Presented at the Thirty-Second Asilomar Conference on Signals, Systems, and Computers - November 1998

Real-time Sonar Beamforming on a Unix Workstation Using Process Networks and POSIX Threads

Gregory E. Allen
Applied Research Laboratories:
The University of Texas at Austin
Austin, TX 78713-8029
gallen@arlut.utexas.edu

Brian L. Evans and David C. Schanbacher
Dept. of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712-1084
{bevans,schanbac}@ece.utexas.edu

Abstract

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 hardware beamformer.

1. Introduction

Sonar beamforming algorithms can require on the order of billions of multiply-accumulates (MACs) per second, and therefore have traditionally been implemented in custom hardware. Current symmetric multiprocessing workstations post benchmarks that meet these capabilities at a fraction of the development and manufacturing costs of a custom hardware solution. However, conventional implementations of the UNIX operating system have not been capable of deterministic real-time performance.

The Portable Operating System Interface (POSIX) is a recent standard with the goal of providing source-code portability across many different platforms. POSIX extensions provide support for real-time applications on UNIX workstations. One such extension, the POSIX Pthread library, provides independent "lightweight" flows of control which can execute on multiple processors.

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. Software development in a workstation environment is generally easier than in a custom embedded hardware environment due to the availability of better affordable development and debugging tools. Workstations also offer better portability, upgradability, and maintainability than custom hardware solutions. In order to facilitate implementation of computationally intensive systems, a reliable formal design methodology is needed for organizing and developing real-time multiprocessor software.

The Process Network [1, 2] model of computation, which is a superset of dataflow, captures concurrency and parallelism in signal processing systems. Implementing this model with Pthreads gives a low-overhead, high-performance, scalable framework. Pthreads are dynamically scheduled by the operating system, and symmetric multiprocessing efficiently utilizes multiple processors.

The goal is to implement a high-resolution multi-fan three-dimensional digital interpolation beamformer which runs in real time on a Unix workstation. This is realized by performing a design space exploration of software beamforming implementations, modeled as a Process Network.

2. Beamforming

High-resolution sonars generally consist of an array of underwater sensors along with a *beamformer* to determine from which direction a sound is coming. The sensor element outputs must be 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 an array of transducers. For M transducers each receiving a signal $x_m(t)$, the output of a single beam can be calculated by

$$b(t) = \sum_{m=1}^{M} \alpha_m \cdot x_m(t - \tau_m)$$

where α_m is the shading coefficient for the mth sensor, and

G. Allen was supported by the Independent Research and Development Program at Applied Research Laboratories: 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.

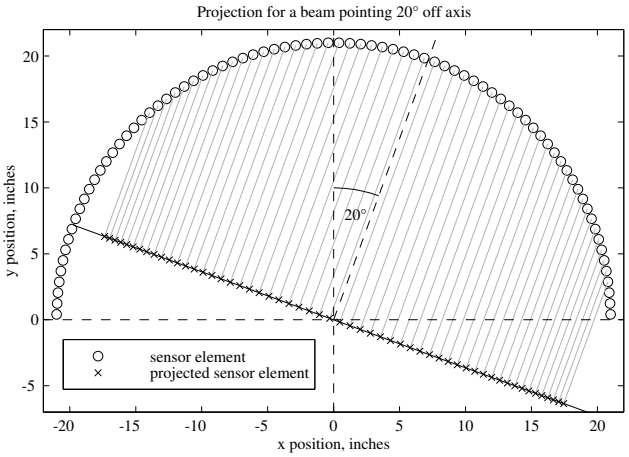

Fig. 1: Projection of sensor elements from a semi-circular array

τ_m is the required time delay for the mth sensor.

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. 1 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 also substantially reduces computation.

2.1. Digital interpolation beamforming

In a digital system, the time delays must be quantized to the nearest sample, which perturbs the beam pattern. Digital interpolation beamforming uses interpolation of the receiver signals to achieve more precise time delay resolution, thus reducing the quantization error at a cost of more computation. Beam degradation introduced by interpolation is controllable and quite small for an interpolation filter of modest design [4].

Fig. 2 shows a digital system with an interpolation beamformer. The sampling interval needed to satisfy the Nyquist criterion is Δ. Digital interpolation is performed to the interval δ, where $\Delta = L\,\delta$, and L is an integer larger than one. Now time delays are quantized to integer multiples of δ, i.e., $\tau_m = N_m\,\delta$.

Modeling interpolation beamforming 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 is $C = PS$, where S is the number of sensors used to calculate each beam, and the sparsity is $1-C/K$. Note that in this model, the digital interpolation lowpass filter is an FIR filter with an impulse response of length $K = LP$.

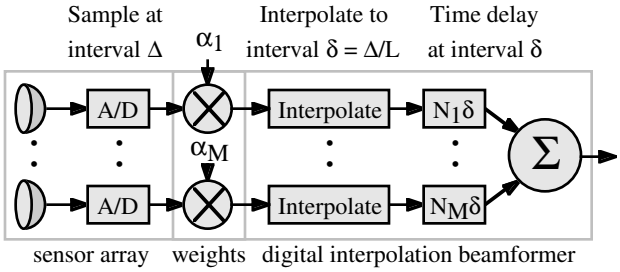

Fig. 2: Digital interpolation beamformer with digitizing sensor array

$$\begin{bmatrix}\text{stave data}\end{bmatrix} \times \begin{bmatrix}\text{beam} & & \text{beam}\\ 1 & \cdots & B\\ \text{coefs} & & \text{coefs}\end{bmatrix} = \begin{bmatrix}\text{beam data}\\ \text{(1 sample)}\end{bmatrix}$$

(1 by K) (K by B) (1 by B)

Fig. 3: 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. 3 shows the matrix operations necessary to calculate B beams from the input data stream.

2.2. Vertical beamforming

For the sensor array utilized in this paper, vertical beamforming (staving) must be performed before digital interpolation (horizontal) 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. Additionally, the vertical beamformer converts the data from integer to 32-bit floating-point format.

These beamforming algorithms have an extremely high degree of parallelism, which can be exploited by using the Process Network model of computation.

3. Process Networks

In the process network 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. 4 shows a simple process network program, in which processes A and B are connected by a communication channel, P.

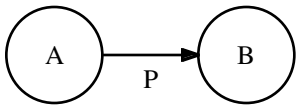

Fig. 4: A process network program

Process nodes may have any number of incoming or outgoing queues, and may communicate only via these queues. A node suspends execution when it attempts to consume data from an empty queue. However, a node is never suspended for producing data, so 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.

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. Parks [3] developed dynamic scheduling rules that 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*, increase the capacity of the smallest full queue until the producer associated with it can fire.

Artificial deadlock is the case where execution has stopped because processes are blocked writing to full channels. This bounded scheduling policy has the desired behavior for all types of programs. 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.

4. Implementation

Our implementation of Process Networks is intended for computationally intense 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.

Each node of a Process Network program corresponds to a different thread. These multiple threads can run concurrently when the program has parallelism, and thus can take advantage of multiple processors. Pthreads provide high performance in a low-overhead environment, are source-code compatible with many versions of Unix, and can be given fixed real-time scheduling priority.

We use nodes of fairly large granularity, where the cost of firing a node is much larger than the cost of a light-weight thread context switch. However, if a node is too computationally costly, it must be divided into smaller pieces in order to run in real time. Generally, there is a tradeoff between overhead and latency.

The queues which connect the process nodes are optimized for data-intensive applications, and are intended to make up for the lack of circular address buffers in general purpose processors. A design goal 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 implementations that interface to these queues.

The queues implement their apparent circular addressing by mirroring the beginning of the queue's data region (up to some 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. 5 illustrates this mirroring in the queue implementation.

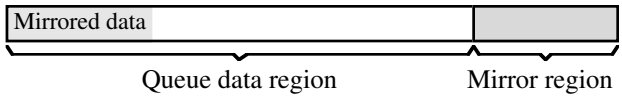

Fig. 5: Queue implementation

These queues have 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 queues from having to copy data at all. 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.

Fig. 6 shows a block diagram of the full beamforming

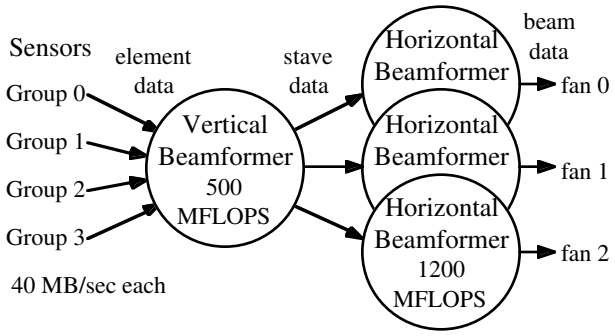

Fig. 6: Beamformer block diagram

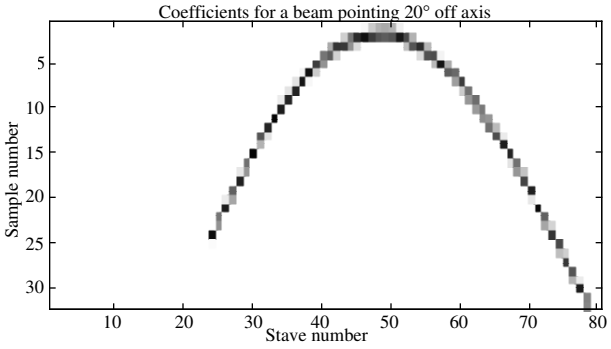

Fig. 7: Coefficients for one beam

system, and the corresponding nodes in the Process Network implementation. The vertical beamformer forms 3 sets of 80 staves from 10 vertical elements each.The horizontal beamformers each form 61 beams from the 80 staves, using a 2-point interpolation filter.

Matlab was used to generate and test beam coefficients, and to verify the results. Each horizontal beamformer performs interpolation beamforming, with 32-bit floating-point numbers. When this operation is modeled as a sparse FIR filter, the filter length is 2560 coefficients, 96% of which are zero.

Fig. 7 shows a sample set of coefficients used. Although organized as a 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 shape of the array is clearly visible in the coefficients.

These beamforming algorithms are 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 [5], using a minimal amount of memory, with minimum latency. Indeed, this method is frequently employed in custom hardware designs that use digital signal processor (DSP) computing engines. However this "DSP-minded" approach suffers from poor cache utilization on a workstation, resulting 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 through superior 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. Within the kernel beamforming routines, care must be taken to heed this memory usage limit.

Because a single thread cannot achieve real-time performance, a beamformer node must divide the task. In order to divide this calculation in time without copying data, a horizontal beamformer node manages multiple worker nodes. The number of worker nodes can easily be increased or decreased, as the processing performance requires. This method is similar to a thread pool, which is a common workstation multiprocessing tool [6].

5. Results

Benchmarks were performed on a Sun Ultra Enterprise 4000 with 8 UltraSPARC-II processors running at 336 MHz. Solaris 2.6 was the operating system 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, so that artificially elevated results would not occur.

Beamforming kernel performance and scalability was measured using traditional thread pools. Fig. 8 plots the results for the horizontal and vertical beamforming kernels.

The execution times (dotted) are used to calculate the useful beamforming MFLOPS (solid). Despite index lookups, the horizontal 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. The performance of horizontal beamforming kernel scales fairly well with additional threads. The real-time goal of beamforming at 1200 MFLOPS is met with 4 threads, where over 385 MFLOPS on each of 4 (336 MHz) processors is delivered.

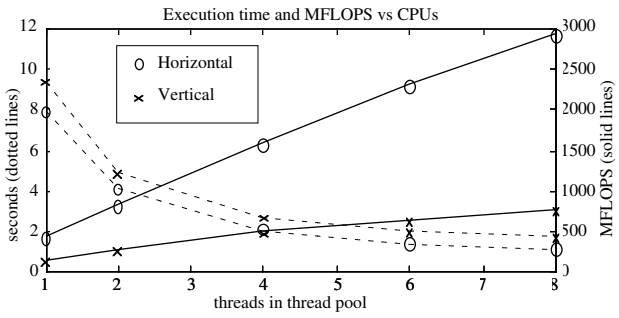

Fig. 8: Beamforming kernel results

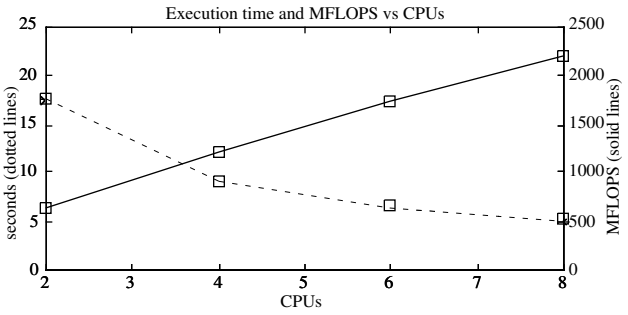

Fig. 9: Process Network beamformer scaling

Because the vertical beamformer accounts for less than 12% of the system's required computation, less time has been spent in its optimization. The vertical beamformer performance is currently unimpressive at 135 MFLOPS, which 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, which currently requires about 5 integer operations per point. 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.

We compare the performance of the full Process Network beamforming system depicted in Fig. 6 with the thread-pool implementations. The thread-pool beamforming system loads all input data into memory, allocates memory for results, and calculates from memory to memory using pools of 8 threads. This batch-mode system uses over 800 Mb of memory for data alone. 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 using 8 threads.

The Process Network beamformer achieves within 1% of the same result, taking just over 5 seconds to process 2.6 seconds of data. This is slightly better than half of the real-time goal. The Process Network system has distinct advantages. Because it is "stream" oriented, it has lower latency and uses 25% less memory. With real-time input and output devices, this memory savings would be more dramatic. All Process Network nodes are operating all of the time, as the flow of data permits, so the Process Network beamformer program is automatically scaled by the operating system according to the number of available processors.

Fig. 9 shows scaling results for the Process Network beamformer on a varying number of CPUs. This test was performed by disabling CPUs in the 8-processor machine. The beamformer scales fairly well from 2 to 8 processors. 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 performance losses due to additional scaling overhead must also be reduced.

6. Conclusion

We implement computationally intensive sonar beamforming algorithms using Process Networks and Pthreads under the Sun Solaris operating system. 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.

Although further optimization is required in the vertical beamforming kernel, it is feasible for this high-resolution multi-fan 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.

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 allows real-time performance at a substantial savings in development cost and time when compared to a custom hardware solution.

References

[1] G. Kahn, "The semantics of a simple language for parallel programming." *Info. Proc.*, pp. 471-475, Aug. 1974.

[2] G. Kahn and D. B. MacQueen, "Coroutines and networks of parallel processes." *Info. Proc.*, pp. 993-998, Aug. 1977.

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

[4] 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.

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

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

[7] 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, http://www.ece.utexas.edu/~allen/MSReport/, Aug. 1998.