Real-Time Sonar Beamforming on Workstations Using Process Networks and POSIX Threads

Gregory E. Allen and Brian L. Evans, *Senior Member, IEEE*

Abstract

We present a scalable framework for real-time data-intensive systems on commodity multiprocessor workstations. The framework is an extension of the Process Network model, which captures parallelism, guarantees determinate execution, and executes in bounded memory. We implement the framework using lightweight POSIX threads, and prototype a 4-GFLOP sonar beamformer on a 12-processor 336-MHz Sun Enterprise server. The beamformer scales nearly linearly from 1 to 12 processors.

Keywords: beamforming, high-performance computing, models of computation, multiprocessor programming, native signal processing, process networks, real-time systems, scalable software

---

G. Allen is with the Applied Research Laboratories, The University of Texas at Austin, Austin, TX 787813-8029, gallen@arlut.utexas.edu. B. Evans is with the Dept. of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX 78712-1084, bevans@ece.utexas.edu.

G. Allen was supported by the Independent Research and Development Program at Applied Research Laboratories at The University of Texas at Austin. B. Evans was supported by the Defense Advanced Research Projects Agency (DARPA) and the US Army under DARPA Grant DAAB07-97-C-J007 through a subcontract from the Ptolemy project at the University of California at Berkeley, and by the US National Science Foundation CAREER Award under Grant MIP--9702707.

1. Introduction

High-resolution sonar beamforming algorithms require on the order of a billion multiply-accumulates (MACs) per second, and have traditionally required custom parallel hardware for real-time implementation. Because these systems are not typically sold in high volumes, the non-recoverable engineering cost for custom hardware development may make this approach prohibitively expensive. The more recent "second-generation" approach connects dozens of commercial programmable processors in customized configurations, e.g. using a VME backplane. Using this commercial off-the-shelf (COTS) approach, one 4-GFLOP sonar beamformer requires about 100 80-MFLOP Analog Devices SHARC digital signal processors. Using COTS components reduces hardware development time and cost over custom parallel hardware, but requires significant software development, system integration, and system testing efforts. Software development involves partitioning of an algorithm onto several dozen processors, synchronizing results computed by the processors, and debugging code that is running concurrently on as many as 100 processors. The software developed using a COTS approach may be so closely tied to the hardware topology that the software cannot easily be reused in other designs or reconfigured in the field.

For the next generation of sonar beamformers, we advocate the use of commodity multiprocessor workstations, which are capable of native signal processing [1] with GFLOPS of performance. For a workstation beamformer, the development cost and time can be significantly reduced when compared to that for custom hardware or COTS systems with the same level of performance. By implementing beamformers in software on commodity high-performance workstations, we take advantage of advances in commercial workstations, thereby sharing hardware development costs with the high-volume workstation server market. We reduce software development efforts because mature high-volume operating systems and software tools can be used [2]. Workstations offer better software portability, hardware upgradability, and hardware maintainability than embedded COTS solutions. Because the development environment and target architectures are the same, the design tools can be deployed with the design to support in-the-field changes, which makes the target system dynamically reconfigurable. Using less powerful and less expensive workstations, software development can be performed in parallel on target hardware. Table 1 compares three generations of 4-GFLOP sonar beamformers.

Modern symmetric multiprocessing workstation operating systems dynamically schedule processes to balance the workload among the available processors. Unix extends this type of load balancing to *threads.* A thread is an independent flow of control within a process that has its own registers and stack, but shares data and code space with the rest of the process. The Portable Operating System Interface (POSIX) provides a standard thread library called Pthreads [3]. Pthreads have low overhead, and each can be scheduled with a fixed real-time priority. By partitioning an algorithm using large granularity threads, we reduce the

dynamic scheduling overhead by reducing context switches.

Two inherent problems with concurrent programming are *non-determinate behavior* and *deadlock*. Deadlock occurs when execution halts prematurely. Non-determinate behavior occurs when different executions of the same program yield different results, and is often due to the use of a shared resource by multiple threads without proper synchronization. Proper synchronization between threads can guarantee determinate behavior, but may reduce parallelism and introduce deadlock. The Process Network model of computation [4,5] captures concurrency and parallelism, provides for correctness and liveness, and guarantees determinate execution regardless of the scheduling algorithm used. Dynamic scheduling will execute the Process Network in bounded memory if a bounded memory implementation exists [6].

In this paper, we present a framework for developing scalable software implementations of high-performance signal processing systems. The framework, which is an extension of bounded memory Process Networks, models the concurrency and parallelism in these systems. We implement the framework in C++ using POSIX threads to obtain low overhead, high performance, and scalability. We evaluate the framework by prototyping a 4-GFLOP three-dimensional digital interpolation beamformer on a Sun Ultra Enterprise workstation. For this application, we achieve near-linear speedup over 1 to 12 processors, and exceed real-time throughput constraints using 12 UltraSPARC-II processors running at 336 MHz. The C++ implementation of this framework is available at

http://www.ece.utexas.edu/~allen/PNSourceCode/

2. Beamforming

A high-resolution sonar generally employs an array of underwater sensors along with a *beamformer* to determine from which direction a sound is coming [7]. The sensor element outputs are combined to form multiple narrow *beams*, each of which “looks” in a single direction and is insensitive to sound in neighboring directions. Time-domain beamforming is realized by weighting, delaying, and summing the outputs of the sensor array. Beamforming time delays are determined by geometrically projecting the elements of the sensor array onto a plane that is perpendicular to the maximum response direction for the desired beam.

In a digital beamformer, quantization of the time delays can distort the beam pattern. *Digital interpolation beamforming* achieves finer time delay quantization by interpolating the sampled sensor data, thereby reducing quantization error but increasing computation. The design of the interpolation filter controls the beam degradation caused by interpolation [7]. Fig. 1 shows a digital interpolation beamformer.

Interpolation beamforming can be modeled as a sparse FIR filter that performs upsampling, lowpass filtering, and beamforming in one pass over the data. Compared to upsampling followed by beamforming, this one-pass method substantially reduces memory bandwidth by eliminating the upsampled data stream, but requires more computation. In this paper, the one-pass method increases the number of MACs by 40%,

but reduces the memory bandwidth by 90% by eliminating more than 500 Mb/s for accessing upsampled sensor data. Because digital interpolation beamformers can be modeled as a sparse FIR filter, they can be specified as consistent Synchronous Dataflow graphs [8], which can always be statically scheduled in bounded memory. Therefore, a bounded memory implementation of digital interpolation beamformers always exists.

3. Design Framework

In Kahn Process Networks [4,5], concurrent processes are connected by unidirectional FIFO queues to form a network. The model uses a directed graph notation, in which each node represents a process and each edge represents a queue. Each node may have any number of incoming or outgoing queues, and may communicate to other nodes only with these queues. This model is natural for describing the streams of data samples in a signal processing system. Fig. 2 shows a simple Process Network program.

A Process Network program is *determinate* [4] – the results produced on all queues are the same for every possible execution order, including concurrent execution. A node suspends execution when it attempts to consume data from (or check the number of unconsumed samples in) an empty queue. However, a node is never suspended for producing data because queues can be infinitely long. Termination and total stream lengths are properties of the program and do not depend on the execution order. However, the number of unconsumed samples that can accumulate on queues does depend on the execution order [6].

Infinitely large queues are undesirable, as execution in bounded memory is necessary for any practical implementation. Determining if a Process Network can be scheduled in bounded memory is undecidable in finite time, so the scheduler must work dynamically as the program executes. Parks [6] developed a scheduling policy that will yield a bounded execution if one exists. Parks' policy suspends a node for writing to a full queue, a.k.a. artificial deadlock. On artificial deadlock, the scheduler enlarges the queue that caused artificial deadlock. This bounded scheduling policy is well-suited for an implementation in threads.

Like Process Networks, *Computation Graphs* [9] is a determinate model consisting of nodes connected by unidirectional FIFO queues. The queues have a firing threshold – a node may require more samples in a queue than it will consume upon firing. By choosing the right firing thresholds, all data samples can be managed by the queues, since none of the samples need to be buffered in the node. This approach efficiently models operations on overlapping data streams, such as FIR filters.

We add queue firing thresholds to the Process Network model to form the Computational Process Network model. The combined model is determinate. The new model provides a framework for implementing scalable software that always runs in bounded memory if a bounded implementation exists. The nodes operate directly on queue memory, which avoids copying of data into a node. Beamformers (and many other signal processing algorithms) can be efficiently implemented as Computational Process Networks.

4. Computational Process Network Implementation

Our implementation of the Computational Process Networks model, which uses C++ and POSIX threads, is intended for computationally intensive algorithms on large symmetric multiprocessing workstations. Each node corresponds to a thread. These multiple threads can run concurrently when the program has parallelism, and thus can take advantage of multiple processors. The cost of firing a node should be much larger than the cost of a lightweight thread context switch (about 10 μs for Solaris). However, if the computation of a node is too costly, then the node may need to be divided into smaller pieces in order to run in real time. Generally, a trade-off exists between overhead, latency, and parallelism.

The queues are optimized for data-intensive applications. They compensate for the lack of modulo addressing [10] (for circular buffering) in general purpose processors. They use C++ templates so that they can queue any data type. In order to prevent unnecessary copying of data, our queue implementation enables nodes to access contiguous blocks of data directly from queue memory by means of pointers. Using pointers reduces overhead. The use of firing thresholds enables the decoupling of communication and computation in the queues and nodes, respectively. Fig. 3 shows the C++ code for a sample Process Network node. The threshold and count values are not static, and may change on each queue transaction. This interface obeys Parks' rules for bounded scheduling of Process Networks.

A queue implements its apparent circular addressing by mirroring the beginning of the queue's data region (up to the maximum threshold) at 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 illustrates the queue's mirroring implementation. On Unix operating systems such as Sun Solaris, the virtual memory manager can be used to manage the mirroring in hardware by mapping the same physical memory pages to multiple virtual addresses.

Artificial deadlock detection is not currently implemented. For the class of bounded real-time problems that we are currently targeting, deadlock detection is not necessary, because queues can be instantiated with a known deadlock avoiding size. However, deadlock detection may be added in the future.

5. Prototype of a 4-GFLOP 3-D Sonar Beamformer

Using our framework implementation in Sun Solaris, we prototype a 4-GFLOP 3-D sonar beamformer. Fig. 5 shows a block diagram of the beamformer. The 800 element sensor array consists of 80 elements horizontally by 10 elements vertically. Digital array data comes from four 40 Mb/s telemetry links. The vertical beamformer computes three sets of vertical responses for 80 logical horizontal elements, or *staves*. Then, three horizontal beamformers compute 61 beams each.

For each time sample, the vertical beamformer computes one dot product per stave per output set. Although this has been described as a 500 MFLOPS algorithm, the algorithm uses integer arithmetic, because the sampled sensor element data is in an integer format. After calculation, this stage performs integer-to-float conversion, and interleaves the telemetry links for the following horizontal beamformer stages. The UltraSPARC-II vertical beamformer kernel implementation uses the Visual Instruction Set (VIS) [11]. The highest precision (and slowest) mode of VIS was required – 16-bit by 16-bit multiplication and 32-bit accumulation. In this mode, the peak MAC performance is only one operation per cycle (on average), compared to two operations per cycle for floating-point. Despite the peak performance numbers, the VIS version is more than twice as fast as the floating-point version, because the overhead in the integer-to-floating point conversion is effectively eliminated [12]. Memory latency hiding techniques, including hand-coded software data prefetching [13], were utilized to increase vertical beamforming performance.

Each horizontal beamformer stage performs interpolation beamforming with single-precision (32-bit) floating-point numbers. For each vertical weighting, 61 beams are formed from 80 stave outputs, which requires 1200 MFLOPS. The UltraSPARC-II horizontal beamformer kernel utilizes highly optimized C++ with loop unrolling. The compiler did not generate software prefetching instructions. The best performance was achieved by making multiple passes through the same data and calculating only a subset of the result each pass [12]. On each pass, the data and a subset of the coefficients fit into the on-chip cache.

We connect these highly optimized beamforming kernels using the Computational Process Network model according to Fig. 5, where each beamformer node corresponds to a Computational Process Network node. Because both kernels require more than one processor to execute in real time, we parallelize the beamforming tasks in time by having each node manage a thread pool, which is a common workstation multiprocessor programming strategy [3]. In a thread pool, the manager task creates a fixed number of worker threads at initialization time which survive for the duration of the program. When the manager has work to do, it places a request on a queue. Workers remove requests from the queue and process them. When a beamformer node fires, it queues a request to each of several worker threads, and then blocks on each worker thread, to wait for completion. The number of worker threads can easily be changed as the processing performance requires. This method is used for both horizontal and vertical beamformer nodes.

6. Results

This section benchmarks different multiprocessor beamformer implementations on a Sun Enterprise 4000 server with 12 336-MHz UltraSPARC-II processors, 3 GB of RAM, and 4 MB of external cache per processor. We used the Sun Solaris 2.6 operating system, with threads executing in the “real-time” class. All reported execution times represent an average over 100 trials to calculate 2.6 s of data. Data is processed in blocks of 1024 samples per thread. We ensure that incoming data was not previously cached.

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 50 staves on average contribute to each beam. Calculation of one sample requires over 12,000 single-precision floating-point operations, in addition to over 6000 index lookups for the sparse FIR filter time index. The benchmark requires 3.2 billion floating-point operations, and consumes about 85 Mb of input data.

First, a reference benchmark for the horizontal beamformer kernel was performed. The first row of Table 2 displays the result of 443.6 MFLOPS for this kernel. At peak performance, the UltraSPARC-II processor can execute two floating-point operations per clock cycle, for 672 MFLOPS at 336 MHz. Despite the index lookups, the beamforming kernel routine is operating at 1.32 floating-point operations per clock cycle, which is 66% of the peak. The remaining rows of Table 2 show the results for batch-mode thread pool beamformer implementations, each with a varying number of threads. The “speedup” and “percent utilization” columns use the kernel in the first row as a reference. Scaling performance for the horizontal beamformer is very good. The real-time goal for horizontal beamforming at 1200 MFLOPS is easily met with three 336 MHz processors.

6.2. Vertical Beamformer Performance

The vertical beamformer calculates three sets of 80 logical horizontal elements (staves) using 10 vertical elements each, for a total of 800 elements. Although a mere 4800 operations per sample is required, the incoming data must be converted from integer to floating-point format. This benchmark requires approximately 1.3 billion operations, and consumes 400 Mb of input data. As described in Section 5, this kernel is performed with integer math using VIS. Despite this, we use MFLOPS for the sake of comparison.

The vertical beamformer kernel benchmark is again used as a reference. The peak performance for this high-precision mode of VIS is one MAC operation every 2 clock cycles (on average). The first row of Table 3 displays the result of 311.7 MFLOPS for the vertical beamformer kernel, which is nearly 93% of peak performance. Table 3 shows the results for vertical beamformers with an increasing number of threads, referenced to the kernel case in the first row. Because of the high memory bandwidth consumed by this algorithm, the scaling performance of the vertical beamformer is poor. Fortunately, only 2 threads are required to surpass the real-time goal of 500 MFLOPS.

6.3. Computational Process Network Beamformer Performance

For the prototype 3-D beamforming system depicted in Fig. 5, we evaluate the Computational Process Network implementation versus a batch-mode thread pool implementation. This analysis is performed on eight 336-MHz UltraSPARC-II processors. Both implementations allocate and lock memory for input and

output data (nearly 500 Mb) so that processing performance can be measured. The thread pool implementation also allocates and locks memory for the results of each stage.

The thread pool implementation first calculates three sets of stave data from the element data, using a vertical beamformer with eight threads. Then, three eight-thread horizontal beamformers sequentially calculate beam results from stave data. These eight threads are matched to the number of processors for best 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 three horizontal beamformers from Sections 6.1 and 6.2 above.

On eight processors, the Computational Process Network beamformer performs about 8% faster, has lower latency, and uses 21% less memory than the thread pool beamformer. The latency and memory usage are lower because the communication channels need only be large enough to prevent artificial deadlock. Because the Computational Process Network beamformer is stream oriented, it is better suited for real-time execution. Four threads were allocated to each beamformer node, which exceeds the real-time requirement. All nodes are independently operating all of the time, as the flow of data permits.

Although the memory, latency, and scheduling issues could be better addressed in the thread pool implementation, these issues are automatically resolved when using the Computational Process Network model. An additional advantage of the Computational Process Network implementation is that the program is scaled by the operating system according to the number of available processors. For best performance, the thread pool beamformer requires knowledge of the number of processors. This is not the case for the Computational Process Network beamformer.

The Computational Process Network beamformer scales without any change to the executable. If there were more processors than the 16 total worker threads, then those extra processors would not be utilized. However, those processors are not needed for the system to meet its real-time goal. Table 5 shows scaling results for the same Computational Process Network beamformer executable, running on a 12-processor Sun. Solaris system administration tools were used to disable processors so that this test could be performed. The Process Network beamformer program scales almost linearly from 1 to 12 processors, and the slowdown on one processor is under 0.15% compared to kernel performance. Our real-time goal of about 4.2 GFLOPS is met with 10.5 336-MHz UltraSPARC-II processors. On 12 processors, it operates with 14% to spare. This Computational Process Network beamformer is designed with the parallelism to linearly scale to about 16 processors.

7. Conclusion

We consider the use of commodity multiprocessor workstations for implementing real-time data-intensive systems. A workstation solution reduces development cost and time, and offers better software portability, reconfigurability, maintainability, and upgradability than custom embedded COTS solutions. We

describe a framework which extends the formal Process Network model. This Computational Process Network framework captures concurrency, provides for correctness and determinacy, and will execute in bounded memory if it is possible.

We implement the framework in C++ using lightweight real-time POSIX threads. This low-overhead, high-performance framework removes the dependency of the software on the hardware topology. Systems using this framework scale automatically, because the operating system dynamically schedules the threads to balance the workload among available processors. When the workstation is both the development platform and the target architecture, development can be performed on less powerful workstations, multiple sets of development hardware are available, and the design environment can be deployed with the design.

To evaluate the framework implementation, we prototype a 4-GFLOP 3-D beamformer. We benchmark the beamformer on a Sun Ultra Enterprise Server with 12 336-MHz UltraSPARC-II processors. On one processor, the horizontal beamformer kernel delivers 440 MFLOPS and the vertical beamformer kernel delivers 310 MFLOPS. The 4-GFLOP 3-D software beamformer scales nearly linearly from 1 to 12 processors, and reaches real-time performance with 14% to spare using 12 processors.

References

[1] P. Lapsley, "NSP Shows Promise on Pentium, PowerPC," *MicroProcessor Report*, 1995.

[2] L.-R. Dung, V. Madisetti, and J. Hines, "Model-Based Architectural Design and Verification of Scalable Embedded DSP Systems - A RASSP Approach," *Proc. IEEE Workshop on VLSI Signal Processing*, Oct. 30, 1996.

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

[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] T. M. Parks, "Bounded Scheduling of Process Networks," *Technical Report UCB/ERL-95-105*, Ph.D. Dissertation, EECS Department, University of California, Berkeley, CA 94720-1770, Dec. 1995.

[7] 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.

[8] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing," *IEEE Trans. on Computers*, vol. 36, no. 1, pp. 24-35, Jan. 1987.

[9] 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.

[10] P. Lapsley, J. Bier, A. Shoham, and E. A. Lee, *DSP Processor Fundamentals: Architectures and Features*, Berkeley Design Technology, Inc., 1996.

[11] *Visual Instruction Set User's Guide*, Sun Microsystems, 1995.

[12] G. Allen, B. Evans, and L. John, "Real-Time High-Throughput Sonar Beamforming Kernels Using Native Signal Processing and Memory Latency Hiding Techniques," *Proc. IEEE Asilomar Conf. on Signals, Systems, and Computers*, Oct. 25-28, 1999, Pacific Grove, CA, accepted.

[13] D. Callahan, K. Kennedy, and A. Porterfield, "Software Prefetching" *Proc. ACM International Conf. on Architectural Support for Programming Languages and Operating Systems*, Apr. 1991, pp. 40-52.

Table Captions

Table 1: Three generations of low-volume 4-GFLOP sonar beamformers (under 50 units). Estimates are based on 1999 technology. The last column is the third-generation sonar beamformer developed in this paper.

Table 2: Horizontal beamforming benchmark results using 336-MHz UltraSPARC-II processors.

Table 3: Vertical beamforming benchmark results using 336-MHz UltraSPARC-II processors.

Table 4: Process Network vs. thread pool performance results using eight 336-MHz UltraSPARC-II processors.

Table 5: Process Network beamformer scalability vs. kernel performance using 336-MHz UltraSPARC-II processors. The results show near-linear speedup.

Figure Captions

Fig. 1: Digital interpolation beamformer with a digitizing sensor array.

Fig. 2: A simple Process Network program. Processes A and B execute concurrently, and A sends data to B through a one-directional channel (FIFO queue) P.

Fig. 3: C++ source code for a sample Computational Process Network node.

Fig. 4: Mirroring for circularity in the Computational Process Network queue implementation.

Fig. 5: A block diagram of the 3-D sonar beamformer. It is an example of a real-time high-throughput signal processing system. Each beamformer node corresponds to a Computational Process Network node.

	Custom Hardware	Embedded COTS	Commodity Workstation
Development cost	$2000K	$500K	$100K
Development time	24 months	12 months	6 months
Physical Size (m^3)	0.067	0.067	0.089
Reconfigurability	low	medium	high
Software portability	low	medium	high
Hardware upgradability	low	medium	high

Table 1: Three generations of low-volume 4-GFLOP sonar beamformers (under 50 units). Estimates are based on 1999 technology. The last column is the third-generation sonar beamformer developed in this paper.

Implementation	Time (s)	MFLOPS	Speedup	Percent Utilization
kernel	7.252	443.6	(1.000)	(100.00%)
2 thread pool	3.691	871.6	1.965	98.24%
4 thread pool	1.895	1697.5	3.827	95.67%
6 thread pool	1.277	2518.6	5.678	94.63%
8 thread pool	0.973	3306.7	7.454	93.18%

Table 2: Horizontal beamforming benchmark results using 336-MHz UltraSPARC-II processors.

Implementation	Time (s)	MFLOPS	Speedup	Percent Utilization
kernel	4.037	311.7	(1.000)	(100.00%)
2 thread pool	2.082	604.3	1.939	96.94%
4 thread pool	1.128	1115.9	3.580	89.50%
6 thread pool	0.802	1543.9	4.953	82.55%
8 thread pool	0.661	1905.0	6.112	76.40%

Table 3: Vertical beamforming benchmark results using 336-MHz UltraSPARC-II processors.

Implementation	Time (s)	MFLOPS	Mbytes
thread pool	3.607	3024.8	832
Process Network	3.329	3277.3	654

Table 4: Process Network vs. thread pool performance results using eight 336-MHz UltraSPARC-II processors.

Number of CPUs	Time (s)	MFLOPS	Speedup	Percent Utilization
1	25.829	422.4	0.999	99.86%
2	13.047	836.2	1.977	98.85%
4	6.561	1662.7	3.931	98.28%
6	4.412	2472.8	5.846	97.44%
8	3.329	3277.3	7.748	96.85%
10	2.689	4056.8	9.591	95.91%
12	2.287	4769.2	11.276	93.96%

Table 5: Process Network beamformer scalability vs. kernel performance using 336-MHz UltraSPARC-II processors. The results show near-linear speedup.

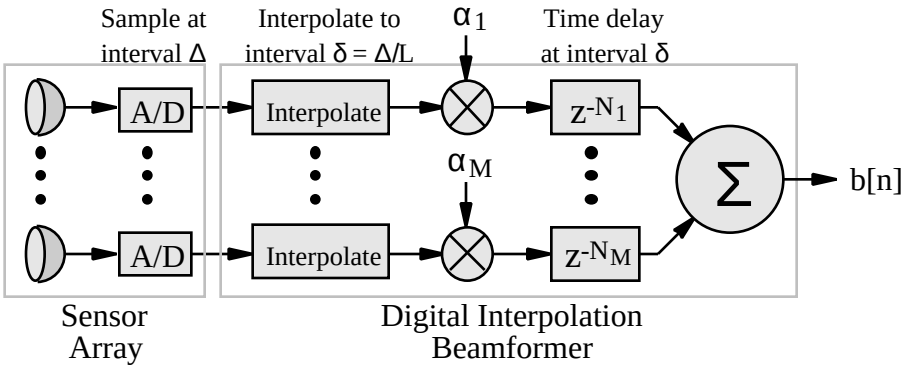

Fig. 1: Digital interpolation beamformer with a digitizing sensor array.

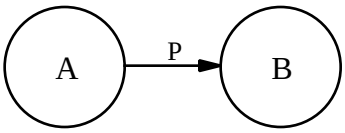

Fig. 2: A simple Process Network program. Processes A and B execute concurrently, and A sends data to B through a one-directional channel (FIFO queue) P.

```
typedef float T;
while (true) {
    // blocking calls to get in/out data pointers
    const T* inPtr  = inputQ.GetDequeuePtr(inThreshold);
    T*       outPtr = outputQ.GetEnqueuePtr(outThreshold);

    DoComputation( inPtr, inThreshold, outPtr, outCount );

    // complete node transactions
    inputQ.Dequeue(inCount);
    outputQ.Enqueue(outCount);
}
```

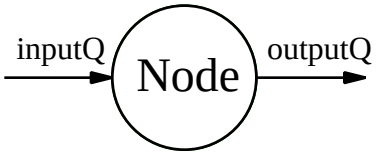

Fig. 3: C++ source code for a sample Computational Process Network node.

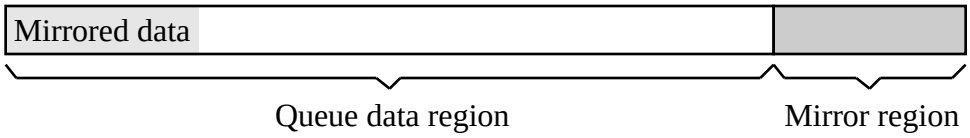

Fig. 4: Mirroring for circularity in the Computational Process Network queue implementation.

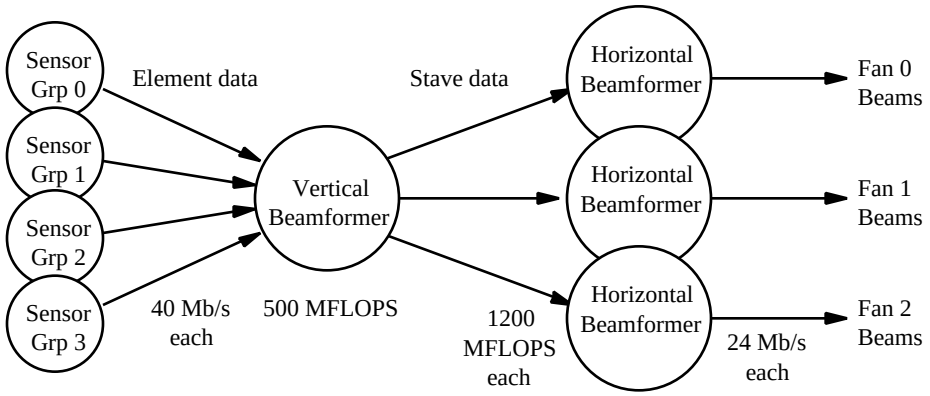

Fig. 5: A block diagram of the 3-D sonar beamformer. It is an example of a real-time high-throughput signal processing system. Each beamformer node corresponds to a Computational Process Network node.