Copyright

by

Gregory Eugene Allen

1998

Real-Time Sonar Beamforming on a Symmetric Multiprocessing UNIX Workstation Using Process Networks and POSIX Pthreads

by

Gregory Eugene Allen, B.S.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

August 1998

Real-Time Sonar Beamforming on a Symmetric Multiprocessing UNIX Workstation Using Process Networks and POSIX Pthreads

Approved by
Supervising Committee:

____
Brian L. Evans

____
Craig M. Chase

Dedication

I would like to dedicate this report to my wife, Dara Marie, and to our 14-month-old daughter, Sabrina Fair. Thank you for providing me with the quiet time I needed to complete this report. Once again, I can join in the chase.

And also to my parents and extended family, for their encouragement, and for always stressing the value of education.

Acknowledgements

I would like to thank the talented people in the Advanced Sonar Group at Applied Research Laboratories, The University of Texas at Austin, for their years of guidance and support, and for providing me the time and equipment to produce this research. I would especially like to thank Terry Brudner, Mike DeSimone, Steve Morrissette, and Jeff Napper for their insight and assistance.

August 10th, 1998

Abstract

Real-Time Sonar Beamforming on a Symmetric Multiprocessing UNIX Workstation Using Process Networks and POSIX Pthreads

Gregory Eugene Allen, M.S.E.

The University of Texas at Austin, 1998

Supervisor: Brian L. Evans

Traditionally, expensive custom hardware has been required to implement data-intensive sonar beamforming algorithms in real-time. We develop a sonar beamformer in software by merging the following recent technologies: (1) symmetric multiprocessing on Unix workstations, (2) lightweight POSIX threads, and (3) the Process Network model of computation. We find that it is feasible for a 4-GFLOP digital interpolation Process Network beamformer to run in real-time on a Sun workstation with 16 UltraSPARC-II processors running at 336 MHz. The workstation beamformer significantly reduces cost and development time over an equivalent custom hardware beamformer.

Table of Contents

List of Tables ix

List of Figures x

Chapter 1: Introduction 1

Chapter 2: Beamforming 6

2.1: Analog Beamforming 6

2.2: Digital Beamforming 8

2.3: Digital Interpolation Beamforming 9

2.3.1: Digital Interpolation 10

2.3.2: A Sparse FIR Filter Model 11

2.4: Summary 12

Chapter 3: Process Networks 14

3.1: Kahn Process Networks 14

3.2: Bounded Scheduling of Process Networks 16

3.3: Karp and Miller Computation Graphs 18

3.4: Summary 19

Chapter 4: Process Network Implementation 21

4.1: The ThresholdQueue Template Class 22

4.2: Process Nodes 26

4.3: Communication Channels 28

4.4: Constructing a Process Network Program 30

4.5: Summary 31

Chapter 5: Beamformer Implementation 32

5.1: Horizontal Beamforming 33

5.1.1: Cache Utilization 35

5.1.2: Integration with Process Networks 36
5.2: Vertical Beamforming 37
5.3: Summary 39

Chapter 6: Results 40
6.1: Horizontal Beamformer Performance 40
6.2: Vertical Beamformer Performance 42
6.3: Process Network Beamformer Performance 44
6.5: Summary 47

Chapter 7: Conclusion 48

References 50

Vita 52

List of Tables

Table 6.1: Horizontal beamforming benchmark results. 41

Table 6.2: Vertical beamforming benchmark results. 43

Table 6.3: Process Network vs. thread-pool performance results. 45

Table 6.4: Process Network beamformer scalability. 46

List of Figures

Figure 1.1: A sample underwater environment.. 1

Figure 1.2: Ahead-looking horizontal beam coverage for a sonar system. Multiple sensor outputs are combined to form each beam................ 2

Figure 2.1: Analog beamformer with hydrophone array. 7

Figure 2.2: Projection of sensor elements from a semi-circular array. 7

Figure 2.3: Digital beamformer with digitizing sensor array. 8

Figure 2.4: Digital interpolation beamformer with digitizing sensor array. 10

Figure 2.5: The steps of digital interpolation. 11

Figure 2.6: Matrix operation to generate one beam set. 12

Figure 3.1: A simple process network program. 14

Figure 3.2: A strictly bounded process network. 17

Figure 4.1: Enqueuing data into a ThresholdQueue. 24

Figure 4.2: Dequeuing data from a ThresholdQueue... 24

Figure 4.3: ThresholdQueue implementation. 25

Figure 4.4: A partial declaration of PNNodeInput... 27

Figure 4.5: A partial declaration of PNNodeOutput. 27

Figure 4.6: Code for a node that copies data. 27

Figure 4.7: A partial declaration of PNThresholdQueue. 29

Figure 4.8: A sample process network program. 30

Figure 4.9: Implementation of the process network program. 31

Figure 5.1: A block diagram of the beamformer stages...................................... 32

Figure 5.2: Beamforming coefficients for one beam. 34

Figure 5.3: A horizontal beamformer node.. 36
Figure 5.4: Interleaving of the vertical beamformer output................................ 38

Chapter 1: Introduction

Sonar, the acoustic analog to radar, is a method for detecting and locating objects using acoustic waves. It is often employed in environments where acoustic wave propagation is superior to electromagnetic wave propagation, such as under water. Sonar can be used for navigation, to identify hazardous or hostile objects, and to map terrain features in the surrounding environment. Fig. 1.1 shows a sample underwater environment with a navigational hazard.

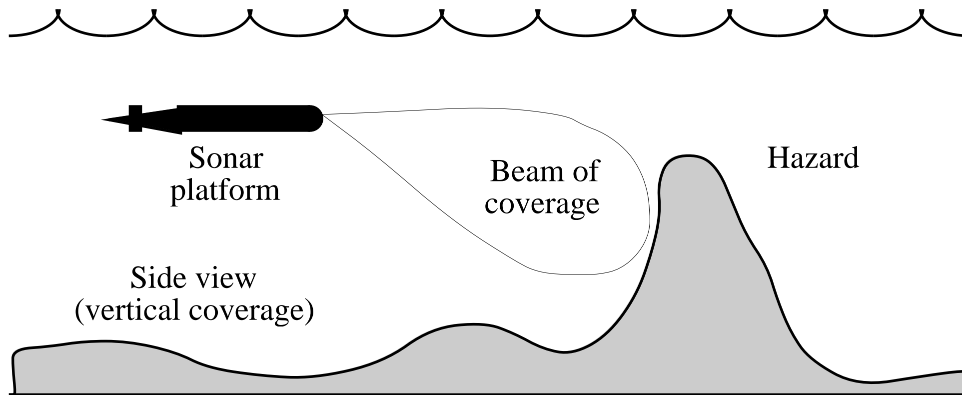

Figure 1.1: A sample underwater environment.

High-resolution sonars generally consist of an array of underwater sensors along with a *beamformer* [1, 10] to determine from which direction a sound is coming. The higher the resolution of a sonar system, the more accurately the location of an object can be determined. To achieve high resolution over a wide coverage area, a large number of beams may be formed. Fig. 1.2 illustrates a high resolution, multi-beam sonar with several narrow horizontal beams covering a wide horizontal sector.

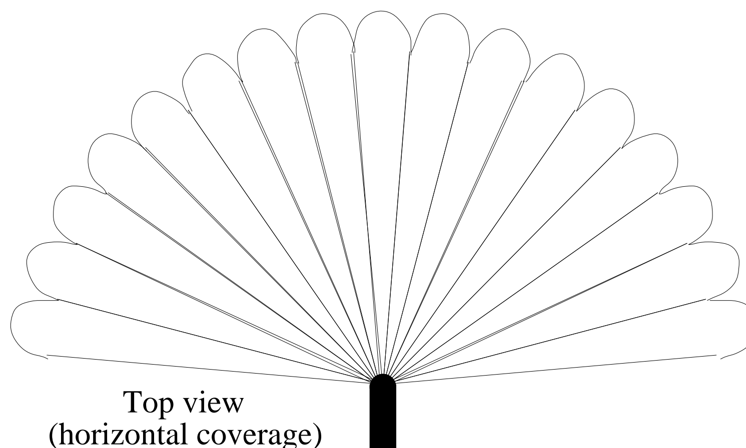

Figure 1.2: Ahead-looking horizontal beam coverage for a sonar system. Multiple sensor outputs are combined to form each beam.

The sensor element outputs must be combined to form these multiple narrow beams, each of which “looks” in a single direction and is insensitive to sound in neighboring directions. This combination must be performed with precise time delays and amplitude weighting applied to the sensor outputs. As a result, beamforming for high-resolution sonar systems is extremely computationally intensive.

Traditionally, custom hardware has been required to implement sonar beamforming algorithms in real-time. However, the performance of today's symmetric multiprocessing UNIX workstations makes it possible to implement these algorithms at a fraction of the development and manufacturing costs of a custom hardware solution.

Software development in a workstation environment is generally easier than in a custom embedded hardware environment due to the availability of

superior and more affordable development and debugging tools. In this implementation, the workstation is both the development platform and the target architecture. Now we can deploy the computer-aided design tools along with the design. Workstations also offer better portability, upgradability, and maintainability than custom hardware solutions. In order to facilitate implementation of large, computationally intensive systems, a reliable formal design methodology is needed for organizing and developing real-time multiprocessor software.

The Process Network [4, 5] model of computation captures the concurrency and parallelism in signal processing systems. The model represents a program in directed graph notation, where each node represents an independent process and each edge represents a one-way FIFO queue of data to be communicated. This model provides for correctness, and guarantees determinate execution of the program regardless of the scheduling algorithm used. Dynamic scheduling based on the availability of data allows execution in bounded memory [7]. This bounded memory Process Network model is well-suited for implementation using the thread model of concurrent programming.

The Portable Operating System Interface (POSIX) is a recent standard [11] with the goal of providing source-code portability across many UNIX platforms. Implementing the Process Network model with POSIX threads (Pthreads) gives a low-overhead, high-performance, scalable framework. Symmetric multiprocessing guarantees efficient utilization of multiple processors, as scheduling of Pthreads is dynamically handled by the operating system. The

POSIX standard optionally allows Pthreads to be scheduled with real-time priority.

The goal is to implement a high-resolution multi-fan three-dimensional digital interpolation beamformer, as described in Chapter 2, which runs at real-time on a Unix workstation. This is realized by performing a design space exploration of software beamforming implementations, modeled as a Process Network. We show that it is feasible for a 4-GFLOP multi-fan three-dimensional digital interpolation beamformer to run in real-time on a Sun workstation with 16 UltraSPARC-II processors running at 336 MHz (5.4 GFLOPS at 1 floating point operation per clock). This beamformer can combine the outputs of both vertical and horizontal sensors to image an underwater environment in three dimensions. The software beamformer reduces manufacturing costs, development costs, and development time by a factor of three, and volume and weight by a factor two, over an equivalent modern hardware beamformer.

Chapter 2 discusses beamforming techniques including the digital interpolation beamfoming algorithm. Modeling this algorithm using sparse finite impulse response (FIR) filters is also discussed. Chapter 3 explains the formal definition and properties of the Process Network model, and addresses operational semantics such as scheduling approaches. Chapter 4 covers the C++ implementation of the Process Network model, including low-overhead circular queues for processors (such as the UltraSPARC) that lack modulo addressing modes. Chapter 5 addresses the C++ implementation and optimization of a computationally intensive beamformer, using the Process Network model.

Chapter 6 compares the performance of batch-mode and Process Network realizations of digital interpolation beamformers in order to evaluate Process Network overhead. Chapter 7 concludes this report.

Chapter 2: Beamforming

A beamformer is a spatial filter that operates on the output of an array of sensors in order to determine from which direction a sound is coming. It enhances the amplitude of a coherent wavefront relative to background noise and directional interference. Time-domain beamforming is realized by delaying and summing the shaded outputs of an array of transducers. Shading is simply amplitude weighting which is done to improve the spatial response characteristics of the beams. The beamforming delays are matched to the anticipated propagation delays of a pressure field incident from a specific direction [1].

Section 2.1 describes simple analog time-delay beamforming, including determination of the beamforming time delays. Section 2.2 discusses a digital implementation of beamforming, and Section 2.3 describes the algorithm for digital interpolation beamforming.

2.1: ANALOG BEAMFORMING

Conceptually, time-domain beamforming in a single dimension is quite simple. For M sensors (transducers) each receiving a signal $x_m(t)$, the output of a single beam can be calculated by

$$b(t) = \overset{M}{\underset{m=1}{}} \quad {}_m x_m(t - \quad {}_m),$$

where ${}_m$ is the shading coefficient for the mth sensor, and ${}_m$ is the required time delay applied to the output of the mth sensor. Fig. 2.1 shows the block diagram for calculating a single beam using strictly analog methods. Note that analog multiplication, time delay, and summation is required.

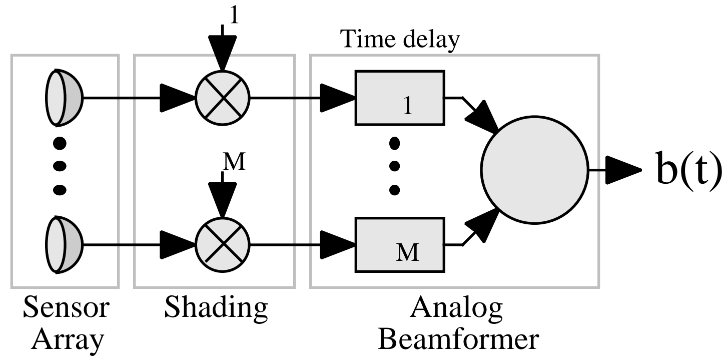

Figure 2.1: Analog beamformer with hydrophone array.

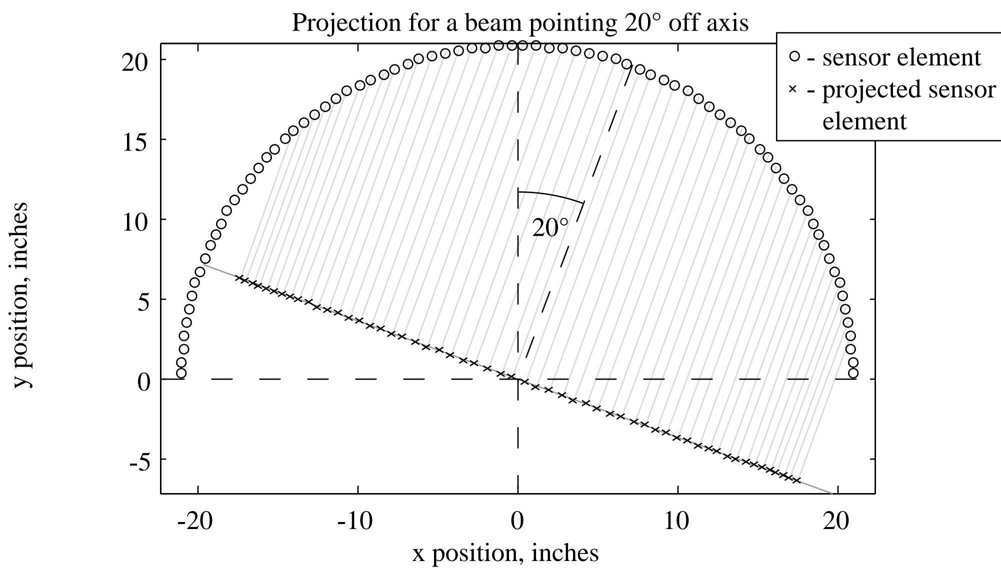

Figure 2.2: Projection of sensor elements from a semi-circular array.

The beamforming time delays are determined by geometrically projecting the elements of the sensor array onto a line that is perpendicular to the Maximum Response Angle for the desired beam. This is demonstrated in Fig. 2.2 with a semi-circular array of 80 elements, for a beam pointing 20° off axis.

The distance from each physical element location to the perpendicular line (divided by the speed of sound) is the necessary time delay for the corresponding element. Note that just over 50 of the elements have been projected, and the remaining elements have been left out. Although the remaining elements could be used in the calculation, their response in the direction of interest is relatively small for this geometry, and they would merely add noise. Leaving these elements out would also substantially reduce computation.

2.2: DIGITAL BEAMFORMING

Sampling the output of each sensor element in time (at t = n   ) creates a digital signal, which requires a digital beamformer implementation. These sets of discrete samples are now digitally delayed and summed in the beamforming operation. The delays must be quantized to increments of the input sampling period,   . Fig. 2.3 shows a block diagram.

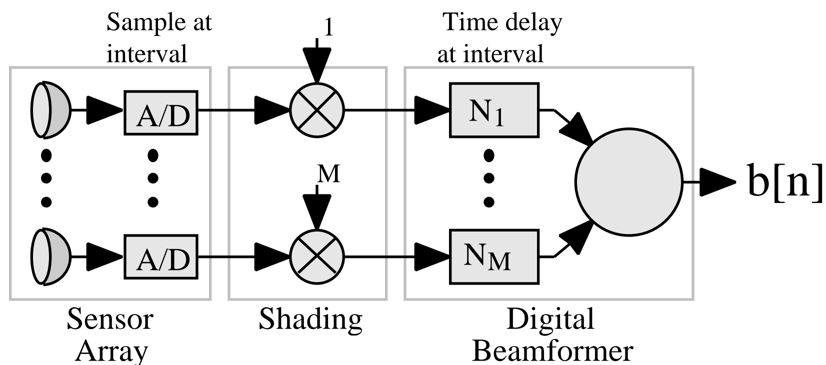

Figure 2.3: Digital beamformer with digitizing sensor array.

The sensor must be sampled at a rate much greater than the Nyquist rate to approximate the time delays required for beam steering [1]. To handle the high

bandwidth and large number of elements needed for high resolution sonar, the necessary sampling rate is quite large. These high sampling rates impose requirements on the analog-to-digital (A/D) converters and on the bandwidth of the cables which connect the A/D converters to the beamformer. Large amounts of memory are also required to handle the long delays associated with large arrays and high sample rates [2]. Because of the high implementation cost, digital interpolation beamformers are generally used instead.

2.3: DIGITAL INTERPOLATION BEAMFORMING

Digital interpolation beamforming is an efficient algorithm that utilizes hydrophone data sampled at just above the Nyquist rate. The desired time-delay quantization is achieved by digital interpolation of the sampled data. Additional computation is required to perform this interpolation, but it is performed efficiently using FIR digital filters.

This principle is a simple one, but with profound consequences. The expense of high frequency A/D converters and high bandwidth cables can now be shared with the digital signal processing hardware in order to optimize the overall system implementation [1].

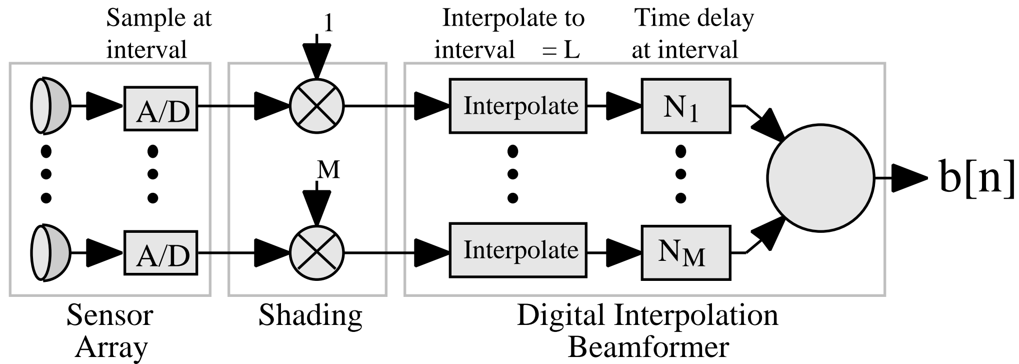

Figure 2.4: Digital interpolation beamformer with digitizing sensor array.

Fig. 2.4 shows a digital system with a digital interpolation beamformer. The sampling interval needed to satisfy the Nyquist criterion is . Digital interpolation is performed to the interval , where = L , and L is an integer larger than one. Now time delays are quantized to integer multiples of , i.e., $_m$ = N_m .

In a wideband sonar, the digital interpolation beamformer technique relaxes specifications on the A/D conversion rate and data transmission bandwidth at the expense of additional computation for digital interpolation. Beam degradation introduced by interpolation is controllable and quite small for an interpolation filter of modest design [1, 3]. Digital interpolation beamformers have been successfully working in the field for at least a decade.

2.3.1: Digital Interpolation

Digital interpolation is a two-step process: zero-insertion and then lowpass filtering [2, 12]. Since the filter and the summation are both linear, the filter can be placed either before or after the beamformer summation. The digital

interpolation beamformer in Fig. 2.4 performs the interpolation before the summation.

The first step of interpolation, zero-insertion, involves transforming the digital stream $x_m(n\ )$ by inserting L-1 zeros after each sample. The resulting stream $x_m(n\ )$ has L times more samples, and has had its sampling period reduced by a factor of L.

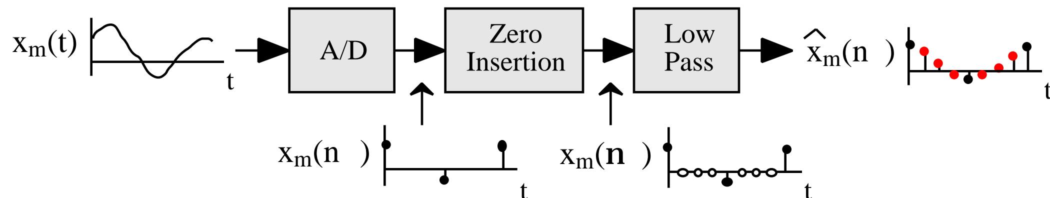

Figure 2.5: The steps of digital interpolation.

To complete the interpolation, an ideal digital lowpass filter with a cutoff frequency at = /L is required. Filtering the stream $x_m(n\ )$ yields the interpolated approximation to the input sampled at the interval . Since FIR filters are not ideal, error is introduced at the beamformer output. Increasing the number of filter coefficients reduces this error, so there is a tradeoff between accuracy and computational complexity. Fig. 2.5 shows the steps of digital interpolation, for L = 4.

2.3.2: A Sparse FIR Filter Model

Modeling the beamformer as a sparse FIR filter allows for a simple, concise organization of the algorithm. If multiple samples of the entire array are stored contiguously in memory, each beam output can be generated by an FIR filter of length K = (D+P-1)M, where D is the maximum sample delay due to the

array geometry, M is the total number of sensors in the array, and P is the number of points used to calculate each interpolation result. Although this can be an extremely long filter, most of the coefficients are zero. The number of non-zero coefficients, C = PS, where S is the number of sensors used to calculate each beam. The sparsity is 1-C/K, and is typically above 75%. Note that in this model, the digital interpolation lowpass filter from Section 2.3.1 is an FIR filter with an impulse response of length K = LP.

$$\underset{(1 \text{ by } K)}{\begin{bmatrix} \text{Incoming Data} \end{bmatrix}} \times \underset{(K \text{ by } B)}{\begin{bmatrix} \text{Beam} & & \text{Beam} \\ 1 & \cdots & B \\ \text{coefs} & & \text{coefs} \end{bmatrix}} = \underset{(1 \text{ by } B)}{\begin{bmatrix} \text{Beam Data} \\ \text{(1 sample)} \end{bmatrix}}$$

Figure 2.6: Matrix operation to generate one beam set.

For each sample of a beam's output, C multiply-accumulates (MACs) are required. When B beams are calculated, (BC) MACs must be executed. Fig. 2.6 shows the matrix operations necessary to calculate B beams from the input data stream. This algorithm has an extremely high degree of parallelism, which can be exploited by using the Process Network model of computation.

2.4: SUMMARY

A beamformer is a spatial filter that operates on the output of an array of sensors in order to image the environment. Time-domain beamforming is realized by delaying and summing the sensor outputs. For digital implementations, the desired time-delay resolution is generally much greater than that provided by the Nyquist rate, which is the minimum sampling frequency required to preserve the

significant frequency content of the data. In a digital interpolation beamformer, this higher resolution time delay is achieved by interpolation of data that was sampled at just above the Nyquist rate. To simplify the computation and storage requirements, this algorithm can be modeled as a sparse FIR filter.

Chapter 3: Process Networks

In the process network [4, 5] model of computation, concurrent processes are connected by unidirectional first-in, first-out (FIFO) queues to form a network. The model uses a directed graph notation, where each node represents a process and each edge represents a communication channel (queue). This model is natural for describing the streams of data samples in a signal processing system. Fig. 3.1 shows a simple process network program, in which processes A and B are connected by a communication channel, P.

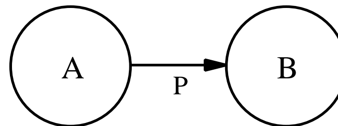

Figure 3.1: A simple process network program.

Section 3.1 explains the formal definition and properties of the Kahn Process Network model. Section 3.2 gives rules for executing Process Networks in bounded memory. Section 3.3 discusses computation graphs, which is a similar, but more restricted model.

3.1: Kahn Process Networks

Kahn [4, 5] dictates that communication channels are the only method that processes may use to communicate. Process nodes may have any number of incoming or outgoing queues. Nodes with outgoing queues produce data elements (tokens) on those queues, and nodes with incoming queues consume data. A

consumer node cannot detect the presence or absence of data on an input queue, and its execution is suspended when it attempts to consume data from an empty queue. However, producers are always enabled for execution, and queues are of infinite length. This can cause unbounded accumulation of data on a given queue.

The results of a process network program do not depend on the order of execution of the process nodes. The tokens produced on all communication channels are the same for every execution order that obeys these semantics. This important property of process networks is called *determinism*. Because process networks are determinate, the can be executed sequentially or in parallel with the same outcome.

Kahn developed a formal, mathematical representation of process networks [4]. By using streams to represent the channels, and functions to represent the processes, a process network program can be described by a set of equations. The histories of the streams in the network correspond to a unique least fixed point of these equations, and are not affected by the scheduling of operations. Kahn uses this fact to prove that process networks are determinate.

For Kahn process networks, termination is a property of the program and does not depend on the execution order. The unique least fixed point determines the value and length of every stream in the program, but there are many possible execution orders that can lead to this solution. In a *terminating* program, all streams in the solution are finite in length. A *non-terminating* program contains at least one stream that is infinitely long.

Although the total stream lengths are a property of the program, the number of unconsumed tokens that can accumulate on communication channels depends on the choice of execution order [7]. Using Fig. 3.1 as an example, if the nodes are executed as {A, B, A, B, …}, then channel P must buffer only one data element. However, if process A executes an infinite number of times before B executes, then P must buffer an infinite number of data elements.

A process network is *bounded* if a complete execution exists in which token accumulation on any channel will not exceed some finite constant. In a *strictly bounded* network, token accumulation on any channel will not exceed some finite constant for *all* complete executions.

For some restricted forms of process networks, such as synchronous dataflow [8], termination and boundedness are decidable, and a static schedule can be determined offline in finite time. However, the problems of determining whether a general Kahn process network will terminate, or can be scheduled in bounded memory are undecidable. In this context, the scheduler must work dynamically, as the program executes.

3.2: BOUNDED SCHEDULING OF PROCESS NETWORKS

Infinitely large queues cause obvious problems; execution in bounded memory is necessary for any practical implementation. Any arbitrary process network can be transformed into a strictly bounded one by adding a feedback channel for every data channel and modifying each process. Fig. 3.2 shows how the simple bounded network in Fig. 3.1 can be made strictly bounded.

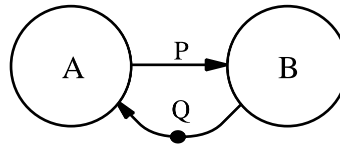

Figure 3.2: A strictly bounded process network.

However, this method could introduce *artificial deadlock*, thus transforming a non-terminating program into a terminating one.

Rather than transform the process network, Parks developed rules for dynamic scheduling in bounded memory [7]. Parks lists two requirements for the scheduler:

1. **Complete Execution** – The scheduler should implement a complete execution of the process network program. If the program is non-terminating, then it should be executed forever without terminating.
2. **Bounded Execution** – The scheduler should (if possible) execute the process network program so that only a bounded number of tokens ever accumulate on any of the communication channels.

When these requirements conflict (such as for unbounded programs), requirement 1 takes precedence over requirement 2. That is, a complete, unbounded execution is preferable to a partial, bounded one.

Parks goes on to show that the following rules will yield a bounded schedule, if one exists.

1. Block when attempting to read from an empty queue.
2. Block when attempting to write to a full queue.

3. If we reach *artificial deadlock*, where execution has stopped because processes are blocked writing to full channels, increase the capacity of the smallest full queue until the producer associated with it can fire.

This bounded scheduling policy has the desired behavior for all types of programs – terminating or non-terminating, strictly bounded, bounded, or unbounded. Now any scheduler will work, because any execution leads to bounded buffering on the queues. This model is well-suited for implementation using the threaded model of concurrent programming.

3.3: Karp and Miller Computation Graphs

Karp and Miller [6] developed a restricted model similar to process networks, called "computation graphs", which are also determinate. The computation is represented by a finite graph containing nodes $v_1,...,v_k$, each associated with a function $O_1,...,O_k$, connected by a set of arcs (edges) $d_1,...,d_t$.

Each arc d_p has four non-negative integer constants associated with it:

- A_p - The number of data words initially present.
- U_p - The number of data words inserted each time the producer node fires.
- W_p - The number of data words removed each time the consumer node fires.
- T_p - The number of data words required to be on the arc before the consumer can fire.

Clearly T_p must be greater than or equal to W_p. Three basic rules apply to the execution of a computation graph:

1. No node will fire unless each input edge d_p has at least T_p data words.

2. The execution will continue until every node has at least one input edge with less than T_p data words.
3. In a "proper execution," every node that has at least T_p data words on each input edge will eventually fire.

For this restricted model, the questions of termination and boundedness are decidable. Karp and Miller provide iterative algorithms to determine these properties. Computation graphs can be statically scheduled. In fact, synchronous dataflow [8] is a special case where $T_p = W_p$ for every arc.

The digital interpolation beamforming algorithms presented in this paper (and many other signal processing algorithms) can be modeled as Karp and Miller computation graphs. The process network implementation presented in Chapter 4 borrows the concept of a firing threshold (T_p) in addition to a dequeue count (W_p) from Karp and Miller. Although the beamformer system can be statically scheduled, dynamic scheduling with Process Networks is preferred so that execution on a symmetric multiprocessing system can effectively utilize parallel hardware. Static scheduling of algorithms across multiple processors is beyond the scope of this implementation.

3.4: SUMMARY

The Process Network model of computation represents a program in directed graph notation, where each node represents an independent process and each edge represents a communication channel. This model provides for correctness, and guarantees determinate execution of the program regardless of the scheduling algorithm used. Dynamic scheduling based on the availability of

data allows execution in bounded memory. This bounded memory Process Network model is well-suited for implementation using the thread model of concurrent programming.

Chapter 4: Process Network Implementation

Our implementation of Process Networks is intended for computationally intense algorithms on large symmetric multiprocessing workstations. Many signal processing algorithms are modeled using directed graphs, where each node represents fine-grain computations such as addition and multiplication. For example, the fast Forrier transform (FFT) butterfly [12] may be modeled in this manner. However, a fine level of granularity is inappropriate for this implementation, because the overhead of dynamic scheduling will dominate the overall execution time.

We use nodes of larger granularity – such as an FFT node, a filter node, or a beamformer node. The graphs drawn using this methodology are essentially block diagrams. The general rule of thumb is that the cost of firing a node should be much larger than the cost of a (relatively lightweight) thread context switch. However, if a node is too computationally costly, it may need to be divided into smaller pieces in order to run in real time. Generally, there is a tradeoff between overhead and latency.

Although our implementation of Process Networks is applied to beamforming in this paper, it could be used on any appropriate processing task, and is in no way limited to this purpose. We use a layered approach based on the C++ inheritance mechanism to build interfaces and functionality.

Section 4.1 discusses the ThresholdQueue class, which provides efficient circular buffers for general-purpose processors. This queue is fundamental to the

Process Network implementation, so implementation details and usage examples are given. It also covers the MmapThresholdQueue, which uses the Unix virtual memory manager to maintain circularity, so that data copying is not required.

Section 4.2 covers the implementation of Process Network nodes, and their relation to threads. It also addresses the interface for nodes to communicate with queues, using the classes PNNodeInput and PNNodeOutput. A code fragment for a sample node is provided. These classes adhere to the rules for bounded scheduling of Process Networks.

Section 4.3 discusses the implementation of Process Network communication channels. The abstracted base classes provide versatility in the delivery mechanism of these channels. The most generally used channel type is the PNThresholdQueue, which provides ThresholdQueue functionality along with bounded scheduling rules.

Section 4.4 explains how these node and queue classes are tied together in order to produce a Process Network program. A directed graph program is given, and its corresponding C++ implementation is provided.

4.1: THE THRESHOLDQUEUE TEMPLATE CLASS

The ThresholdQueue C++ class is near the base of the inheritance hierarchy, and is optimized for data-intensive applications. The ThresholdQueue works much like a typical queue, but is intended to make up for the lack of circular address buffers in general purpose processors. Because ThresholdQueue is a template class, it can queue any type of data. A goal in the design of this class was to prevent unnecessary copying of data. Therefore, the user reads and writes

data directly from queue memory, and data is guaranteed to be contiguous in memory. This reduces overhead, and simplifies the implementation of algorithms that interface to these queues.

The Karp and Miller concept of separating the firing threshold (T_p) from the dequeue count (W_p) is fundamental to the ThresholdQueue. In addition to the type of data being queued, instantiation requires the initial queue length and a maximum threshold that will ever be requested. This threshold is the maximum of Karp and Miller's U_p, W_p, and T_p parameters for a queue.

The basic interface to the ThresholdQueue is reused throughout the process networks implementation, using pointers to avoid data copying by the user. A transaction with a ThresholdQueue is a three-step process:

1. Get a pointer to some number of data elements (which are contiguous in memory) using the GetEnqueuePtr or GetDequeuePtr method, each of which takes a threshold and returns the pointer.
2. Operate on the data by de-referencing the pointer, up to the threshold length.
3. Actually insert or remove the data by calling the Enqueue or Dequeue method, each of which takes a count.

The code fragments in Figs. 4.1 and 4.2 demonstrate insertion and removal transactions for a ThresholdQueue.

```
int count = KarpAndMillerUp;
T* writePtr = theThresholdQueue.GetEnqueuePtr(count);
for (int i=0; i<count; i++)
       writePtr[i] = DataToWrite(i);
theThresholdQueue.Enqueue(count);
```

Figure 4.1: Enqueuing data into a ThresholdQueue.

```
int threshold = KarpAndMillerTp;
int count = KarpAndMillerWp;
const T* readPtr = theThresholdQueue.GetDequeuePtr(threshold);
for (int i=0; i<threshold; i++)
       DataToRead( readPtr[i] );
theThresholdQueue.Dequeue(count);
```

Figure 4.2: Dequeuing data from a ThresholdQueue.

Although the relationships to Karp and Miller's parameters are demonstrated in these code fragments, the threshold and count values are not static, and may change on each queue transaction. The count must always be equal to or less than the threshold, although this is not strictly enforced. For enqueueing, the count and the threshold are generally equal.

The ThresholdQueue implements its apparent circular addressing by mirroring the beginning of the queue's data region (up to the maximum threshold) just past the end of the queue's data region. Using this methodology, the queue can provide a pointer to a contiguous block of data elements even when operating near the end of the data region. The queue manages this mirroring, and guarantees that the same data resides in both locations. Fig. 4.3 illustrates the ThresholdQueue implementation.

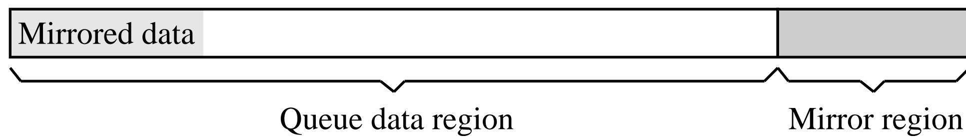

Figure 4.3: ThresholdQueue implementation.

The ThresholdQueue has a tradeoff between memory usage and performance. When the data region is much larger than the mirror region, the queue rarely needs to copy data. When the mirror region is as large as the data region, copying must occur frequently, increasing overhead and sacrificing performance. Fortunately, memory is usually abundant on a workstation.

On some systems (including Sun Solaris), the virtual memory manager can be used to prevent the ThresholdQueue from having to copy data at all. The Unix system call `mmap()` is used to map virtual memory objects into a process's address space. By mapping a shared memory object to multiple virtual addresses, the same physical memory pages appear at multiple addresses, and apparent circular addressing is achieved.

The C++ class MmapThresholdQueue is derived from the ThresholdQueue class. It implements the same functionality and interface, but never has to perform copying of data when managing the mirror region. As a side effect, the queue data and mirror regions must both be multiples of the system memory page size, which is 8 kilobytes (kb) in Solaris. Therefore, the MmapThresholdQueue rounds the queue size and the threshold size up to the next multiple of the page size.

4.2: PROCESS NODES

The process nodes of the Process Network model are implemented using the thread model of concurrent programming. Each node in the Process Network program is a different thread. These multiple threads can run concurrently when the program has parallelism, and thus can take advantage of multiple processors. Thread implementations are generally intended to provide high performance in a low-overhead environment.

The Portable Operating System Interface (POSIX) provides a standard thread interface, called Pthreads, which is source-code compatible with many versions of Unix. A particular advantage of Pthreads is that they can be given real-time scheduling priority. By realizing Process Networks with POSIX Pthreads, our implementation can be run on many different Unix platforms.

The base class for all process nodes is the C++ class PNNode, which is derived from a Pthread class. Since all nodes are derived from PNNode, porting this system to a different thread implementation should be fairly simple, requiring only a change to the PNNode class.

Since the only way that nodes can communicate with each other is via communication channels, an abstracted interface to these channels has been provided: the classes PNNodeInput and PNNodeOutput. In order to simplify node implementation and reduce the overhead of data copying, these classes use the same transaction mechanism as the ThresholdQueue. Again, these classes are templates, and can be used to communicate any data type. Figs. 4.4 and 4.5 show declarations of PNNodeInput and PNNodeOutput.

```
template<class T> class PNNodeInput {
	virtual const T*  GetDequeuePtr(ulong thresh) = 0;
	virtual void      Dequeue(ulong count) = 0;
	// others omitted for brevity
};
```

Figure 4.4: A partial declaration of PNNodeInput.

```
template<class T> class PNNodeOutput {
	virtual T*        GetEnqueuePtr(ulong thresh) = 0;
	virtual void      Enqueue(ulong count) = 0;
	// others omitted for brevity
};
```

Figure 4.5: A partial declaration of PNNodeOutput.

The methods GetDequeuePtr and GetEnqueuePtr are intended to be blocking. That is, they will not return until the threshold amount of data (or free space) is available. Note that these are virtual base classes – they cannot be instantiated. They only provide an interface for writing process nodes. Using these interfaces, the code for a node that eternally copies data is as simple as that in Fig. 4.6.

```
for(;;) {
	const T* readPtr = theInputQ.GetDequeuePtr(copySize);
	T* writePtr = theOutputQ.GetEnqueuePtr(copySize);
	for (int i=0; i<copySize; i++)
		writePtr[i] = readPtr[i];
	theInputQ.Dequeue(copySize);
	theOutputQ.Enqueue(copySize);
}
```

Figure 4.6: Code for a node that copies data.

The thread implementing this node (and running this code) will block on its input queue in GetDequeuePtr until there is data available to be copied. It will also block in GetEnqueuePtr until there is sufficient space in the output queue. Once there is data to copy and a place to put it, the node is free to perform the copy operation. After completion, the node notifies the input queue that it is done with the incoming data, and then notifies the output queue that data is present to be inserted.

Note that this interface obeys Parks' rules for bounded scheduling of Process Networks. Also notice that the node is not concerned with the implementation of the communication channels, but instead only their interface.

4.3: COMMUNICATION CHANNELS

Because of the abstracted interface to the communication channels, many different implementations could exist. A channel could send data to another process via shared memory, or to another computer through a network. A channel could also save its entire history to disk for program verification or debugging. However, most of the time data will simply be sent from one thread to another within the same process. The primary mechanism for this is the C++ template class PNThresholdQueue.

This class is multiply inherited from the classes PNNodeInput, PNNodeOutput, and ThresholdQueue. Again, it is a template class, and can send and receive any type of data. Since it is derived from PNNodeInput and PNNodeOutput, it can be used as the input or output of any process node. Fig. 4.7 shows a partial declaration of PNThresholdQueue.

```
template<class T> class PNThresholdQueue
:       public ThresholdQueue<T>,
        public PNNodeInput<T>,
        public PNNodeOutput<T>
{
        PNThresholdQueue(ulong qLength, ulong maxThresh);
        const T*    GetDequeuePtr(ulong thresh);
        void        Dequeue(ulong count);
        T*          GetEnqueuePtr(ulong thresh);
        void        Enqueue(ulong count);
        // others omitted for brevity
};
```

Figure 4.7: A partial declaration of PNThresholdQueue.

The PNThresholdQueue is responsible for blocking any nodes according to Parks' firing rules. The POSIX Pthread condition variable mechanism is used to awaken nodes at the proper time. When a producer enqueues data into a queue and there is a blocked consumer waiting at the other end, the consumer will be signaled to awaken if the operation provides enough data for the consumer to fire. This signaling occurs inside the queue implementation, without the knowledge of the involved processing nodes. Similarly, when a consumer dequeues data and there is a blocked producer at the other end of the queue, the producer will be awakened if there is enough free space in the queue for the producer to fire. This method awakens the threads as soon as data (or free space) is available, minimizing latency.

At this time, deadlock detection as described in Section 3.2 is not implemented. For the class of real-time problems that this implementation is intended to solve, deadlock detection is not necessary -- it is required for avoiding artificial deadlock, and for execution of unbounded programs. In this

implementation, queues are generally allocated to be much larger than their minimum possible (deadlock avoiding) size, because of the performance reasons described in Section 4.1. Unbounded programs have no place in real-time, and can never be fully executed in a real-world implementation. However, the addition of deadlock detection would make this a more complete implementation of the Process Network model, and will be implemented in the future.

4.4: CONSTRUCTING A PROCESS NETWORK PROGRAM

In order to tie these concepts together, an example of constructing a Process Network program is provided. The Process Network graph is shown in Fig. 4.8.

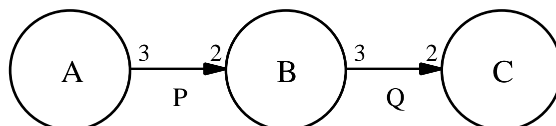

Figure 4.8: A sample process network program.

The arcs ‘P’ and ‘Q’ are used to connect nodes ‘A’, ‘B’, and ‘C’. Each time ‘A’ fires, it produces 3 tokens. Each time ‘B’ fires, it consumes 2 tokens and produces 3 tokens. Each time ‘C’ fires, it consumes 2 tokens. Fig. 4.9 gives simple code to implement this network.

```
int main() {
        PNThresholdQueue<T>      arcP(4,3);
        PNThresholdQueue<T>      arcQ(4,3);
        MyProducerNode           nodeA(arcP);
        MyTransmuterNode         nodeB(arcP, arcQ);
        MyConsumerNode           nodeC(arcQ);
}
```

Figure 4.9: Implementation of the process network program.

Recall that the parameters to the PNThresholdQueue constructor are the queue length, and the maximum threshold. In each arc, the maximum number of elements enqueued or dequeued is 3. In order to prevent artificial deadlock, the minimum queue lengths are 4 in both cases. This is the case that uses the minimal amount of memory. If more memory is available, then context switching can be reduced (and latency increased) by increasing these queue sizes.

Currently, we only provide support for building a process network by programming in the C++ language. Our future goal is to have the capability to build process network programs from a text file or a graphical user interface, using a tool such as Ptolemy [13] from the University of California at Berkeley.

4.5: SUMMARY

This chapter discusses the implementation of Process Networks in C++. The efficient circular buffering classes are described, with examples and implementation details. We give examples of constructing nodes and communication channels that adhere to the rules for bounded scheduling of Process Networks. A directed graph program is given, along with its implementation in C++.

Chapter 5: Beamformer Implementation

Fig. 5.1 shows a block diagram of the beamformer stages implemented. These stages correspond to nodes in the Process Network implementation.

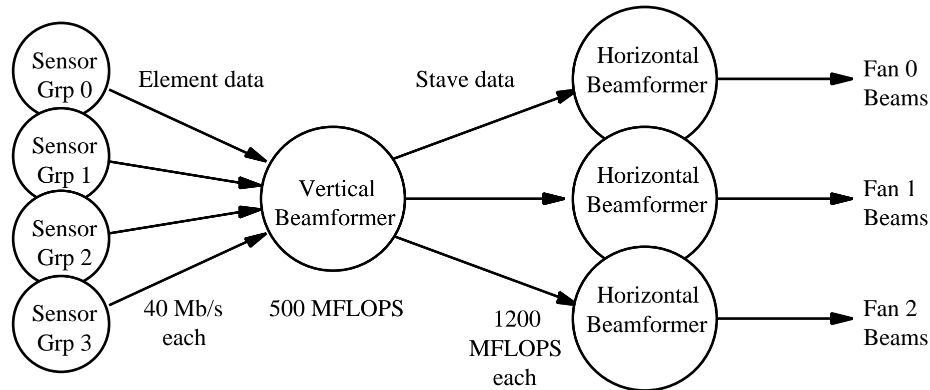

Figure 5.1: A block diagram of the beamformer stages.

The sensor array consists of 80 elements horizontally by 10 elements vertically, for a total of 800 elements. An analog-to-digital converter samples each element, and this digital data comes from the sensor array via four telemetry links, each at 40 megabytes per second.

The first stage of this system is vertical beamforming or "staving." In this stage, the vertical elements are weighted and summed, thus calculating 80 logical horizontal elements, or staves. The element data is first converted to 32-bit floating-point, and then three different sets of weights are used to calculate 3 sets of stave outputs.

The three horizontal beamformers combined perform approximately 3.6 billion (giga) floating-point operations per second (GFLOPS) at real-time, which

is the bulk of the computational requirement in the system. The horizontal beamformer has been a large focus of this research, and is discussed in Section 5.1. The vertical beamformer requires approximately 500 million floating-point operations per second (MFLOPS) at real-time, and is discussed in Section 5.2.

5.1: HORIZONTAL BEAMFORMING

Each horizontal beamformer stage performs digital interpolation beamforming as described in Chapter 2, using single precision (32-bit) floating-point numbers. For each vertical weighting, the 80 stave outputs are used to form 61 beams. On average, approximately 50 staves contribute to each beam, and 2 points are used to calculate each interpolation result. The maximum sample delay due to the array geometry is 32. When modeling this digital interpolation beamformer operation as a sparse FIR filter, the filter length is 2560 coefficients, 96% of which are zero.

Fig. 5.2 shows a sample set of coefficients used. Although organized as a 2560-point one-dimensional FIR filter, the information contained in the coefficients is more evident when plotted as sample number vs. stave number. In the 2-D grid, zero coefficients are white and non-zero coefficients are black. The curved shape of the array is clearly visible in the coefficients.

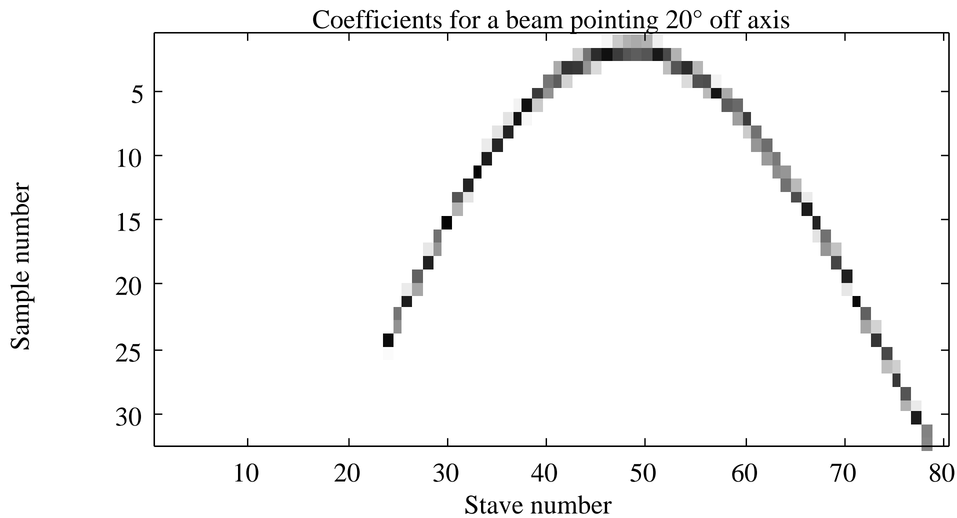

Figure 5.2: Beamforming coefficients for one beam.

All beamforming coefficients were generated in Matlab, a computing environment from The MathWorks, Inc. [14]. Coefficient indices are based on 2-point interpolation of the signal delay due to the array geometry and steering angle. No shading was used in generation of the coefficient values, but this does not affect the number of operations performed during beamforming.

The digital interpolation beamforming algorithm is highly parallelizable, and several different methods for dividing the task among threads were examined. One obvious approach is to calculate different beams using different threads, thus dividing the task by beam. This follows naturally from "partial-sum" beamforming [2], and uses a minimal amount of memory, with minimum latency. Indeed, this method is frequently employed in custom hardware designs that use digital signal processor (DSP) style computing engines. However this "DSP-

minded" approach suffers from poor cache utilization on a workstation, which results in poor performance.

A more "workstation-minded" approach is to divide the task in time. Memory bandwidth, not raw processing power, is the major obstacle. This method requires more memory and gives higher latency, but delivers better performance on a workstation through superior cache utilization. Section 5.1.1 discusses the cache utilization.

Now, each thread performs exactly the same set of operations, only on different data. This simplifies code development – only one highly optimized horizontal beamforming routine is required. This beamforming kernel routine needs to know the beamforming coefficients, a pointer to the incoming data (staves), a pointer that tells where to put the results (beams), and the number of samples to calculate. For organizational purposes, the C++ classes, HorizontalBeamformerCoefs and HorizontalBeamformer were developed. By examining the assembler output of the compiler, the C++ source code has been hand-optimized for improved performance. Section 5.1.2 explains the integration of the horizontal beamformer with the Process Network framework.

5.1.1: Cache Utilization

Best performance is obtained when the calculation is small enough to fit in the cache, so that the number of cache misses is relatively small. The coefficients for this implementation are 36 kilobytes (kb), and the calculation of each sample requires approximately 10 kb of data. The level 2 cache, which is external to the processor, typically has a size on the order of megabytes. Thousands of samples

can be calculated from within this cache. However, the (internal) level 1 cache of an UltraSPARC II processor is only 16 kb.

Within the kernel beamforming routine, care must be taken to heed this memory usage limit. The best performance so far has been achieved by making multiple passes through the same data, calculating only a subset of the result each pass, such that both the element data and a subset of the coefficients fit in the level 1 cache at each step. For each additional output sample calculated, 80 more staves of input data must be fetched from the level 2 cache.

5.1.2: Integration with Process Networks

Implementation of a node that simply calls the horizontal beamformer kernel routine is easy, but this node cannot achieve real-time performance. A method for dividing the beamforming task in time is needed. In order to divide this calculation without copying data, a horizontal beamformer node manages multiple worker nodes. This is illustrated in Fig. 5.3.

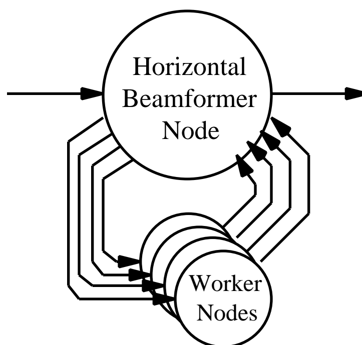

Figure 5.3: A horizontal beamformer node.

When the horizontal beamformer node fires, it sends a task description to each of several worker nodes, and then blocks on each worker node by waiting for a result. The task description sent across the communication channels is the same as that provided to the beamforming kernel routine -- a pointer to the incoming data (staves), a pointer that tells where to put the results (beams), and the number of samples to calculate. Each worker node simply returns an integer that indicates error status. The number of worker nodes can easily be increased or decreased, as the processing performance requires. This method is similar to a thread pool [9], which is a common workstation multiprocessing tool.

5.2: VERTICAL BEAMFORMING

For the vertical beamformer, no time delay is necessary, and no digital interpolation is required. For each sample of the logical 80 staves, one dot product per vertical shading set must be calculated. Given 10 vertical elements and 3 shadings, this is only 4800 operations per sample (40% of a single horizontal beamformer, and under 12% of the entire system). All of the element data is consecutive in memory, and the coefficients are small, so the small size of the level 1 cache is less of an issue.

Although the vertical beamforming operation itself is extremely simple, this stage must also synchronize the element data, convert it to floating-point format, and arrange the result for the following horizontal beamformer. Efficient operation of the horizontal beamformer requires that the stave outputs be interleaved, as shown in Fig. 5.4.

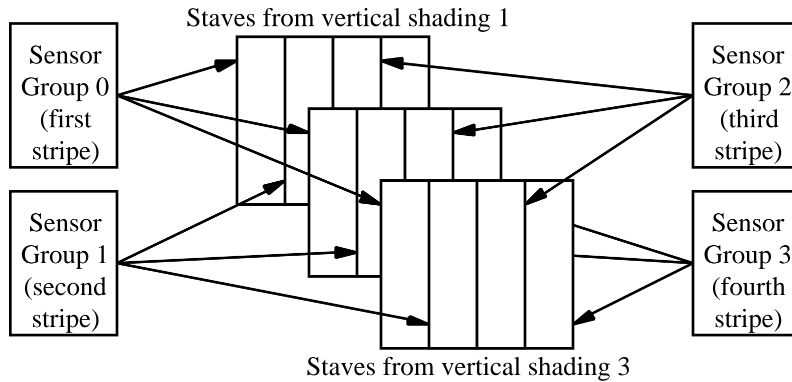

Figure 5.4: Interleaving of the vertical beamformer output.

When different processors are performing interleaving by writing to the same memory area, cache contention (thrashing) can result. As with the horizontal beamformer, dividing the task by time gives superior performance. Once again, a kernel beamforming routine was developed and optimized. The innermost loop of the vertical beamformer kernel routine was written in UltraSPARC assembly language.

Similarly to Section 5.2.2, a vertical beamformer node is used to manage multiple worker nodes so that the beamformer can operate in real time. When the vertical beamformer node fires, it sends a task description to each of the worker nodes via a communication channel. It then blocks on the return channel of each worker, waiting for the result which indicates completion.

Although the vertical beamformer performs a relatively small number of operations on behalf of beamforming, it must convert the incoming data and interleave the results. These factors significantly affect the performance of the

vertical beamformer. Further optimization is possible, as demonstrated with the results in Chapter 6.

5.3: SUMMARY

This chapter describes the implementation of beamforming algorithms in C++. The digital interpolation beamformer is the most computationally intensive algorithm in this system; much time has been dedicated to its optimization. Dividing the computing load by time gives the best cache utilization, and therefore the best performance. Although vertical beamforming is a simpler algorithm, this stage must convert the incoming data and interleave the results. Again, dividing the task by time yields superior performance.

We leverage existing tools and software to implement the digital interpolation beamformer. Matlab was used to generate and test beamforming coefficients, and to verify the beamformer output.

Beamforming kernel routines have been developed which can execute in parallel on multiple processors. The Process Network implementation uses master nodes with teams of worker nodes. This method avoids copying data, and is similar to the thread pool model.

Chapter 6 discusses beamformer performance results.

Chapter 6: Results

This chapter presents and discusses benchmark results for various different beamformer implementations. These benchmarks were performed on a Sun Ultra Enterprise 4000 with 8 UltraSPARC-II processors running at 336MHz each. The Sun Solaris operating system version 2.6 was used, with threads executing in the “real-time” class.

All results are determined as the average time over 10 trials to calculate about 2.6 seconds of data. Care was taken to prevent the caching of incoming data before the benchmarks were performed. If this had not been the case, artificially elevated results would have been obtained; a real-time system does not have pre-cached data to work with.

Section 6.1 presents the performance results for horizontal beamforming. The beamforming kernel speed is evaluated, and scaling is measured using thread-pool algorithms. Section 6.2 similarly presents the vertical beamformer results. Section 6.3 presents results for the Process Network beamformer system presented in Fig. 5.1, and compares them to thread-pool results. It also discusses the scalability of Process Networks.

6.1: Horizontal Beamformer Performance

The horizontal beamformer calculates 61 beams from an array of 80 logical horizontal elements (staves). Two samples are interpolated to calculate each beamforming time delay, and approximately 50 staves contribute to each beam, on average. Calculation of one sample requires over 12,000 single-

precision floating-point operations, in addition to over 6000 index lookups for location of the proper time index in the sparse FIR filter model. The benchmark performed requires 3.2 billion floating-point operations.

In order to evaluate the performance of the digital interpolation beamformer kernel, a non-threaded reference benchmark was performed. The UltraSPARC-II processor can execute 2 floating-point operations per clock cycle. At 336 MHz, this is a peak performance of 672 million floating-point operations per second (MFLOPS). The first row of Table 6.1 displays the result of 410 MFLOPS for the non-threaded horizontal beamformer. Despite the index lookups, the beamforming kernel routine can keep the utilization of the floating-point units at 61%, i.e. 1.22 floating-point operations are performed per clock cycle. This routine is currently written in highly optimized C++ code. Perhaps a hand-optimized assembly coding could push this performance even further.

Beamformer Type	time (sec)	MFLOPS	speedup	percent utilization
non-threaded	7.847	410.0	(1.000)	(100.0)
1 thread pool	7.795	412.7	1.007	100.7
2 thread pool	4.022	799.9	1.951	97.6
4 thread pool	2.075	1550.4	3.781	94.5
6 thread pool	1.422	2262.3	5.518	92.0
8 thread pool	1.112	2893.0	7.057	88.2

Table 6.1: Horizontal beamforming benchmark results.

The remaining rows of Table 6.1 show the results for various batch-mode thread-pool beamformer implementations, each with a different number of threads. The “speedup” and “percent utilization” columns are referenced to the

non-threaded case in the first row. Because the thread-pool synchronization overhead is low, the addition of a processor and its associated cache gives the one-thread thread-pool implementation a slight (less than 1%) performance advantage over the non-threaded reference case.

As Table 6.1 shows, the performance of these thread-pool horizontal beamformers scales fairly well with additional threads. The real-time goal of beamforming at 1200 MFLOPS is met with 4 threads, where the horizontal beamformer delivers over 385 MFLOPS on each of 4 (336MHz) processors. Note that for the 8-thread case, the beamformer must share a CPU with the operating system. In order to achieve the best scaling performance, an additional level of detail must be provided when mapping software on to the hardware. Binding of threads onto individual CPUs was not performed in this implementation.

6.2: VERTICAL BEAMFORMER PERFORMANCE

The vertical beamformer calculates 3 sets of 80 logical horizontal elements (staves) using 10 vertical elements each, for 800 total elements. Although a mere 4800 floating-point operations per sample is required, the incoming data must be converted from the native format of the A/D converter to 32-bit floating-point format, and the outputs must be interleaved (as shown in Section 5.2). Unlike the horizontal beamformer, no index lookup is required. This benchmark requires approximately 1.3 billion floating-point operations, consuming 400 megabytes of element data.

Again, the kernel beamforming routine is evaluated by using a non-threaded reference benchmark. As Table 6.2 shows, the vertical beamformer

performance is currently unimpressive at 134.7 MFLOPS. This is only 20% of the peak performance rate of the floating-point units. The real performance problem lies in the conversion to floating-point format.

Conversion of 800 A/D outputs to floating-point format requires approximately 4000 operations, thereby drastically reducing beamforming performance. Integer implementations for the vertical beamformer have been attempted, but no performance increase has been seen. Use of the UltraSPARC Visual Instruction Set (VIS), which works only with fixed-point data, would result in an unacceptable loss of precision and dynamic range.

Beamformer Type	time (sec)	MFLOPS	speedup	percent utilization
non-threaded	9.339	134.7	(1.000)	(100.0)
1 thread pool	9.308	135.2	1.003	100.3
2 thread pool	4.864	258.7	1.920	96.0
4 thread pool	2.641	476.4	3.536	88.4
6 thread pool	2.010	626.0	4.646	77.4
8 thread pool	1.686	746.3	5.539	69.2

Table 6.2: Vertical beamforming benchmark results.

Table 6.2 shows the results for various batch-mode thread-pool vertical beamformers, referenced to the non-threaded case in the first row. Although the real-time goal of 500 MFLOPS is nearly met with 4 threads, the scaling performance is currently rather disappointing. Clearly more optimization effort is needed on the vertical beamformer implementation. Further optimization is possible by developing hand-optimized assembly language in conjunction with Sun’s cycle-accurate simulator.

6.3: PROCESS NETWORK BEAMFORMER PERFORMANCE

Thread pools are the traditional method for lightweight multiprocessing on a UNIX workstation. In order to evaluate the performance of the full Process Network beamforming system depicted in Fig. 5.1, it is compared with thread-pool implementations.

The thread-pool beamforming system loads all input data into memory, and allocates memory for placing the results of each stage. First, a thread-pool vertical beamformer calculates three sets of stave data from the element data. Then 3 thread-pool horizontal beamformers sequentially execute and calculate beam results from the stave data. This system uses over 800 Mb of memory for data alone.

Each of the thread-pools in this system uses 8 worker threads, because that is the number of processors on the workstation executing the benchmark, and this gives the greatest performance. Not surprisingly, the time taken to execute the full benchmark is roughly the same as the sum of the times for a vertical beamformer and 3 horizontal beamformers from Sections 6.1 and 6.2 above.

As shown in Table 6.3, the thread-pool beamformer and the Process Network beamformer achieve approximately the same results. Processing 2.6 seconds of data on 8 CPUs takes just over 5 seconds, which is slightly better than half of the real-time goal.

Beamformer Type	time (sec)	MFLOPS
thread pool	5.053	2159.0
process network	5.024	2171.5

Table 6.3: Process Network vs. thread-pool performance results.

The Process Network system has distinct advantages. Because it is more "stream" oriented, it has lower latency and uses less memory. Without real-time input and output devices, the same amount of memory is required for the input and output data. However, the communication channels in the network need only be large enough to prevent artificial deadlock. In this implementation, the resulting data memory size is reduced by over 200 Mb.

All nodes of the Process Network are operating all of the time, as the flow of data permits. In the thread-pool implementation, each beamforming stage uses 8 worker threads because that is the number of CPUs. In the Process Network implementation, each stage needs only enough worker nodes to keep up with the real-time requirement. For the horizontal beamformer, 4 worker nodes are required. The poorly performing vertical beamformer uses 6 worker nodes in this implementation.

Although, the memory, latency, and scheduling issues can be better addressed in the thread-pool implementation, these advantages come automatically when using the Process Network model of computation. An additional advantage of the Process Network implementation is its scalability; the same Process Network beamformer program would automatically be scaled by the operating system according to the number of available processors.

The thread-pool beamformer was written with knowledge of the number of CPUs in the hardware. If the number of CPUs were to increase, then the number of worker threads would have to be changed in order to utilize these additional processors. Arbitrarily creating large numbers of worker threads would cause unnecessary overhead.

The Process Network beamformer will scale without any change to the executable. If there were more processors than the sum of all worker nodes, those processors would not be utilized. However, those extra processors are not needed for the system to meet its real-time goal. In order to increase the performance past that point, more worker nodes would simply be added.

Unfortunately, an 8-processor machine was the largest configuration available to this project for benchmarking. Table 6.4 shows scaling results for the same Process Network beamformer executable, running on a varying number of CPUs. Solaris system administration tools were used to disable CPUs in the 8-processor machine, so that this test could be performed. The Process Network beamformer scales fairly well from 2 to 8 processors.

Number of CPUs	time (sec)	MFLOPS	speedup	percent utilization
2 CPUs	17.458	624.9	(1.000)	(100.0)
4 CPUs	9.046	1206.0	1.930	96.5
6 CPUs	6.408	1702.5	2.724	90.8
8 CPUs	5.031	2168.4	3.470	86.8

Table 6.4: Process Network beamformer scalability.

Based on these benchmarks, real-time operation of this Process Network beamforming system on 16 CPUs is an attainable goal. Better optimization of the

vertical beamformer kernel routine is required, and its very poor scaling performance must be addressed. Performance losses due to additional scaling overhead must also be reduced. Binding threads to individual processors may help to attain this goal.

6.5: SUMMARY

Horizontal beamforming accounts for the bulk of this system's processing requirement. We have focused on optimizing this stage, and performance and scaling are very good. However, the vertical beamforming kernel needs further optimization, because the format conversion of the incoming data to floating-point is causing the performance to suffer. The scalability of the vertical beamformer is also disappointing, and needs improvement.

The Process Network beamforming implementation compares favorably to the more traditional batch-mode thread-pool implementation. It is slightly faster, uses less memory, and has lower latency than the thread-pool version. An additional advantage of the Process Network implementation is its superior (and automatic) scalability to parallel hardware.

Chapter 7: Conclusion

Computationally intensive sonar beamforming algorithms have been implemented using Process Networks and POSIX Pthreads under the Sun Solaris operating system. These software systems have been benchmarked on a Sun workstation with 8 UltraSPARC-II processors, running at 336MHz. Highly optimized software has been developed to implement (horizontal) digital interpolation beamforming algorithms, and performance of better than 385 MFLOPS on each of 4 processors has been observed.

The Process Network model provides for correctness and determinacy, and can guarantee execution in bounded memory. This model is excellent for digital signal processing systems, and captures their concurrency and parallelism. The Process Network implementation provided compares favorably with the more traditional thread-pool model, and provides a low-overhead, high-performance, scalable framework.

Our future goal is to have the capability to build process network programs from within a graphical user interface, using a tool such as Ptolemy [13]. In this implementation, the workstation is both the development platform and the target architecture, and we can deploy the computer-aided design tools along with the design.

Implementing this beamforming system on a commercial Unix workstation reduces manufacturing costs, development costs, and development time by a factor of three, and volume and weight by a factor two when compared

to a custom hardware solution. This software implementation also provides superior portability, upgradability, and maintainability.

Although further optimization is required for the vertical beamforming software, it is feasible for this high-resolution multi-fan digital interpolation beamformer to execute in real-time on a Unix workstation. This 4 GFLOP system would require 16 UltraSPARC-II processors running at 336 MHz.

References

[1] R. G. Pridham and R. A. Mucci, “A Novel Approach to Digital Beamforming.” *Journal Acoustical Society of America,* vol. 63, no. 2, pp. 425-434, Feb. 1978.

[2] R. A. Mucci, “A Comparison of Efficient Beamforming Algorithms.” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-32, no. 3, pp. 548-558, June 1984.

[3] R. G. Pridham and R. A. Mucci, “Digital Interpolation Beamforming for Low-Pass and Bandpass Signals.” *Proceedings of the IEEE*, vol. 67, no. 6, pp. 904-919, June 1979.

[4] G. Kahn, "The semantics of a simple language for parallel programming." *Information Processing,* pp. 471-475, Stockholm, Aug. 1974.

[5] G. Kahn and D. B. MacQueen, "Coroutines and networks of parallel processes." *Information Processing*, pp. 993-998, Toronto, Aug. 1977.

[6] R. M. Karp and R. E. Miller, “Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing.” *SIAM Journal*, vol. 14, pp. 1390-1411, Nov. 1966.

[7] T. M. Parks, “Bounded Scheduling of Process Networks.” *Technical Report UCB/ERL-95-105,* PhD Dissertation, EECS Department, University of California, Berkeley, CA 94720, Dec. 1995.

[8] E. A. Lee and D. G. Messerschmitt, “Static scheduling of synchronous dataflow programs for digital signal processing.” *IEEE Transactions on Computers*, vol. C-36, no. 1, pp. 24-35, Jan. 1987.

[9] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads Programming*. O’Reilly and Associates, Sebastopol, CA, 1996.

[10] R. J. Urick, *Principles of Underwater Sound.* McGraw-Hill Book Company, New York, NY, 1975.

[11] The IEEE Portable Applications Standards Committee (PASC) Web Page: http://www.pasc.org/

[12] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*. Prentice Hall, Englewood Cliffs, NJ, 1989.

[13] The Ptolemy Web Page: http://ptolemy.eecs.berkeley.edu/

[14] The MathWorks Web Page: http://www.mathworks.com/

Vita

Gregory Eugene Allen was born to Gordon Eugene and D'Maris Anne Allen on October 30, 1968 in Austin, Texas. He attended Austin's John H. Reagan High School, and graduated salutatorian in 1987. In 1991, Greg received his Bachelor of Science in Electrical Engineering from The University of Texas at Austin, graduating with Highest Honors. In 1993, he returned to The University as a part time graduate student.

Greg has been employed with Applied Research Laboratories (ARL) at UT's J. J. Pickle Research Campus since the summer of 1986, when he was hired through the High School Apprentice Program. Since 1988 he has worked on high-frequency, high-resolution sonar systems in the Advanced Sonar Group at ARL. In addition to design engineering, Greg has been involved in system-level testing, installation, and deployment aboard US Navy submarines, including a surfacing at the North Pole in 1993.

Greg and his wife, Dara Marie, have a daughter named Sabrina Fair, born in 1997.

Permanent address: 12011 Scribe Drive
Austin, TX 78759

This report was typed by the author.