolved the signals arriving from 0°, and -30° and 30°, respectively. The two signals arriving from -30° and 30° were not perfectly resolved, as their gain in dB is less than 0. This less than ideal resolution is an inherent limitation in the MVDR algorithm. More advanced algorithms such as MUSIC exhibit better resolution. The power plots for the parallel algorithm in Figure B.2 are almost identical to those of the sequential algorithm, which is evidence of its correctness. They are not identical because they were executed using different random data generated with the same general characteristics.



(a) (b)

Figure B.1. Sequential 8-node results.
(a) Signal arriving from 0°; (b) Signals arriving from –30° and 30°.



(a) (b)

Figure B.2. Parallel 8-node results.
(a) Signal arriving from 0°; (b) Signals arriving from –30° and 30°.

APPENDIX C
PARALLEL AND SEQUENTIAL C++ CODE

The data used by each of the following programs was generated in MATLAB and written as text files to be read by the programs. For the parallel algorithm, the MATLAB program generates a separate data file for each node to read. Each node that executes the parallel program is identified by a rank number, a unique integer assigned to it by a call to the function *MPI_Comm_rank*. The node knows which file to read by finding it's rank number in the name of the data file. For the sequential algorithm, the MATLAB program generates one data file with a matrix of values whose columns pertain to each node.

File: *inline_par.C*

The *inline_par.C* file is the parallel program each node executes.

```
/* Minimum Variance Distortionless Response (MVDR) Beamforming
      Jesus Garcia
      Parallel program for MVDR algorithm
      High Performance Computing and Simulation Research Lab

      garcia@hcs.ufl.edu
      Began: 12/2/98
      Last modified: 5/99
      Filename: ~garcia/dpsa/c/par/parbak/inline_par.C

*/
#include <nodedata.h>
#include <setup.h>
#include <pthread.h>
#include <stdio.h>
#include <abf.h>
#include <mpi.h>
```

```
#include <string.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <iostream.h>
#include <signal.h>
#include <fstream.h>
#include <time.h>
#include <assert.h>

#ifndef _ISE
  #define PON ;
  #define POFF ;
  #define TOTAL_ITER 576
#else
  #define TOTAL_ITER 576
#endif

#define SWAP(a, b) {temp=(a);(a)=(b);(b)=temp;}    //Macro used in
                                                   //matrix inversion
                                                   //for pivoting

/* Global variables used by communication and main threads */

int sig1_received = 0, signal_received_main = 0;
int my_rank, stage=0, p, time_avg = 10000, numdata;
int *done = new int(0);
int *startnow = new int(0);
int size;
void **tstatus;
DPSAcomplex **array, *x;
MPI_Status status;
hrtime_t comm_start, comm_end, *comm_time, thread_start, thread_end,
*thread_time;
int numcomm=0, numcsm=0, numinv=0, numbeam=0, numthread =0, iter=-1,
total, count=0, sys;

pthread_mutex_t my_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
pthread_mutex_t start_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t start_cond = PTHREAD_COND_INITIALIZER;
pthread_mutex_t datalock = PTHREAD_MUTEX_INITIALIZER;
pid_t parent;

/*********************************************************************
void thread_began (int signo)

Description:
Signal handler to alert main program when the communication thread has
begun.
*********************************************************************/
```

```
void thread_began(int signo)
{
  pthread_mutex_lock(&my_lock);
  signal_received_main = 1;
  pthread_mutex_unlock(&my_lock);
}


/*********************************************************************
void *communicate (void *arg)

Description:
Thread dedicated to communication with other nodes.
*********************************************************************/
void *communicate(void *arg)
{

  MPI_Status status;
  int mult;
  mult = sizeof(DPSAcomplex)/sizeof(MPI_DOUBLE);


  if (kill(parent, SIGUSR2)==-1) {
     cout <<"Node "<<my_rank;perror(" could not send signal");
    cout <<endl;
    exit(1);
  }

  while (numcomm < TOTAL_ITER*p/numdata - 1) {

   /* Wait here until main program tells you to begin */

    pthread_mutex_lock(&start_lock);
    while (!(*startnow))
      pthread_cond_wait(&start_cond, &start_lock);
    *startnow = 0;
    pthread_mutex_unlock(&start_lock);

    pthread_mutex_lock(&datalock);
    memcpy(array[my_rank], x, numdata*sizeof(DPSAcomplex));
   pthread_mutex_unlock(&datalock);

   /* Barrier synchronization */
    MPI_Barrier(MPI_COMM_WORLD);
#ifdef _ISE
    comm_start = MPI_Wtime();
#else
    comm_start = gethrtime();
#endif
  /* Begin communication */
    for (int comvar=0;comvar<p;comvar++)
      if (comvar!=my_rank) {
      if (MPI_Send(array[my_rank], mult*numdata, MPI_DOUBLE, comvar,
                           numcomm*my_rank+numcomm, MPI_COMM_WORLD)) {
```

```
        cout <<"Node "<<my_rank;
        perror(" could not send");
        exit(1);
      }

      }
    for (int comvar=0;comvar<p;comvar++) {

      if (comvar!=my_rank) {
        if (MPI_Recv(array[comvar], mult*numdata, MPI_DOUBLE, comvar,
             numcomm*comvar+numcomm, MPI_COMM_WORLD, &status)) {
             cout <<"Node "<<my_rank;
             perror(" could not receive");
             exit(1);
        }
      }
    }


#ifdef _ISE
    comm_end = MPI_Wtime();
#else
    comm_end = gethrtime();
#endif
   /* End communication and tell main program you are done
      communicating*/

    pthread_mutex_lock(&my_lock);
    *done=1;
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&my_lock);
    comm_time[numcomm] = comm_end - comm_start;
    numcomm++;

  }

 return NULL;
}

/*********************************************************************
int main (int argc, char **argv)

Description:
Main program that handles all calculations and controls communication
thread.
*********************************************************************/
int main(int argc, char **argv)
{


  pthread_t pid;

  FILE            *timeout;
  DPSAcomplex     ***steering_vectors;
```

```
  DPSAcomplex     **inverse_csm, **prev_inv, **currentarray;
  DPSAcomplex     *col;
  DPSAcomplex     w2, *w3, one;
  double          *final;
  double          lambda = 0.9;   //forgetting factor
  double          center = 0;
  double          *degs;
  double          *node_distances;
  double          **delay;
  int source;
  int dest;
  int name_length;
  char  str2[10], matrixstr[12], currentarraystr[18];
  char my_name[11];
  int ise_iter;
  int index;
  DPSAcomplex **csm;
  DPSAcomplex zero;
  zero.real = zero.imag = 0.0;
  int num_of_fft = NUM_OF_FFT;
  int handling_freq_bin;
  int num_of_steering_angles = NUM_OF_STEERING_ANGLES;
  int num_of_output_angles = NUM_OF_OUTPUT_ANGLES;
  int num_of_samples = NUM_OF_SAMPLES;
  int length_of_window = LENGTH_OF_WINDOW;
  int m,g,j,k,i, h;
  int imax;
  DPSAcomplex  dum, sum;
  int icol, l, irow,ll;
  DPSAcomplex pivinv, temp;
  int *indxc, *indxr, *ipiv;
  double big;
  int ii=0, ip;
  double speed_of_sound = SPEED_OF_SOUND;
  double sampling_freq = SAMPLING_FREQ;
  double spacing = SPACING;
  double temp1, temp2;
  double freq_resolution;
  double epsilon = EPSILON;
  char filename[80];

  int numtotal=0;


  /* MPI Initialization */

#ifdef _ISE
  MPI_Notime_init(&argc, &argv);
#else
  MPI_Init(&argc, &argv);
#endif
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &p);
  MPI_Get_processor_name(my_name, &name_length);
```

```
  my_name[name_length-1] = '\0';

#ifdef _ISE
  double  beam_start, beam_end, beam_time;
  double steering_start, steering_end, steering_time, inverse_start,
  inverse_end, inverse_time;

  double csm_start, csm_end, csm_time;
  double total_start, total_end, total_time;
  double comm_start, comm_end, comm_time;
#else
  hrtime_t beam_start, beam_end, *beam_time;
  hrtime_t csm_start, csm_end, *csm_time, inverse_start, inverse_end,
  *inverse_time;

#endif



  if (p==4) numdata = p*p;
  else numdata = p*p;


  csm_time  = new hrtime_t[TOTAL_ITER*p](0);
  comm_time = new hrtime_t[TOTAL_ITER*p/numdata-1](0);
  beam_time = new hrtime_t[TOTAL_ITER*p - 2*p](0);
  inverse_time = new hrtime_t[TOTAL_ITER*p - p](0);
  thread_time = new hrtime_t[TOTAL_ITER*p/numdata-1](0);

  char *ext;

  if (my_rank<10) {
    ext = new char[6];
    ext = "_.dat";
  }
  else {
    ext = new char[5];
    ext = ".dat";
  }
  /* File handling code */

  char my_num[3], nodes[3];
  itoa(my_rank, my_num);
  itoa(p, nodes);

  strcpy(str, "sim");
  strcat(str, my_num);
  strcat(str, ext);
  strcpy(datastr, "node");
  strcat(datastr, my_num);
  strcat(datastr, ext);

  strcpy(commstr, "commtimes");
  strcat(commstr, my_num);
```

```
  strcat(commstr, ext);

  strcpy(csmstr, "csmtimes");
  strcat(csmstr, my_num);
  strcat(csmstr, ext);

  strcpy(threadstr, "threadtimes");
  strcat(threadstr, my_num);
  strcat(threadstr, ext);

  strcpy(invstr, "invtimes");
  strcat(invstr, my_num);
  strcat(invstr, ext);

  strcpy(beamstr, "beamtimes");
  strcat(beamstr, my_num);
  strcat(beamstr, ext);

  ifstream mydata(datastr,ios::in);
  freq_resolution = sampling_freq/num_of_fft;
  handling_freq_bin = (int)((num_of_fft/2)+1);

  /*Beamforming variables*/


  one.real = 1;
  one.imag = 0;
/* ---------------- Allocate dynamic memory ---------------- */



  //tempcsm = (DPSAcomplex**)malloc(p*sizeof(DPSAcomplex*));
  array = new DPSAcomplex*[p];
  currentarray = new DPSAcomplex*[p];
  csm = (DPSAcomplex**)malloc(p*sizeof(DPSAcomplex*));
  inverse_csm = (DPSAcomplex**)malloc(p*sizeof(DPSAcomplex*));
  prev_inv = (DPSAcomplex**)malloc(p*sizeof(DPSAcomplex*));
  x = new DPSAcomplex[numdata];
  col = new DPSAcomplex[p];
  final = (double*)malloc(num_of_steering_angles*sizeof(double));
  w3 = new DPSAcomplex[num_of_steering_angles];
  degs = (double*)malloc(NUM_OF_STEERING_ANGLES*sizeof(double));
  node_distances = (double*)malloc(p*sizeof(double));
  steering_vectors =
(DPSAcomplex***)malloc(NUM_OF_STEERING_ANGLES*sizeof(DPSAcomplex**));
  delay = (double**)malloc(NUM_OF_STEERING_ANGLES*sizeof(double*));
  indxc = new int[p];
  indxr = new int[p];
  ipiv = new int[p];

  for (m=0; m<p; m++) {
    array[m] = new DPSAcomplex[numdata](zero);
    currentarray[m] = new DPSAcomplex[numdata](zero);
    prev_inv[m] = (DPSAcomplex*)malloc(p*sizeof(DPSAcomplex));
```

```
  inverse_csm[m] = (DPSAcomplex*)malloc(p*sizeof(DPSAcomplex));
  csm[m] = (DPSAcomplex*)malloc(p*sizeof(DPSAcomplex));
  // tempcsm[m] = (DPSAcomplex*)malloc(p*sizeof(DPSAcomplex));
}

for(i=0; i<NUM_OF_STEERING_ANGLES; i++){
  delay[i] = (double*)malloc(p*sizeof(double));
  steering_vectors[i] =
 (DPSAcomplex**)malloc(handling_freq_bin*sizeof(DPSAcomplex*));
  w3[i] = zero;
  final[i] = 0.0;
  for(j=0; j<handling_freq_bin; j++)
    steering_vectors[i][j] =
   (DPSAcomplex*)malloc(p*sizeof(DPSAcomplex));
}

for (i=0;i<p;i++)
  for (m=0;m<p;m++)
    csm[i][m]=inverse_csm[i][m]=prev_inv[i][m] = zero;

/* ---------- Calculate relative position of the nodes ----------- */

for(i=0; i<p; i++) {
  node_distances[i] = i*SPACING;
  center += node_distances[i];
}
center /= p;

for(i=0; i<p; i++) {
  node_distances[i] -= center;
}

/* ---------------- Calculate steering vectors ---------------- */

k = num_of_steering_angles%2;

if(k==1) {
  temp1 = (num_of_steering_angles-1)/2;
  temp2 = 1/temp1;
  for(j=0; j<num_of_steering_angles; j++)
    degs[j] = (-1+(j*temp2));
}
else
  return -1;

for(i=0; i<num_of_steering_angles; i++)
  for(j=0; j<p; j++)
    delay[i][j] = node_distances[j]*degs[i]/speed_of_sound;

 free(node_distances);
for(i=0; i<num_of_steering_angles; i++) {
  degs[i] = 180*asin(degs[i])/PI;
  for(j=0; j<handling_freq_bin; j++) {
    for(k=0; k<p; k++) {
```

```
    temp1 = 2.0*PI*freq_resolution*j*delay[i][k];
    steering_vectors[i][j][k] = DPSA_Complex_etothej(temp1);
    }
  }
}

free(degs);
/*--------------Get first matrix: initialization -----------------*/


strcpy(matrixstr, "matrix");
strcat(matrixstr, nodes);
strcat(matrixstr, ext);

strcpy(currentarraystr, "currentarray");
strcat(currentarraystr, nodes);
strcat(currentarraystr, ext);

ifstream inmatrix(matrixstr,ios::in);
for (m = 0;m<p;m++)
  for (l = 0;l<p;l++)
    inmatrix >> csm[m][l].real >> csm[m][l].imag;


inmatrix.close();

ifstream incurrentarray(currentarraystr,ios::in);
for (l = 0;l<p;l++)
  for (int t=0;t<numdata;t++)
    incurrentarray >> currentarray[l][t].real >>
    currentarray[l][t].imag;

incurrentarray.close();

/*------------------- END of initialization --------------------*/

/* Signal handler setup and thread creation */

int firstpause = 1;
int first = 1;
int firstthread = 1;
sigset_t sigset;
struct sigaction doneact;

doneact.sa_handler = thread_began;
sigemptyset(&doneact.sa_mask);
doneact.sa_flags = 0;




parent = getpid();

sigemptyset(&sigset);
```

```
  sigaddset(&sigset, SIGUSR2);
  if (sigaction(SIGUSR2, &doneact, NULL)==-1)
    perror("Could not install signal handler");

  if (pthread_create(&pid, NULL, communicate, NULL)) {
    perror("Thread creation was not successful");
    exit(1);
  }
/* Wait until signal handler returns */

  while(signal_received_main==0)
    pause();

  signal_received_main = 0;

 /* Barrier synchronization */

  MPI_Barrier(MPI_COMM_WORLD);

  cout <<"Node "<<my_rank<<": "<<my_name<<" is ready to
       begin..."<<endl;
  for(sys=0; sys<TOTAL_ITER; sys++)
    {

     for (total=0;total<p;total++) {
      iter++;

      /* Read data */
      if ((first || count==numdata-1) && sys < (TOTAL_ITER-
          (2*numdata/p))) {
        first=0;

        for (int t=0;t<numdata;t++)
          mydata >> x[t].real >> x[t].imag;
      }


      //  -------------- End reading data -----------


/* Tell thread to begin first communication cycle */
        if (firstthread) {
          firstthread = 0;
          MPI_Barrier(MPI_COMM_WORLD);
          pthread_mutex_lock(&start_lock);
          *startnow=1;
          pthread_cond_signal(&start_cond);
          pthread_mutex_unlock(&start_lock);
        }

        if (count==numdata) {
          cout<<"Node "<<my_rank<<" count = "<<count<<" and must
               exit"<<endl;
          exit(1);
```

```
        }

/* CSDM creation and summation to running average */
#ifdef _ISE
        csm_start = MPI_Wtime();
#else
        csm_start = gethrtime();
#endif

        for (i=0;i<p;i++)
          for(j=0; j<p;j++){
            if (currentarray[i][count]==zero ||
                                        currentarray[j][count]==zero) {
            cerr<<"Must exit due to currentarray element = 0"<<endl;
            exit(1);
            }
            csm[i][j] +=
                        currentarray[i][count]*!currentarray[j][count];
          }
#ifdef _ISE
        csm_end = MPI_Wtime();
#else
        csm_end = gethrtime();
#endif

        csm_time[numcsm++] = csm_end - csm_start;
        count++;


        if (count==numdata && sys<TOTAL_ITER-(numdata/p)  ) {

      /* Wait for thread to tell you it is done communicating */
          pthread_mutex_lock(&my_lock);
          thread_start = gethrtime();
          while (!(*done))
            pthread_cond_wait(&cond, &my_lock);
          thread_end = gethrtime();
          *done = 0;
          pthread_mutex_unlock(&my_lock);

          thread_time[numthread] = thread_end - thread_start;
          numthread++;

          count = 0;
           pthread_mutex_lock(&datalock);
          for (i=0;i<p;i++)
            memcpy(currentarray[i], array[i],
           numdata*sizeof(DPSAcomplex));
           pthread_mutex_unlock(&datalock);

          if (sys < (TOTAL_ITER-(2*numdata/p))) {
         /* Barrier synchronization */
            MPI_Barrier(MPI_COMM_WORLD);
```

```
        /* Tell thread to begin new communication cycle */
            pthread_mutex_lock(&start_lock);
            *startnow=1;
            pthread_cond_signal(&start_cond);
            pthread_mutex_unlock(&start_lock);
          }
        }


        if (!(sys==TOTAL_ITER-1 && total>=my_rank)) {

          if (total == my_rank) {

/* Ensemble averaging */
#ifdef _ISE
            csm_start = MPI_Wtime();
#else
            csm_start = gethrtime();
#endif

      for (m = 0;m<p;m++)
            for (j = 0;j<p;j++)
              inverse_csm[m][j] = csm[m][j]/(iter+time_avg);

#ifdef _ISE
            csm_end = MPI_Wtime();
#else
            csm_end = gethrtime();
#endif

            csm_time[numcsm-1] += csm_end - csm_start;


            for (j=0;j<p;j++) ipiv[j] = 0;

            stage = 0;
          }


          if (iter >= my_rank) {
/* Begin one stage of Gauss-Jordan elimination for matrix inversion */
#ifdef _ISE
            inverse_start = MPI_Wtime();
#else
            inverse_start = gethrtime();
#endif
            big = 0.0;
            for (j=0;j<p;j++)
            if (ipiv[j] != 1)
              for (k=0;k<p;k++) {
                if (ipiv[k]==0) {
                  if (DPSA_Complex_abs(inverse_csm[j][k]) >= big) {
                  big = DPSA_Complex_abs(inverse_csm[j][k]);
                  irow = j;
```

```
                icol = k;
                }
              } else if (ipiv[k]>1) exit(1);
            }
          ++(ipiv[icol]);

          if (irow!=icol)
          for (l=0;l<p;l++) SWAP(inverse_csm[irow][l],
              inverse_csm[icol][l]);

          indxr[stage] = irow;
          indxc[stage] = icol;

          if (inverse_csm[icol][icol] == zero) exit(1);
          pivinv = 1.0/inverse_csm[icol][icol];

          inverse_csm[icol][icol].real =
          1.0;inverse_csm[icol][icol].imag = 0.0;
          for (l=0;l<p;l++) inverse_csm[icol][l] *= pivinv;
          for (ll=0;ll<p;ll++)
          if (ll!=icol) {
            dum = inverse_csm[ll][icol];
            inverse_csm[ll][icol] = zero;
            for (l=0;l<p;l++) inverse_csm[ll][l] -=
                                          inverse_csm[icol][l]*dum;
          }
          if (stage == (p-1)) {
            for (l=p-1;l>=0;l--) {
              if (indxr[l] != indxc[l])
                for (k=0;k<p;k++)
               SWAP(inverse_csm[k][indxr[l]],inverse_csm[k][indxc[l]]);
            }
          }
          stage++;
#ifdef _ISE
          inverse_end = MPI_Wtime();

#else
          inverse_end = gethrtime();
#endif

          inverse_time[numinv++] = inverse_end - inverse_start;

/* End matrix inversion stage */

          if (stage == p) {
          // Saving it as transpose for faster memory access
          for (i=0;i<p;i++)
            for (j=0;j<p;j++)
              prev_inv[i][j] = inverse_csm[j][i];
          }
/* Begin one steering stage */
          if (iter >= p+my_rank) {
#ifdef _ISE
```

```
            beam_start = MPI_Wtime();
#else
            beam_start = gethrtime();
#endif

            for (k = 0;k<num_of_steering_angles;k++) {
              w2 = zero;

              for (j=0;j<p;j++)
                w2 += !steering_vectors[k][21][j]*prev_inv[stage-1][j];

              w3[k] += w2 * steering_vectors[k][21][stage-1];

              if (stage == p) {
                final[k] += DPSA_Complex_abs(1.0/(w3[k]));
                w3[k] = zero;
              }
            }
#ifdef _ISE
            beam_end =  MPI_Wtime();
#else
            beam_end = gethrtime();
#endif

            beam_time[numbeam++] = beam_end-beam_start;

/* End steering stage */
            }
          }
        }
      }
    }

  MPI_Barrier(MPI_COMM_WORLD);

/* Kill the thread */
  pthread_join(pid, NULL);
  MPI_Finalize();

/* Do final averaging of results */
  for (k = 0;k<num_of_steering_angles;k++) final[k]/=((TOTAL_ITER-
       2)*num_of_steering_angles/p);

/* ================= End of beamforming ================== */
  mydata.close();

  ofstream dataout(str, ios::out);
  for(i=0; i<NUM_OF_STEERING_ANGLES; i++)
    dataout <<final[i]<<endl;
  dataout.close();
  free(final);

  // Deallocate memory -------------------------
```

```
for (m=0;m<p;m++) {
  free(inverse_csm[m]);
  free(prev_inv[m]);
  free(csm[m]);
}


for(i=0; i<NUM_OF_STEERING_ANGLES; i++) {
  free(delay[i]);
  for(j=0; j<handling_freq_bin; j++ )
    free(steering_vectors[i][j]);
  free(steering_vectors[i]);
 }

free(steering_vectors);
delete [] ipiv;
delete [] indxr;
delete [] indxc;
delete done;
delete startnow;
delete [] col;
free(delay);
free(csm);
free(prev_inv);
free(inverse_csm);
delete [] x;
delete [] w3;
delete [] currentarray;
delete [] array;
/* Write timing results to files */
//----------------------------------Do Commtimes -------------------
ofstream commout(commstr, ios::out);

if (!commout) {
  cerr<<"Could not open commtimes_.dat"<<endl;
  exit(1);
}

for (int z=0;z<numcomm;z++) {
  commout <<comm_time[z];
  commout <<endl;
}


commout.close();

//----------------------------------End Commtimes -------------------

//----------------------------------Do CSMtimes ---------------------

ofstream cutitout(csmstr, ios::out);

if (!cutitout) {
    cerr<<"Could not open csmtimes_.dat"<<endl;
```

```
    exit(1);
    }

 for (int z=0;z<numcsm;z++) {
  cutitout <<csm_time[z];
  cutitout <<endl;

}

    cutitout.close();

    //----------------------------------End CSMtimes ----------------

    //----------------------------------Do INVtimes -----------------

    ofstream invout(invstr, ios::out);

    if (!invout) {
    cerr<<"Could not open invtimes_.dat"<<endl;
    exit(1);
    }

    for (int z=0;z<numinv;z++) {
    invout <<inverse_time[z];
    invout <<endl;
    }

    invout.close();
    //----------------------------------End INVtimes ----------------

    //---------------------------------Do threadtimes ---------------

    ofstream threadout(threadstr, ios::out);

    if (!threadout) {
    cerr<<"Could not open threadtimes_.dat"<<endl;
    exit(1);
    }

    for (int z=0;z<numthread;z++) {
    threadout <<thread_time[z];
    threadout <<endl;
    }

    threadout.close();
    //---------------------------------End threadtimes --------------

    //---------------------------------Do beamtimes -----------------

 ofstream beamout(beamstr, ios::out);

 if (!beamout) {
   cerr<<"Could not open beamtimes_.dat"<<endl;
   exit(1);
```

```
   }

   for (int z=0;z<numbeam;z++) {
     beamout <<beam_time[z];
     beamout <<endl;
   }

   beamout.close();
   //---------------------------------End beamtimes ------------------

   delete [] inverse_time;
   delete [] comm_time;
   delete [] thread_time;
   delete [] csm_time;
   delete [] beam_time;
   cout <<"Node "<<my_rank<<": "<<my_name<<" has ended!"<<endl;

}
```

File: *inline_strict_sequential.C*

The *inline_strict_sequential.C* program is executed by a single workstation from the same cluster that executed the parallel code.

```
/* Minimum Variance Distortionless Response (MVDR) Beamforming
      Jesus Garcia
      Sequential program of MVDR algorithm
      High Performance Computing and Simulation Research Lab

      garcia@hcs.ufl.edu
      Began: 12/2/98
      Last modified: 5/99
      Filename: ~garcia/dpsa/c/seq/mincomp/strictly_sequential/
                inline_strict_sequential.C
*/

#include <setup.h>
#include <abf.h>
#include <sys/time.h>
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
#include <math.h>
#include <time.h>


#ifndef _ISE
  #define PON ;
  #define POFF ;
  #define TOTAL_ITER 576
```

```
#else
  #define TOTAL_ITER 576
#endif
/* Macro used in matrix inversion for pivoting */
#define SWAP(a, b) {temp=(a);(a)=(b);(b)=temp;}

int main(int argc, const char *argv[])
{
  srand(time(NULL));

  FILE            *timeout;
  DPSAcomplex     ***steering_vectors;
  DPSAcomplex     **csm, **inverse_csm;
  DPSAcomplex     *w2,*w3, one, *x;
  DPSAcomplex     tempcomplex;
  double          *final;
  double          *angles;
  double          rad;
  double          center = 0;
  double          *degs;
  double          *node_distances;
  double          **delay;
  double          value1, value2;
  double          db;
  double d;
  DPSAcomplex **tempcsm;
  DPSAcomplex zero;
  int time_avg = 10000;
  zero.real = zero.imag = 0.0;
  int num_of_steering_angles = NUM_OF_STEERING_ANGLES;
  int num_of_nodes = NUM_OF_NODES;
  int num_of_output_angles = NUM_OF_OUTPUT_ANGLES;
  int num_of_samples = NUM_OF_SAMPLES;
  int length_of_window = LENGTH_OF_WINDOW;
  int half_node;
  int m,g,j,k,i,div_pt, sys, h;
  int ise_iter;
  int icol, l, irow,ll;
  double big;
  DPSAcomplex  dum, pivinv, temp;
  int *indxc, *indxr, *ipiv;
  double speed_of_sound = SPEED_OF_SOUND;
  double sampling_freq = SAMPLING_FREQ;
  double spacing = SPACING;
  double temp1, temp2;

  double freq_resolution;

  char filename[80], matrixstr[12], currentarraystr[16];

  /* Timing variables */
#ifdef _ISE
  double  beam_start, beam_end, beam_time;
  double steering_start, steering_end, steering_time, inverse_start,
```

```
inverse_end, inverse_time;
  double csm_start, csm_end, csm_time;
  double total_time;
#else
  hrtime_t beam_start, beam_end, beam_time;
  hrtime_t csm_start, csm_end, csm_time, inverse_start, inverse_end,
inverse_time;
  hrtime_t total_time;

#endif


  csm_time = beam_time = total_time = inverse_time = 0;


#ifdef _ISE
  MPI_Iteration(&ise_iter);
#endif
  num_of_nodes = atoi(*(argv+1));

  /*Beamforming variables*/


  w2 = (DPSAcomplex*)malloc(num_of_nodes*sizeof(DPSAcomplex));
  w3 = (DPSAcomplex*)malloc(sizeof(DPSAcomplex));
  one.real = 1;
  one.imag = 0;

  /* ---------------- Allocate dynamic memory ---------------- */


  csm = (DPSAcomplex**)malloc(num_of_nodes*sizeof(DPSAcomplex*));
  inverse_csm =
  (DPSAcomplex**)malloc(num_of_nodes*sizeof(DPSAcomplex*));
  final = (double*)malloc(num_of_steering_angles*sizeof(double));
  delay = (double**)malloc(NUM_OF_STEERING_ANGLES*sizeof(double*));
  degs = (double*)malloc(NUM_OF_STEERING_ANGLES*sizeof(double));
  angles = (double*)malloc(NUM_OF_OUTPUT_ANGLES*sizeof(double));
  node_distances = (double*)malloc(num_of_nodes*sizeof(double));
  tempcsm = (DPSAcomplex**)malloc(num_of_nodes*sizeof(DPSAcomplex*));
  indxc = new int[num_of_nodes];
  indxr = new int[num_of_nodes];
  ipiv = new int[num_of_nodes];
  x = new DPSAcomplex[num_of_nodes];
  for (m=0; m<num_of_nodes; m++) {
    csm[m] = (DPSAcomplex*)malloc(num_of_nodes*sizeof(DPSAcomplex));
    inverse_csm[m] =
                (DPSAcomplex*)malloc(num_of_nodes*sizeof(DPSAcomplex));
    tempcsm[m] =
                (DPSAcomplex*)malloc(num_of_nodes*sizeof(DPSAcomplex));

  }
```

```
steering_vectors =
(DPSAcomplex***)malloc(NUM_OF_STEERING_ANGLES*sizeof(DPSAcomplex**));

for(i=0; i<NUM_OF_STEERING_ANGLES; i++)
  {
    delay[i] = (double*)malloc(num_of_nodes*sizeof(double));
    steering_vectors[i] =
       (DPSAcomplex**)malloc(HANDLING_FREQ_BIN*sizeof(DPSAcomplex*));
    final[i] = 0.0;
    for(j=0; j<HANDLING_FREQ_BIN; j++)
    steering_vectors[i][j] =
              (DPSAcomplex*)malloc(num_of_nodes*sizeof(DPSAcomplex));
  }

/* -------- Calculate relative position of the nodes ------------- */

for(i=0; i<num_of_nodes; i++) {
  node_distances[i] = i*SPACING;
  center += node_distances[i];
}
center /= num_of_nodes;

for(i=0; i<num_of_nodes; i++) {
  node_distances[i] -= center;
}

/* ---------------- Calculate steering vectors ---------------- */

k = num_of_steering_angles%2;
freq_resolution = (double)SAMPLING_FREQ/NUM_OF_FFT;
if(k==1) {
  temp1 = (num_of_steering_angles-1)/2;
  temp2 = 1/temp1;
  for(j=0; j<num_of_steering_angles; j++) degs[j] = (-1+(j*temp2));
}
else
  return -1;

for(i=0; i<num_of_steering_angles; i++)
  for(j=0; j<num_of_nodes; j++)
    delay[i][j] = node_distances[j]*degs[i]/speed_of_sound;

for(i=0; i<num_of_steering_angles; i++) {
  for(j=0; j<HANDLING_FREQ_BIN; j++) {
    for(k=0; k<num_of_nodes; k++) {
    temp1 = 2.0*PI*freq_resolution*j*delay[i][k];
    steering_vectors[i][j][k] = DPSA_Complex_etothej(temp1);
    }
  }
}

for(i=0; i<num_of_steering_angles; i++)
  degs[i] = 180*asin(degs[i])/PI;
```

```
  /* =========== End of initial phase and start beamforming ======= */


  int n = num_of_nodes;

  for (m=0;m<num_of_nodes;m++)
    for (j=0;j<num_of_nodes;j++)
      inverse_csm[m][j] = csm[m][j] = tempcsm[m][j] = zero;

/* File handling */
  switch (num_of_nodes) {
  case 4:
    memcpy(matrixstr, "matrix4.dat", 12);memcpy(currentarraystr,
          "data_array4.dat",16);break;
  case 6:
    memcpy(matrixstr, "matrix6.dat", 12);memcpy(currentarraystr,
          "data_array6.dat",16);break;
  case 8:
    memcpy(matrixstr, "matrix8.dat", 12);memcpy(currentarraystr,
          "data_array8.dat",16);break;
  }

  ifstream inmatrix(matrixstr,ios::in);
  ifstream ourdata(currentarraystr, ios::in);

/* Read in initialization matrices */
  for (m = 0;m<num_of_nodes;m++)
    for (j = 0;j<num_of_nodes;j++)
      inmatrix >> tempcsm[m][j].real >> tempcsm[m][j].imag;


  inmatrix.close();

/* Begin algorithm */
  int total = TOTAL_ITER*num_of_nodes;

  for (sys=0;sys<total;sys++) {

    /* Read in the data */
    for (i=0;i<num_of_nodes;i++)
      ourdata >> x[i].real >> x[i].imag;

/* CSDM formation and ensemble averaging */
#ifdef _ISE
   csm_start = MPI_Wtime();
#else
    csm_start = gethrtime();
#endif

    for (m = 0;m<num_of_nodes;m++)
      for (j = 0;j<num_of_nodes;j++)
         tempcsm[m][j] += x[m]*!x[j];

    for (m = 0;m<num_of_nodes;m++)
```

```
      for (j = 0;j<num_of_nodes;j++)
         inverse_csm[m][j] = tempcsm[m][j]/(sys+1+time_avg);

#ifdef _ISE
    csm_end = MPI_Wtime();
#else
    csm_end = gethrtime();
#endif

    csm_time += csm_end - csm_start;


/* Matrix inversion using Gauss-Jordan elimination */
#ifdef _ISE
    inverse_start = MPI_Wtime();
#else
    inverse_start = gethrtime();
#endif

    //--------------- Matrix inverse -----------------------


    for (j=0;j<n;j++) ipiv[j] = 0;
    for (i=0;i<n;i++) {
      big = 0.0;
      for (j=0;j<n;j++)
      if (ipiv[j] !=1)
        for (k=0;k<n;k++) {
          if (ipiv[k]==0) {
            if (DPSA_Complex_abs(inverse_csm[j][k]) >= big) {
            big = DPSA_Complex_abs(inverse_csm[j][k]);
            irow = j;
            icol = k;
            }
          } else if (ipiv[k]>1) exit(1);
        }
      ++(ipiv[icol]);
      if (irow!=icol)
      for (l=0;l<n;l++) SWAP(inverse_csm[irow][l],
                             inverse_csm[icol][l]);
      indxr[i] = irow;
      indxc[i] = icol;
      if (inverse_csm[icol][icol] == zero) exit(1);
      pivinv = 1.0/inverse_csm[icol][icol];
      inverse_csm[icol][icol].real = 1.0;inverse_csm[icol][icol].imag =
      0.0;
      for (l=0;l<n;l++) inverse_csm[icol][l] *= pivinv;
      for (ll=0;ll<n;ll++)
      if (ll!=icol) {
        dum = inverse_csm[ll][icol];
        inverse_csm[ll][icol] = zero;
        for (l=0;l<n;l++) inverse_csm[ll][l] -=
                                              inverse_csm[icol][l]*dum;
      }
```

```
    }
    for (l=n-1;l>=0;l--) {
      if (indxr[l] != indxc[l])
        for (k=0;k<n;k++)
           SWAP(inverse_csm[k][indxr[l]],inverse_csm[k][indxc[l]]);
    }


#ifdef _ISE
    inverse_end = MPI_Wtime();
#else
    inverse_end = gethrtime();
#endif
    inverse_time += inverse_end - inverse_start;
    /*------------------ End of Inverse ----------------------*/

    /* Begin beamforming */
#ifdef _ISE
    beam_start = MPI_Wtime();
#else
    beam_start = gethrtime();
#endif

    for (k = 0;k<NUM_OF_STEERING_ANGLES;k++) {
      for (i=0;i<num_of_nodes;i++) {
      w2[i] = zero;
      for (j=0;j<num_of_nodes;j++)
        w2[i] += !steering_vectors[k][21][j]*inverse_csm[j][i];
      }
      *w3 = zero;
      for(i=0; i<num_of_nodes; i++)
      *w3 += w2[i]*steering_vectors[k][21][i];

      final[k] += DPSA_Complex_abs(1.0/(*w3));
    }
#ifdef _ISE
    beam_end =  MPI_Wtime();
#else
    beam_end = gethrtime();
#endif
    beam_time += beam_end-beam_start;

}
/* Averaging of final results */


    for (k = 0;k<NUM_OF_STEERING_ANGLES;k++) final[k] /= total;



    /* ================= End of beamforming ================== */


    ofstream results("result.dat", ios::out);
```

```
  ofstream angles2("angles.dat", ios::out);
  results.precision(6);

  for(i=0; i<NUM_OF_STEERING_ANGLES; i++) {
    // cout <<final[i]<<endl;
  results<<final[i]<<endl;
  angles2 <<degs[i]<<endl;
  }

  angles2.close();



  beam_time /= total;
  csm_time /= total;
  inverse_time /= total;

results <<"avg_csm "<<csm_time<<endl;
results <<"avg_inverse "<<inverse_time<<endl;
results <<"avg_beam "<<beam_time<<endl;

results.close();



  /* ---------------- Release dynamic memory ---------------- */

  for (m=0;m<num_of_nodes;m++) {
    free(inverse_csm[m]);
    free(tempcsm[m]);
    free(csm[m]);
  }

for(i=0;i<NUM_OF_STEERING_ANGLES;i++) {
  free(delay[i]);
  for(j=0;j<HANDLING_FREQ_BIN;j++) free(steering_vectors[i][j]);
  free(steering_vectors[i]);
}

free(steering_vectors);
delete [] ipiv;
delete [] indxr;
delete [] indxc;
delete [] x;
free(delay);
free(csm);
free(inverse_csm);
free(w2);
free(w3);
free(degs);
free(angles);
free(node_distances);
free(final);
free(tempcsm);
```

```
}
```

File: *abf.h*

The following header file, *abf.h*, contains the declarations of the functions used to overload the operators to be used by the DPSAcomplex data type. It is included in all source files used for this thesis.

```
/*header file with declarations*/
#ifndef _ABF_H_
#define _ABF_H_

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>

typedef struct
{
  double real;
  double imag;
} DPSAcomplex;

void printmatrix(DPSAcomplex **,int);
int window(double *weight, int length_of_window);
int steered_time(double *out_tj, double center1, double center2, double
*degs);
void svdcomplex(DPSAcomplex **a, int m, int n, DPSAcomplex w[],
DPSAcomplex **v);

DPSAcomplex DPSA_Complex_multiply(DPSAcomplex c1, DPSAcomplex c2);
DPSAcomplex DPSA_Complex_divide(DPSAcomplex c1, DPSAcomplex c2);
DPSAcomplex DPSA_Complex_divide2(DPSAcomplex c1, DPSAcomplex c2);
int DPSA_Complex_colvect_times_rowvect(DPSAcomplex **result, int rows1,
  DPSAcomplex *vect1, int columns2, DPSAcomplex *vect2, int
conj_vect2);
DPSAcomplex DPSA_Complex_conj(DPSAcomplex c);
DPSAcomplex DPSA_Complex_sqrt(DPSAcomplex c);
DPSAcomplex DPSA_Complex_etothej(double arg);
double DPSA_Complex_abs(DPSAcomplex c);
double DPSA_Complex_abs_nosqrt(DPSAcomplex c);
int DPSA_Complex_matmult(DPSAcomplex **result, int rows1, int columns1,
DPSAcomplex **mat1, int rows2, int columns2, DPSAcomplex **mat2);
int DPSA_Complex_colvect_times_rowvect(DPSAcomplex **result, int rows1,
DPSAcomplex *vect1, int columns2, DPSAcomplex *vect2, int conj_vect2);
int DPSA_Complex_rowvect_times_colvect(DPSAcomplex *result, int
columns1, DPSAcomplex *vect1, int rows2, DPSAcomplex *vect2, int
conj_vect1);
int DPSA_Complex_conj_trans(DPSAcomplex **input, DPSAcomplex **result,
```

```
long rows, long cols);
int DPSA_Nonoptimized_inplacefft(DPSAcomplex *xforms, int nn, int
isign);

ostream &operator<<(ostream &output, const DPSAcomplex &c1);
DPSAcomplex operator-(DPSAcomplex c1, DPSAcomplex c2);
DPSAcomplex operator-(DPSAcomplex c1);
DPSAcomplex operator+(DPSAcomplex c1, DPSAcomplex c2);
DPSAcomplex operator*(DPSAcomplex c1, DPSAcomplex c2);
DPSAcomplex operator*(DPSAcomplex c1, double val);
DPSAcomplex operator*(DPSAcomplex c1, float val);
DPSAcomplex &operator++(DPSAcomplex &c1);
DPSAcomplex operator/(DPSAcomplex c1, int val);
DPSAcomplex operator/(DPSAcomplex c1, DPSAcomplex c2);
DPSAcomplex operator!(DPSAcomplex &c1);
DPSAcomplex &operator*=(DPSAcomplex &c1, double val);
DPSAcomplex &operator*=(DPSAcomplex &c1, DPSAcomplex c2);
DPSAcomplex &operator/=(DPSAcomplex &c1, DPSAcomplex c2);
DPSAcomplex &operator/=(DPSAcomplex &c1, double val);
DPSAcomplex &operator+=(DPSAcomplex &c1, DPSAcomplex c2);
DPSAcomplex &operator+=(DPSAcomplex &c1, double val);
DPSAcomplex &operator-=(DPSAcomplex &c1, double val);
DPSAcomplex &operator-=(DPSAcomplex &c1, DPSAcomplex c2);
DPSAcomplex operator/(DPSAcomplex c1, DPSAcomplex c2);
int operator==(DPSAcomplex c1, DPSAcomplex c2);
int operator>=(DPSAcomplex c1, DPSAcomplex c2);
int operator<=(DPSAcomplex c1, DPSAcomplex c2);
int operator>(DPSAcomplex c1, DPSAcomplex c2);
int operator<(DPSAcomplex c1, DPSAcomplex c2);
DPSAcomplex operator/(double val, DPSAcomplex c2);
DPSAcomplex operator+(double val, DPSAcomplex c2);
DPSAcomplex operator*(double val, DPSAcomplex c1);
int DPSA_Complex_rowvect_times_matrix(DPSAcomplex *result, DPSAcomplex
*vect1, int columns1, DPSAcomplex **c1, int columns2, int conj1);
void DPSA_Complex_separate(DPSAcomplex **c, int rows, int cols, double
**a, double **b);
void reverse(char s[]);
void itoa(int n, char s[]);
#endif
```

File: *abf.c*

The file, *abf.c*, contains the definitions, and other functions, for the overloaded operators to handle the DPSAcomplex data type.

```
/* MVDR library file
   Jesus Garcia
   High Performance Computing and Simulation Research Lab
```

```
   garcia@hcs.ufl.edu
   12/98
   Filename: ~garcia/dpsa/c/utils/abf.c
*/

#include <abf.h>
#include <setup.h>
#include <math.h>
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

void reverse(char s[])
{
  int c, i, j;

  for (i=0, j=strlen(s)-1;i<j;i++,j--) {
    c=s[i];
    s[i]=s[j];
    s[j] = c;
  }
}

void itoa(int n, char s[])
{
  int i, sign;
  if ((sign=n)<0) n = -n;
  i = 0;
  do {
       s[i++] = n%10 + '0';
    } while ((n/=10) > 0);
  if (sign < 0)
    s[i++] = '-';
    s[i] = '\0';
  reverse(s);
}


void printmatrix(DPSAcomplex **a, int n)
{
  for (int row = 0;row<n;row++) {
    for (int col = 0;col<n;col++)
      cout << a[row][col]<<" ";
    cout <<endl;
  }
}



/* ------------------------------------------------- */
```

```
/* -- Some of below part came from Jeff Markwell's source code -- */


ostream &operator<<(ostream &output, const DPSAcomplex &c1)
{
  output.precision(6);
  output <<setw(11)<< c1.real <<" + "<<setw(11)<<c1.imag<<" i ";
  return output;
}

DPSAcomplex DPSA_Complex_multiply(DPSAcomplex c1, DPSAcomplex c2)
{
  DPSAcomplex r;

  r.real = c1.real*c2.real - c1.imag*c2.imag;
  r.imag = c1.real*c2.imag + c1.imag*c2.real;

  return r;
}

DPSAcomplex DPSA_Complex_divide(DPSAcomplex c1, DPSAcomplex c2)
{
  DPSAcomplex r;

  r.imag = (c1.imag-c2.imag*c1.real/c2.real)/c2.real
    /(1+c2.imag*c2.imag/c2.real/c2.real);
  r.real=(c1.real+c2.imag*r.imag)/c2.real;

  return r;
}

DPSAcomplex operator-(DPSAcomplex c1, DPSAcomplex c2)
{
  DPSAcomplex result;
  result.real=c1.real-c2.real;
  result.imag=c1.imag-c2.imag;
  return result;
}

DPSAcomplex operator-(DPSAcomplex c1)
{
  DPSAcomplex result;
  result.real = -c1.real;
  result.imag = -c1.imag;
  return result;
}

DPSAcomplex operator+(DPSAcomplex c1, DPSAcomplex c2)
{
  DPSAcomplex result;
  result.real=c1.real+c2.real;
  result.imag=c1.imag+c2.imag;
  return result;
}
```

```
DPSAcomplex operator+(double val, DPSAcomplex c2)
{
  DPSAcomplex result;
  result.real = c2.real + val;
  result.imag = c2.imag;
  return result;
}

DPSAcomplex operator*(DPSAcomplex c1, DPSAcomplex c2)
{
  DPSAcomplex result;
  result.real = c1.real*c2.real - c1.imag*c2.imag;
  result.imag = c1.real*c2.imag + c1.imag*c2.real;
  return result;
}

DPSAcomplex operator*(DPSAcomplex c1, double val)
{
  DPSAcomplex result;
  result.real = c1.real*val;
  result.imag = c1.imag*val;
  return result;
}

DPSAcomplex operator*(double val, DPSAcomplex c1)
{
  DPSAcomplex result;
  result.real = c1.real*val;
  result.imag = c1.imag*val;
  return result;
}

DPSAcomplex &operator++(DPSAcomplex &c1)
{
  c1.real = c1.real + 1;
  c1.imag = c1.imag;
  return c1;
}


DPSAcomplex operator*(DPSAcomplex c1, float val)
{
  DPSAcomplex result;
  result.real = c1.real*val;
  result.imag = c1.imag*val;
  return result;
}

DPSAcomplex operator/(DPSAcomplex c1, int val)
{
  DPSAcomplex result;
  result.real = c1.real/val;
  result.imag = c1.imag/val;
```

```
  return result;
}

DPSAcomplex &operator/=(DPSAcomplex &c1, double val)
{
  c1.real = c1.real/val;
  c1.imag = c1.imag/val;
  return c1;
}

DPSAcomplex operator/(double val, DPSAcomplex c2)
{
  DPSAcomplex temp, result;
  temp  = DPSA_Complex_conj(c2);
  result = (temp * val)/(c2 * temp);
  return result;
}

int operator==(DPSAcomplex c1, DPSAcomplex c2)
{
  if (c1.real == c2.real && c1.imag == c2.imag)
    return 1;

  return 0;
}

int operator>=(DPSAcomplex c1, DPSAcomplex c2)
{
  if (DPSA_Complex_abs(c1) >= DPSA_Complex_abs(c2))
    return 1;

  return 0;
}

int operator>(DPSAcomplex c1, DPSAcomplex c2)
{
  if (DPSA_Complex_abs(c1) > DPSA_Complex_abs(c2))
    return 1;

  return 0;
}
int operator<(DPSAcomplex c1, DPSAcomplex c2)
{
  if (DPSA_Complex_abs(c1) < DPSA_Complex_abs(c2))
    return 1;

  return 0;
}
int operator<=(DPSAcomplex c1, DPSAcomplex c2)
{
  if (DPSA_Complex_abs(c1) <= DPSA_Complex_abs(c2))
    return 1;

  return 0;
```

```
}

DPSAcomplex &operator*=(DPSAcomplex &c1, double val)
{
  c1.real=c1.real*val;
  c1.imag=c1.imag*val;
  return c1;
}

DPSAcomplex &operator*=(DPSAcomplex &c1, DPSAcomplex c2)
{
  DPSAcomplex result;
  result.real = c1.real*c2.real - c1.imag*c2.imag;
  result.imag = c1.real*c2.imag + c1.imag*c2.real;
  c1=result;
  return c1;
}

DPSAcomplex &operator/=(DPSAcomplex &c1, DPSAcomplex c2)
{ // see page 14 of my notes
  c1.imag = (c1.imag-c2.imag*c1.real/c2.real)/c2.real
    /(1+c2.imag*c2.imag/c2.real/c2.real);
  c1.real=(c1.real+c2.imag*c1.imag)/c2.real; // note: this is supposed
                                             // to use the modified
c1.imag
  return c1;
}


DPSAcomplex &operator+=(DPSAcomplex &c1, DPSAcomplex c2)
{
  c1.real=c1.real+c2.real;
  c1.imag=c1.imag+c2.imag;
  return c1;
}

DPSAcomplex &operator+=(DPSAcomplex &c1, double val)
{
  c1.real=c1.real+val;
  return c1;
}

DPSAcomplex &operator-=(DPSAcomplex &c1, double val)
{
  c1.real -= val;
  return c1;
}


DPSAcomplex &operator-=(DPSAcomplex &c1, DPSAcomplex c2)
{
  c1.real=c1.real-c2.real;
  c1.imag=c1.imag-c2.imag;
  return c1;
```

```
}

DPSAcomplex operator/(DPSAcomplex c1, DPSAcomplex c2)
{
  DPSAcomplex c;
  double r, den;
  if (fabs(c2.real) >= fabs(c2.imag)) {
    r = c2.imag/c2.real;
    den = c2.real + r*c2.imag;
    c.real = (c1.real + r*c1.imag)/den;
    c.imag = (c1.imag - r*c1.real)/den;
  } else {
    r=c2.real/c2.imag;
    den = c2.imag + r*c2.real;
    c.real = (c1.real*r + c1.imag)/den;
    c.imag = (c1.imag*r - c1.real)/den;
  }
  return c;
}

DPSAcomplex operator!(DPSAcomplex &c1)
{
  DPSAcomplex result;
  result.real = c1.real;
  result.imag = -c1.imag;
  return result;
}

double DPSA_Complex_abs(DPSAcomplex c)
{
  double x,y,ans,temp;
  x = (double)fabs(c.real);
  y = (double)fabs(c.imag);
  if (x == 0.0) ans = y;
  else if (y == 0.0)
    ans = x;
  else if (x > y) {
    temp  = y/x;
    ans=x*sqrt(1.0 + temp*temp);
  } else {
    temp = x/y;
    ans=y*sqrt(1.0 + temp*temp);
  }
  return ans;
}

DPSAcomplex DPSA_Complex_divide2(DPSAcomplex c1, DPSAcomplex c2)
{
  DPSAcomplex r;
  double abs1, abs2;
  double arg1, arg2;
  abs1 = DPSA_Complex_abs(c1);
  arg1 = atan(c1.imag/c1.real);
  arg2 = atan(c2.imag/c2.real);
```

```
  abs2 = DPSA_Complex_abs(c2);
  r.real = (abs1/abs2)*cos(arg1 - arg2);
  r.imag = (abs1/abs2)*sin(arg1 - arg2);

  return r;
}
double DPSA_Complex_abs_nosqrt(DPSAcomplex c)
{
  double val;

  val = c.real*c.real + c.imag*c.imag;

  return val;
}

DPSAcomplex DPSA_Complex_conj(DPSAcomplex c)
{
  DPSAcomplex r;

  r.real = c.real;
  r.imag = -c.imag;

  return r;
}

int DPSA_Complex_conj_trans(DPSAcomplex **input, DPSAcomplex **result,
long rows, long cols)
{
  for (int x=0;x<rows;x++)
    for(int y=0;y<cols;y++)
      result[y][x] = DPSA_Complex_conj(input[x][y]);

  return 0;
}

DPSAcomplex DPSA_Complex_etothej(double arg)
{
  DPSAcomplex r;

  r.real = cos(arg);
  r.imag = sin(arg);

  return r;
}

int DPSA_Complex_matmult(DPSAcomplex **result, int rows1, int columns1,
  DPSAcomplex **mat1, int rows2, int columns2, DPSAcomplex **mat2)
{
  int row, column, tempidx;
  DPSAcomplex multiplication;

  if(columns1!=rows2)
    return (-1);
```

```
  for(row=0; row<rows1; row++)
  {
    for(column=0; column<columns2; column++)
    {
      result[row][column].real = 0;
      result[row][column].imag = 0;
      for(tempidx=0; tempidx<columns1; tempidx++)
      {
        multiplication = mat1[row][tempidx] * mat2[tempidx][column];
        result[row][column].real += multiplication.real;
        result[row][column].imag += multiplication.imag;
      }
    }
  }
  return 0;
}

int DPSA_Complex_colvect_times_rowvect(DPSAcomplex **result, int rows1,
  DPSAcomplex *vect1, int columns2, DPSAcomplex *vect2, int conj_vect2)
{
  int row, column;

  for(row=0; row<rows1; row++)
    for(column=0; column<columns2; column++)
    {
      if(conj_vect2)
        result[row][column] = DPSA_Complex_multiply(vect1[row],
          DPSA_Complex_conj(vect2[column]));
      else
        result[row][column] = DPSA_Complex_multiply(vect1[row],
vect2[column]);
    }
  return 0;
}

int DPSA_Complex_rowvect_times_colvect(DPSAcomplex *result,
  int columns1, DPSAcomplex *vect1, int rows2, DPSAcomplex *vect2, int
conj_vect1)
{
  DPSAcomplex temp, *temp2;
  temp2 = (DPSAcomplex*)malloc(columns1*sizeof(DPSAcomplex));

  int idx, i;

  if(columns1!=rows2)
    return (-1);

  result->real=0;
  result->imag=0;
  if (conj_vect1) for (i=0;i<columns1;i++) temp2[i] = !vect1[i];
  else  for (i=0;i<columns1;i++) temp2[i] = vect1[i];

    for(idx=0; idx<rows2; idx++)
  {
```

```
    temp = DPSA_Complex_multiply(temp2[idx], vect2[idx]);
    result->real += temp.real;
    result->imag += temp.imag;
  }
  free(temp2);
    return 0;
}

int DPSA_Complex_rowvect_times_matrix(DPSAcomplex *result, DPSAcomplex
*vect1, int columns1, DPSAcomplex **c1, int columns2, int conj1)
{
  int i, j;
  DPSAcomplex sum, *temp;
  temp = (DPSAcomplex*)malloc(columns1*sizeof(DPSAcomplex));

  if (conj1) for (i=0;i<columns1;i++) temp[i] = !vect1[i];
  else  for (i=0;i<columns1;i++) temp[i] = vect1[i];

  for (i=0;i<columns2;i++) {
    sum.real = 0;
    sum.imag = 0;
    for (j=0;j<columns1;j++) {
      sum += temp[j]*c1[j][i];
    }
    result[i] = sum;
  }
  free(temp);
}

DPSAcomplex DPSA_Complex_sqrt(DPSAcomplex z)
{
  DPSAcomplex c;
  double x, y, w, r;
  if ((z.real == 0.0) && (z.imag = 0.0)) {
    c.real = 0.0;
    c.imag = 0.0;
    return c;
  } else {
    x = fabs(z.real);
    y = fabs(z.imag);
    if (x >= y) {
      r=y/x;
      w=sqrt(x)*sqrt(0.5*(1.0+sqrt(1.0+r*r)));
    } else {
      r=x/y;
      w=sqrt(y)*sqrt(0.5*(r+sqrt(1.0+r*r)));
    }
    if (z.real >= 0.0) {
      c.real = w;
      c.imag = z.imag/(2.0 * w);
    } else {
      c.imag = (z.imag >= 0) ? w : -w;
      c.real = z.imag/(2.0*c.imag);
    }
```

```
    return c;
  }
}
```

APPENDIX D
MATLAB SOURCE CODE

File: *cbf_in.m*

The file, *cbf_in.m*, was written by Ken Kim to generate one data vector for one frequency bin. It was used to generate data used by the parallel and sequential programs. It was also used to generate the data used by all MATLAB programs.

```
function [f_waves] = cbf_in(M,D,C,F,angles,sn_ratio)
% [f_waves] = cbf_in(M,D,C,F,angles,sn_ratio)
% Make sonar data (M-by-1) for the angle (in frequency domain)
% f_waves : frequency data (M-by-1)
% M : number of nodes
% D : spacing between nodes (feet)
% C : speed of sound (feet/sec)
% F : desired frequency of data (Hz)
% angles : incoming angles in vector form (degree)
% sn_ratio : Signal to noise ratio in decibel (dB)
%
% ex) f_waves = cbf_in(10,10,1500,60,[0 45 60],10);
% generate 10-by-1 vector with 60 Hz frequency, incomming angles are 0,
45, 60 superimposed
% SNR is 10dB
%
% Ken Kim : kim@hcs.ufl.edu
% High-performance Computing and Simulation Research Lab
% 2/8/99

if nargin < 6, error('Not enough arguments.'), end

%clear;
%M = 10;
%D = 10;
%C = 1500;
%F = 60;
%fs = 1000;
%numsamp = 256;
%angles = [0 45 60];
%sn_ratio = 10;
%fo = 30;
```

```
%fmin = 0.2;
%fmax = 3;
%fnum = 128;
%maxmag = 2;

num_sig = max(size(angles));

if num_sig > M
   error('Number of signals cannot exceed the number of nodes !');
end

% SNR = 10*log(P_signal/P_noise)
% P_noise = P_signal/(exp(SNR/10))

j = sqrt(-1);
pw_sig = 1;
pw_noi = 1/(10^(sn_ratio/10));
Fx = randn(num_sig,1)*sqrt(pw_sig/(2*num_sig));
Fy = randn(num_sig,1)*sqrt(pw_sig/(2*num_sig));
S = Fx+j*Fy;
Nx = randn(M,1)*sqrt(pw_noi/2);
Ny = randn(M,1)*sqrt(pw_noi/2);
N = Nx+j*Ny;
DOA = pi*angles/180;
Td = D*sin(DOA)/C;

MM = (0:1:M-1);
M_pos = MM.*D;
c1 = 0;

for i = 1:M,
      c1    = c1+M_pos(i);
end

c1 = c1/M;
rel_pos = M_pos(1:1:M)-c1;
rel_pos = rel_pos/D;
A = exp(-j*rel_pos'*Td*2*pi*F);
f_waves = A*S+N;
```

File: *cbf_vt.m*

The file, *cbf_vt.m*, was also written by Ken Kim to generate the replica vectors used for conventional and MVDR beamforming.

```
function [vt,degs] = cbf_vt(M,D,C,F,ndeg)
%
% Generate steering vectors for Conventional beamforming
```

```
% Single phase center (in the middle of the array)
%
% [vt,degs] = cbf_vt(M,D,C,F,ndeg)
% Calculate steering vectors and degree steered
% vt : steering vector (M-by-ndeg) in freq. domain
% N : vector of bad nodes
% M : number of nodes
% D : spacing between nodes (feet)
% C : speed of sound (feet/sec)
% F : processing freq. (Hz)
% ndeg : number of degree (odd number)
%
% Ken Kim : kim@hcs.ufl.edu
% High-performance Computing and Simulation Research Lab
% 2/8/99

if nargin < 5, error('Not enough arguments'), end

mm = (0:1:M-1);
m_pos = mm.*D;
c1 = 0;

for i = 1:M,
      c1    = c1+m_pos(i);
end

c1 = c1/M;
rel1 = m_pos(1:1:M)-c1;

if rem(ndeg,2) == 1,
      temp1 = ndeg-1;
      temp2 = temp1/2;
      temp3 = 1/temp2;
      incdeg = (-1:temp3:1);
else
      error('Number of degree should be odd number');
end

degs = 180.*asin(incdeg)./pi;
delay = (rel1'*incdeg./C);
j = sqrt(-1);
vt = exp(-j*2*pi*F*delay);
```

File: *abf.m*

The file, *abf.m*, was used to compare MVDR and conventional beamformers. It takes as arguments the number of nodes and the angles of incoming signals and generates

beampatterns and final beamforming plots for both types of beamformers.

```
function abf(M, in_angle)

%function abf(M, in_angle);


% M : number of nodes
% in_angle : incoming angles of signal

%clear
J=0;
F     = 30;             % Base freq. processing band 0.2F-3.0F
T     = 1/F;                  % T_none
C     = 1500;                 % Speed of sound(m/s)
L     = C/F;                  % wave length(m)
D     = 0.45*L;         % Spacing between nodes(m)
FS    = 12*F;                 % Sampling frequency(Hz)
wave_number = (2*pi)/L;
look = 0;
numsamp = 10000;
alpha = 0.9;
epsilon = .7;
nangle      = 91;             % Number of angle steered
      replicas = zeros(1,M);
      [replicas, degs] = cbf_vt(M, D, C, F, nangle);
      %replicas = replicas./M*2;
      %normal = conj(replicas).*replicas
      R = zeros(M, M);
      Rav = zeros(M, M);
      w_good_mvdr = zeros(M, nangle);
      power_good_mvdr = zeros(1,nangle);
      power_cbf = zeros(1,nangle);
      for i = 1:numsamp,
            x = cbf_in(M, D, C, F, in_angle, 25);
            %     R = x*x' + epsilon*eye(M);
                  R = R + x*x';
            %     Rav = (alpha)*Rav + (1-alpha)*R;
      end;
      R = R/numsamp;

      invR = inv(R);

      for k = 1:nangle,
            power_good_mvdr(k) = 1/(replicas(:,k)'*invR*replicas(:,k));
            w_good_mvdr(:, k) =
invR*replicas(:,k)/(replicas(:,k)'*invR*replicas(:,k));
                  power_cbf(k) = replicas(:,k)'*R*replicas(:,k);
            %ws = sum(conj(w_good_mvdr(:,k)).*replicas(:,k))
      end;

figure(1)
plot(degs, 10*log10(abs(power_good_mvdr)));
```

```
axis([min(degs) max(degs) min(10*log10(real(power_good_mvdr)))
max(10*log10(real(power_good_mvdr)))]);
%plot(degs, real(power_good_mvdr));
grid on;
ylabel('Power (dB)');
xlabel('look direction (degrees)');
title('MVDR Power Plot');

figure(4)
plot(degs, 10*log10(abs(power_cbf)));
axis([min(degs) max(degs) min(10*log10(real(power_cbf)))
max(10*log10(real(power_cbf)))]);
%plot(degs, real(power_cbf));
grid on;
ylabel('Power (dB)');
xlabel('look direction (degrees)');
title('CBF Power Plot');


figure(2)
[res2, rads1] = beam(w_good_mvdr, look, degs);
res2 = res2./max(res2);
%plot(degs, 10*log10(res2));
plot(degs, res2);
axis([min(degs) max(degs) min(res2) max(res2)]);
grid on
xlabel('Bearing (degrees)');
ylabel('Beam power');
title('Beampattern for MVDR');
areares2 =  sum(res2)
%res2


figure(3)
[rescbf, radscbf] = beam(replicas, look, degs);
%plot(degs, 10*log10(rescbf));
plot(degs, rescbf);
axis([min(degs) max(degs) min(rescbf) max(rescbf)]);
grid on
xlabel('Bearing (degrees)');
ylabel('Beam power');
title('Beampattern for CBF');


figure(6)
polar(radscbf, rescbf);
title('Polar plot of CBF beampattern');
```

REFERENCES

1. A. O. Steinhardt and B. D. Van Veen, "Adaptive Beamforming," International J. of Adaptive Control and Signal Processing, **3**, 253-281 (1989).

2. L. Castedo and A. R. Figueiras-Vidal, "An Adaptive Beamforming Technique Based on Cyclostationary Signal Properties," IEEE Trans. on Signal Processing, **43** (7), 1637-1650 (1995).

3. M. Zhang and M. H. Er, "Robust Adaptive Beamforming for Broadband Arrays," Circuits, Systems, and Signal Processing, **16** (2), 207-216 (1997).

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), 1481-1494 (1989).

5. H. Cox, R. M. Zeskind, and M. M. Owen, "Robust Adaptive Beamforming," IEEE Trans. On Acoustics, Speech, and Signal Processing, **35** (10), 1365-1377 (1987).

6. A. D. George, J. Markwell, and R. Fogarty, "Real-time sonar beamforming on high-performance distributed computers," submitted to Parallel Computing, Aug. 1998.

7. A. D. George and K. Kim, "Parallel Algorithms for Split-Aperture Conventional Beamforming," to appear in J. Computational Acoustics, (1999).

8. 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, **1770**, 86-97 (1992).

9. M. Moonen, "Systolic MVDR Beamforming with Inverse Updating," IEE Proc.-F, Radar and Signal Processing, **140** (3), 175-178 (1993).

10. J. G. McWhirter and T. J. Shepherd, "A Systolic Array for Linearly Constrained Least Squares Problems," Proc. SPIE, Advanced Algorithms and Architectures for Signal Processing, **696**, 80-87 (1986).

11. F. Vanpoucke and M. Moonen, "Systolic Robust Adaptive Beamforming with an Adjustable Constraint," IEEE Trans. On Aerospace and Electronic Systems, **31** (2), 658-669 (1995).

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), 367-384 (1994).

13. S. Chang and C. Huang, "An application of systolic spatial processing technique in adaptive beamforming," J. Acoust. Soc. Am., **97** (2), 1113-1118 (1995).

14. M.Y. Chern and T. Murata, "A Fast Algorithm for Concurrent LU Decomposition and Matrix Inversion," Proceedings of the International Conference on Parallel Processing, 79-86 (1983).

15. D. H. Bailey and H. R. P. Ferguson, "A Strassen-Newton Algorithm for High-Speed Parallelizable Matrix Inversion," Proceedings Supercomputing, 419-424 (1988).

16. D. S. Wise, "Parallel Decomposition of Matrix Inversion using Quadtrees," Proceedings of the International Conference on Parallel Processing, 92-99 (1986).

17. V. Pan and J. Reif, "Fast and Efficient Parallel Solution of Dense Linear Systems," Computers Math. Applic., **17** (11), 1481-1491 (1989).

18. B. Codenotti and M. Leoncini, "Parallelism and Fast Solution of Linear Systems," Computers Math. Applic., **19** (10), 1-18 (1990).

19. K. K. Lau, M. J. Kumar, and S. Venkatesh, "Parallel Matrix Inversion Techniques," Proc. of the IEEE 2nd International Conference on Algorithms and Architectures for Parallel Processing, 515-521, (1996).

20. Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," Technical Report CS-94-230, Computer Science Dept., Univ. of Tennessee, April 1994.

21. K.A. Robbins and S. Robbins, *Practical Unix Programming: A Guide to Concurrency, Communication, and Multithreading*, Prentice Hall, (1996).

22. P.M. Clarkson, *Optimal and Adaptive Signal Processing*, CRC Press, (1993).

23. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*, 2nd ed., Cambridge University Press, (1992).

24. G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed., Johns Hopkins University Press, (1996).

25. R.O. Schmidt, "Multiple Emitter Location and Signal Parameter Estimation," IEEE Trans. on Antennas and Propagation, **34** (3), 276-280 (1986).

BIOGRAPHICAL SKETCH

Jesus Luis Garcia was born on December 25, 1975, in Cardenas, Cuba, where he lived until he was four years old. His parents, Ruben and Blanca, took advantage of the opportunity to escape communism and moved the family, which includes little brother Jorge, to Spain before finally settling in sunny Miami, FL.

Jesus attended South Miami Senior High where he graduated magna cum laude in 1993. He then made the best decision of his life and continued his education at the University of Florida.

Jesus studied hard during the academic school year and obtained employment during most of the summers. He held jobs at Incredible Universe and WLTV Channel 23. Jesus's first major at UF was industrial engineering but while taking Physics II he realized that electrical engineering was his true calling and quickly changed majors.

In Gainesville, Jesus had the good fortune of finding a job at Z-Systems, where he learned about DSP and gained valuable technical experience. After graduating with his BSEE degree, Jesus decided to attend graduate school at UF. In fall of 1997 he joined the HCS Lab where he conducted the research that is the focus of this thesis.

Jesus's career search was an extensive one but he finally secured a position as a Rotational Engineer with Intel Corporation. Jesus is excited about his move to Folsom, CA, where he will begin his career at Intel.