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 an approach for realizing real-time data-intensive systems on commodity multiprocessor workstations. We develop scalable software using a 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.**

***Index Terms* – beamforming, high-performance computing, models of computation, multiprocessor programming, native signal processing, process networks, real-time systems, scalable software**

I. 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 approach, one 4-GFLOP sonar beamformer requires about 100 80-MFLOPS Analog Devices SHARC digital signal processors. Although using commercial off-the-shelf (COTS) components reduces hardware development cost and time over custom parallel hardware, software development and system integration and testing are difficult and costly. Partitioning the algorithm among several dozen processors while guaranteeing synchronization is difficult, and debugging the software is difficult due to low observability of program state. Once complete, the software developed for a custom COTS configuration is closely tied to the hardware topology, thereby making code reuse or in-the-field configuration changes difficult.

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 performance. For a workstation beamformer, development cost and time can be significantly reduced compared to that for custom hardware or an embedded COTS system with the same level of performance. By implementing beamformers in software on commodity high-performance workstations, we take advantage of advances in commercial workstations, thus sharing hardware development costs with the high-volume workstation server market. We also reduce software development efforts because mature high-volume operating systems and tools can be used. Workstations also offer better software portability, and better hardware upgradability and maintainability, than embedded COTS solutions. Because the development environment and target architecture are the same, the design tools can be deployed with the design for in-the-field changes, thereby making the target system dynamically reconfigurable. Also, some development can be performed on less powerful workstations, which makes multiple sets of target hardware available. Table 1 compares three generations of 4-GFLOPS sonar beamformers.

Modern workstation operating systems dynamically schedule processes to balance the workload among 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) [2] provides a standard thread library, called Pthreads. Pthreads have low overhead, and can be scheduled with a fixed real-time priority. By partitioning a software beamformer using large granularity threads, we reduce the dynamic scheduling overhead by reducing context switches.

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.

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. Synchronization between threads can guarantee determinate behavior, but can reduce parallelism and introduce deadlock. The Process Network model of computation [3,4] captures concurrency and parallelism, provides for correctness and liveness, and guarantees determinate execution regardless of the scheduling algorithm used. Dynamic scheduling based on the availability of data allows execution in bounded memory [5].

In this paper, we present a formal framework for developing scalable software implementations of high-performance signal processing systems, such as sonar beamformers. The framework models the concurrency and parallelism in these systems using the bounded memory Process Network model. 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 will be available at:

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

II. 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 [6]. 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. The 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 system, quantization of the time delays can distort the beam pattern. *Digital interpolation beamforming* achieves the desired time-delay quantization by interpolating the sampled sensor data, reducing quantization error at the expense of increased computation. The beam degradation caused by interpolation is controlled by the design of the interpolation filter [7,8]. Fig. 1 shows a digital system with an interpolation beamformer.

Interpolation beamforming can be modeled as a sparse FIR filter. This model performs upsampling, low-pass filtering, and beamforming in one pass over the data. While this "one-pass" method increases the number of multiply-accumulates performed, it substantially reduces memory bandwidth, thereby making it more suitable for workstation applications. For the interpolation beamformers presented in this paper, the number of MACs is increased by 40%, but the memory bandwidth is reduced by 90% by eliminating more than 500 Mb/s for writing and reading the upsampled sensor data. Because digital interpolation beamformers can be modeled as a sparse FIR filter, they can be specified as Synchronous Dataflow graphs [9]. Therefore, a bounded memory implementation of digital interpolation beamformers always exists.

III. Design Framework

In Kahn Process Networks [3,4], concurrent processes are connected by unidirectional FIFO queues to form a network. The model uses a directed graph notation, where 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 [3] – the results produced on all queues are the same for every possible execution order, including concurrent execution. In this model, a node suspends execution when it attempts to consume data from an empty queue, and cannot detect the presence of data in a queue. However, queues are of infinite length and so a node is never suspended for producing data. Termination and total stream lengths are properties of the program and do not depend on the execution order. However, the number of unconsumed tokens that can accumulate on queues does depend on the execution order [5].

Infinitely large queues cause obvious problems; execution in bounded memory is necessary for any practical implementation. The problems of determining whether a Process Network will terminate, and whether it can be scheduled in bounded memory are undecidable. In this context, the scheduler must work dynamically, as the program executes. Parks [5] developed a dynamic scheduling policy that will yield a bounded execution, if one exists. Parks' policy suspends nodes for writing to full a queue, and grows a queue size if artificial deadlock is reached. This bounded scheduling

policy has the desired behavior for all types of Process Network programs, and is well-suited for implementation with threads.

Karp and Miller [10] developed *computation graphs*, a restricted determinate model similar to Process Networks. Computation graphs can be statically scheduled, because the questions of termination and boundedness are decidable in finite time. A unique feature of computation graphs is that queues have a firing threshold. That is, a node may require more tokens on a queue than it will consume upon firing. This easily models algorithms which operate on an overlapping stream of data (such as FIR filters).

Our framework extends bounded Process Networks by borrowing and extending the concept of a firing threshold from computation graphs. This allows nodes to operate directly out of queue memory, which avoids the copying of data into a local buffer. The beamforming algorithms presented in this paper (and many other signal processing algorithms) can be modeled as computation graphs, but are dynamically scheduled with Process Networks 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.

IV. Process Network Implementation

Our C++ implementation is intended for computationally intensive algorithms on large symmetric multiprocessing workstations. Although our implementation 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 large granularity nodes – such as an FFT node, a filter node, or a beamformer node. The graphs drawn using this methodology are essentially block diagrams. 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.

Each node in the Process Network program corresponds to a thread. These multiple threads can run concurrently when the program has parallelism, and thus can take advantage of multiple processors. POSIX Pthreads are portable to many different Unix platforms, and can be given fixed real-time scheduling priority in some implementations (e.g. Solaris).

Our Process Network queues are designed and optimized for data-intensive applications, and are intended to compensate for the lack of circular address buffers in general purpose processors. They use C++ templates so that they can queue any type of data. In order to prevent unnecessary copying of data, our queues have firing thresholds and use pointers so that nodes can access contiguous blocks of data directly from queue memory. This reduces overhead, and simplifies node implementation by separating computation from communication. 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.

The queues 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.

V. Prototype of a 4-GFLOP Sonar Beamformer

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

For each time sample, the vertical beamformer computes one dot product per stave per vertical shading set. Although this has been described as a 500 MFLOPS algorithm, note that it is actually performed with 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. For the kernel implementation, the best performance on an UltraSPARC-II processor was obtained by

using the Visual Instruction Set (VIS) [11]. The highest precision (and slowest) mode of VIS was required – 16-bit by 16-bit multiplies and 32-bit accumulates. 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 [14]. Memory latency hiding techniques, including hand-coded software data prefetching [12], were utilized to increase vertical beamforming performance.

Each horizontal beamformer stage performs digital interpolation beamforming with single-precision (32-bit) floating-point numbers. For each vertical weighting, 61 beams are formed from 80 stave outputs, requiring 1200 MFLOPS. The horizontal beamformer kernel implementation for the UltraSPARC-II processor utilizes highly optimized C++ with loop unrolling. The compiler did not generate software prefetching instructions, and attempts to improve performance with hand-insertion were unsuccessful [13]. The best performance was achieved by making multiple passes through the same data and calculating only a subset of the result each pass. On each pass, both the data and a subset of the coefficients fit into the on-chip cache.

We connect these high performance kernels with our process network according to Fig. 5. 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 [14]. 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 increased or decreased as the processing performance requires. This method is used for both horizontal and vertical beamformer nodes. This thread pool can also be modeled as a Process Network [15].

VI. Results

This section benchmarks different multiprocessor beamformer implementations. These benchmarks were performed on a Sun Ultra Enterprise 4000 with twelve 336-MHz UltraSPARC-II processors and 3 GB of RAM. We used the Sun Solaris 2.6 operating system, with threads executing in the “real-time” class. All results are determine as the average time over 100 trials to calculate 2.6 s of data. We ensure that incoming data was not cached before the benchmarks were performed.

A. 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 location of the proper time index in the sparse FIR filter model. The benchmark requires 3.2 billion floating-point operations.

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 2 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.

B. 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, consuming 400 Mb of element data. As described in Section 5, this kernel is performed with integer math using the Visual Instruction Set. Despite this, we still 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.

C. Process Network Beamformer Performance

For the prototype 3-D beamforming system depicted in Fig. 5, we evaluate the 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 8 threads. Then, three 8-thread horizontal beamformers sequentially execute, calculating beam results from the 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 3 horizontal beamformers from Sections 6.1 and 6.2 above.

On eight processors, the Process Network beamformer performs about 8% faster than the thread pool beamformer. It also has lower latency and uses less memory – the communication channels in the network need only be large enough to prevent artificial deadlock. In this comparison, the resulting memory size is reduced by over 21 percent. Because it is more "stream" oriented, it is better suited for real-time execution. Four threads were allocated to each beamformer node, which is more than enough to keep up with 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 advantages come automatically when using the Process Network model of computation. An additional advantage of the Process Network implementation is that the program is scaled by the operating system according to the number of available processors. The thread pool beamformer was written with knowledge of the number of processors. Utilizing more processors requires more worker threads at each pool, and arbitrarily creating large numbers of worker threads causes unnecessary overhead.

The Process Network beamformer continues to scale without any change to the executable. If there were more processors than the sum of all worker threads, those processors would not be utilized. However, those extra processors are not needed for the system to meet its real-time goal. Table 5 shows scaling results for the same Process Network beamformer executable, running on a twelve-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. Our real-time goal of about 4.2 GFLOPS is met with 10.5 336-MHz UltraSPARC-II processors. On twelve processors, it operates with 14% to spare. This Process Network beamformer is designed with the parallelism to scale linearly to about 16 processors.

VII. Conclusion

We consider the use of commodity multiprocessor workstations for implementing real-time data-intensive systems, such as sonar beamformers. 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 formal framework based on the Process Network model, which captures concurrency, provides for correctness and determinacy, and can guarantee execution in bounded memory.

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 as, the operating system dynamically schedules the threads to balance the workload among the 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 electronic design automation tools 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. The kernels have been highly optimized for the UltraSPARC-II processor. 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 linearly, 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] The IEEE Portable Applications Standards Committee Web Page: http://www.pasc.org/

[3] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," *Information Processing*, pp. 471-475, Stockholm, Aug. 1974.

[4] G. Kahn and D. B. MacQueen, "Coroutines and Networks of Parallel Processes," *Information Processing*, pp. 993-998, Toronto, Aug. 1977.

[5] 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.

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

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

[9] 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.

[10] 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.

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

[12] 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.

[13] G. Allen, "An Evaluation of Native Signal Processing on the UltraSPARC-II with Sonar Beamforming Algorithms," Final Report for EE382M Computer Performance Evaluation and Benchmarking, Dept. of Electrical and Computer Engineering, The University of Texas, Austin, TX 78712-1084, Dec. 1998. http://www.ece.utexas.edu/~allen/EE382M-F98

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

[15] G. Allen, *Real-Time Sonar Beamforming on a Symmetric Multiprocessing UNIX Workstation Using Process Networks and POSIX Pthreads*, Master's Report, Dept. of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX 78712-1084, Aug. 1998. http://www.ece.utexas.edu/~allen/MSReport/

	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.

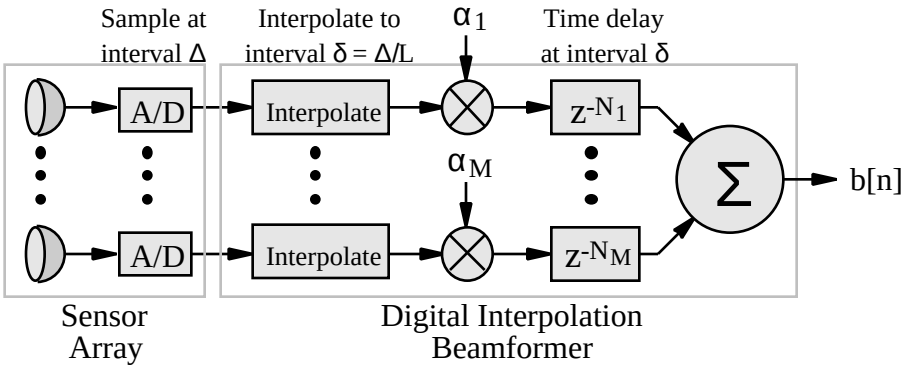

Fig. 1: Digital interpolation beamformer with a digitizing sensor array.

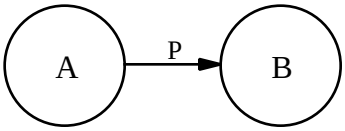

Fig. 2: A simple Process Network program. Processes A and B execute concurrently, and A sends data to B through a one-directional channel (FIFO queue) P.

```
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);
}
```

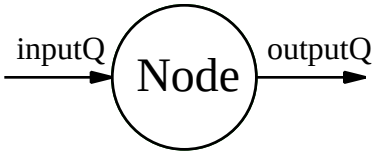

Fig. 3: C++ source code for a sample Process Network node.

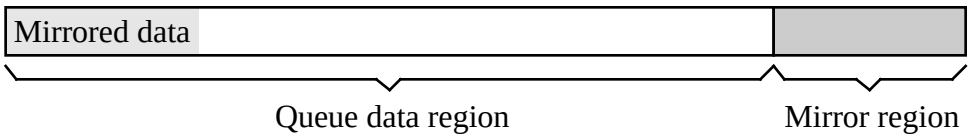

Fig. 4: Mirroring for circularity in the ThresholdQueue implementation.

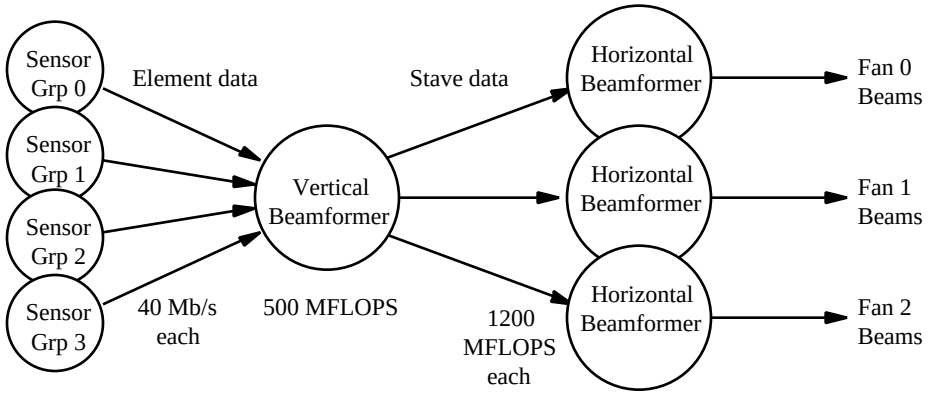

Fig. 5: A block diagram of the beamformer.

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.