A Comparison of Parallel Workstation Sonar Beamforming Implementations

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

Abstract

We implement a prototype for a 20-GFLOP sonar beamformer with multiple frameworks for parallelism: MPI, Computational Process Networks, and a hybrid of the two. Computational Process Networks is a framework based on the formal Process Network model and Computation Graphs, implemented with lightweight POSIX threads. We benchmark these systems on a Sun Enterprise 4000. On eight UltraSPARC-II processors, we measure speedups of 6.5 for Computational Process Networks, 3.5 for MPI, and 2.9 for the hybrid.

1. Introduction

High-resolution sonar beamforming algorithms require on the order of billions of multiply-accumulates (MACs) per second. Traditionally, custom parallel hardware has been required to implement them in real time. The non-recoverable engineering cost for custom hardware development may make this approach prohibitively expensive because sonar systems are typically not sold in high volumes. 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.

G. Allen was supported by the Independent Research and Development Program at Applied Research Laboratories at The University of Texas at Austin.

For the next generation of sonar beamformers, we advocate the use of commodity multiprocessor workstations, which are capable of native signal processing 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 [1]. 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. For the domain of problems which exceed the capabilities of a single workstation, multiple workstations can be connected together, forming a cluster of workstations.

The Portable Operating System Interface (POSIX) is a recent standard [2] with the goal of providing source-code portability across many UNIX platforms. Implementing the Computational Process Network [3] model with POSIX threads (Pthreads) gives a low-overhead, high-performance, scalable framework. Symmetric multiprocessing (SMP) guarantees efficient utilization of multiple processors, as scheduling of Pthreads is dynamically handled by the operating system. POSIX optionally allows Pthreads to be scheduled with real-time priority.

The Message Passing Interface (MPI) is a recent standard [4] for writing message-passing programs, designed for high performance on both massively parallel machines and on workstation clusters. The goal of MPI is to establish an easy-to-use, portable, efficient, and flexible standard for writing software in distributed memory environments. Where the thread model breaks down

(i.e. on different workstations within a cluster), MPI can be used to implement Process Network queues.

In this paper, we evaluate multiple parallel implementations of a prototype 20-GFLOP digital interpolation beamformer on a Sun Ultra Enterprise workstation. Using the same beamforming kernel routines, we compare the performance of multiple frameworks: MPI, Computational Process Networks, and an MPI / Process Network hybrid.

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 [5]. 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. For M sensors (transducers), the output of a single beam is ideally calculated by (1), where $x_m(t)$ is the signal received at the m^{th}

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

sensor, α_m is the weighting (shading) coefficient for the m^{th} sensor, and τ_m is the time delay applied to the output of the m^{th} sensor. 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. This is demonstrated in Fig. 1 with a semi-circular array of 80 elements for a beam pointing 20˚ off axis.

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 [5]. Fig. 2 shows a digi-

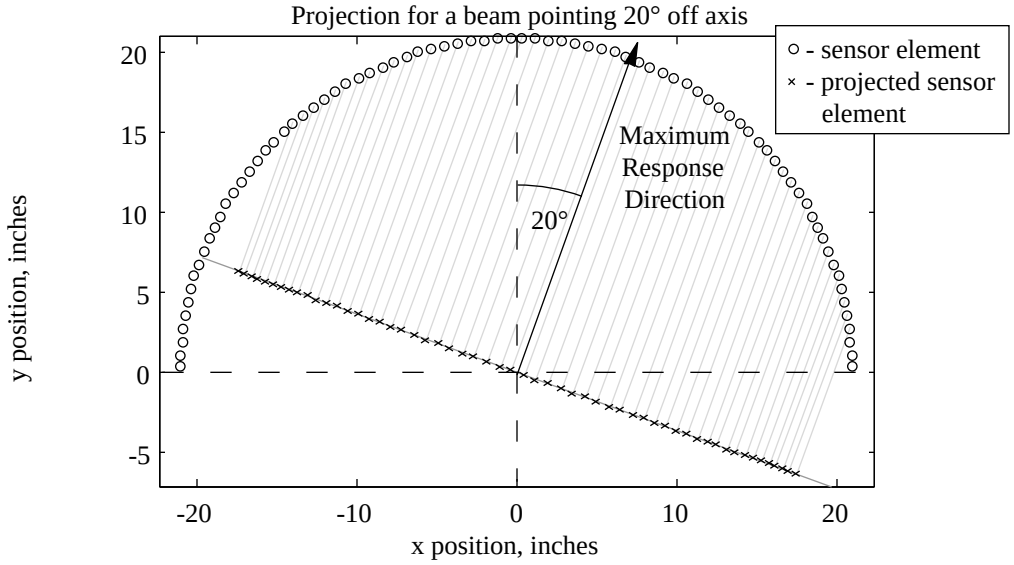

Fig. 1: Projection of sensor elements from a semi-circular array.

tal interpolation beamformer.

The interval needed to satisfy the sampling theorem is Δ. Digital interpolation is performed to the interval δ, where $\delta = \Delta / L$, and L is an integer larger than one. Interpolation consists of padding L-1 zeros after each sample, followed by lowpass filtering with a cutoff frequency of π/L radian/sample. Now the weighted delay and sum operation is performed digitally, and time delays are quantized to integer multiples of δ, i.e., $\tau_m = N_m \delta$. While it is possible to form beams at the higher rate δ, it is commonly desirable to generate output samples at the same rate as the input.

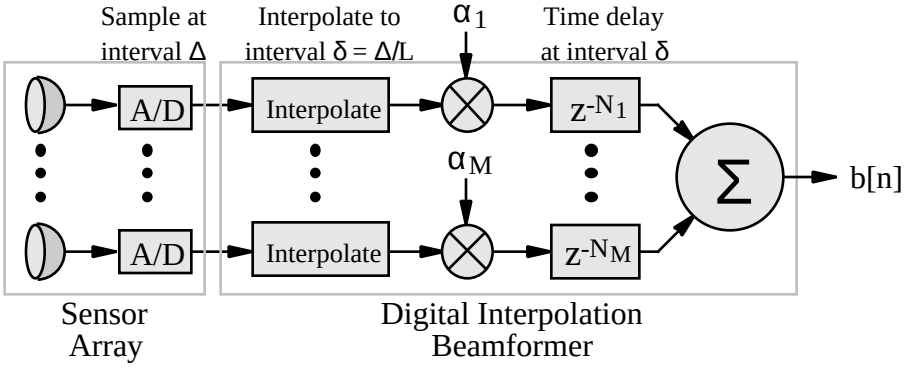

Fig. 2: Digital interpolation beamformer with a digitizing sensor array.

This can be performed by calculating only every L^{th} point – no further filtering is required.

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.

Time-domain beamforming is a linear, time-invariant (LTI) operation, using only multiplication and addition. Because of this, the computations can be broken down and performed in parallel, and these partial results summed to yield the final result. This technique is commonly used in the implementation of real-time beamformers, and has been given the name “partial beamforming”. When utilizing this technique, the beamforming problem is broken down to a granularity where each part can performed in real-time by a computational unit. By providing enough parallel computational hardware, the full beamforming operation can be performed in real time.

3. A Prototype Beamformer

We use partial beamforming to prototype a very large beamforming system which forms 201 beams from an array of 360 sensor elements. Using the one-pass interpolation beamforming method, this beamformer requires about 21 GFLOPS of computation, and will require on the order of fifty 336 MHz UltraSPARC-II processors to run at real time. Although we cannot expect to create a real-time implementation for this system, we can explore how to divide the problem and perform some indicative benchmarks to assess parallelism and scalability.

Fig. 3 shows the one-pass coefficients for a single beam of our beamforming system. The beamforming coefficients are modeled as a sparse FIR filter of length 39600. Clearly, data parallelism on a single SMP workstation cannot achieve real-time performance. Fig. 4 represents an appropriate way to functionally divide the beamforming parallelism. Each arch in Fig. 4 corre-

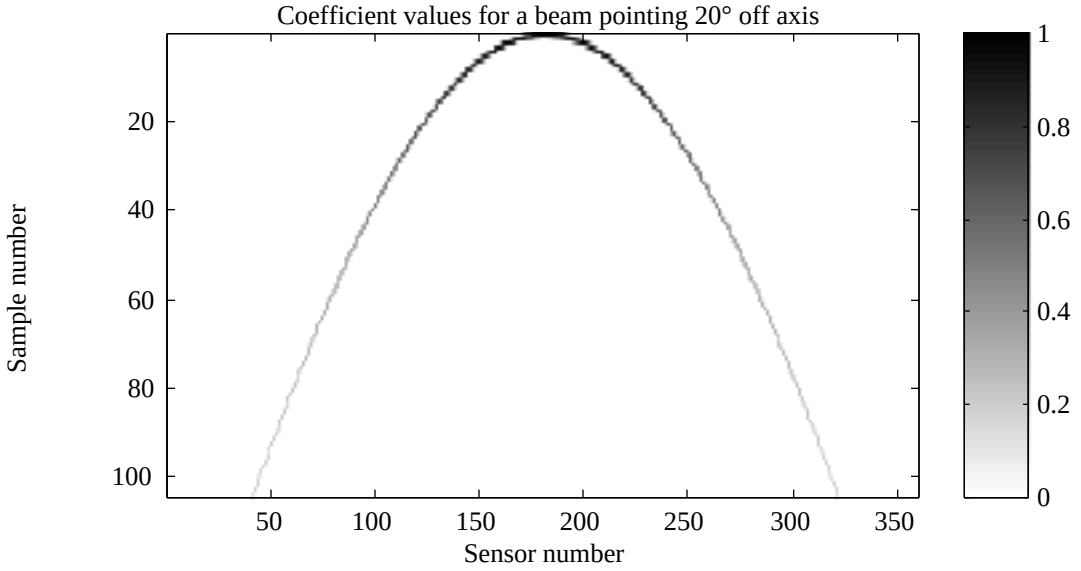

Fig. 3: Beamforming coefficients for a single beam.

sponds to the coefficients for a single beam (as shown in Fig. 3). For clarity, only 7 of the 201 beams are shown. The full beamformer can be divided into 12 similar partial beamformers, corresponding to the 12 columns in Fig. 4. Here, each component computes the contribution of 90 sensor elements to 67 partial beams. By summing the partial beams from each set of sensors, we have computed the full beams.

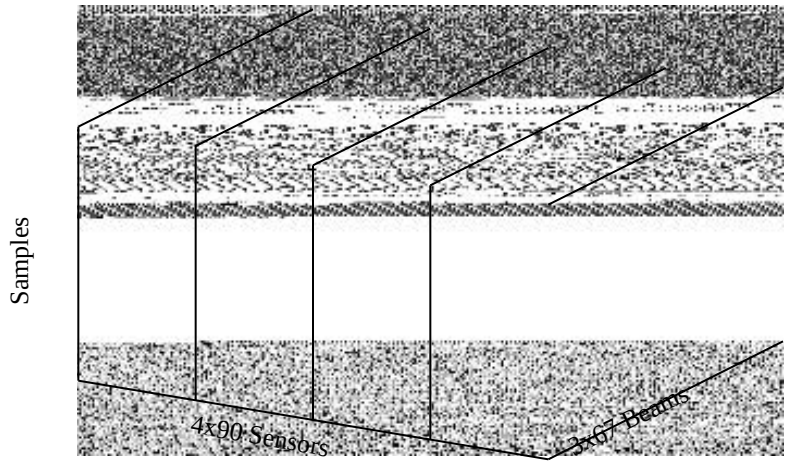

Fig. 4: Dividing the parallelism.

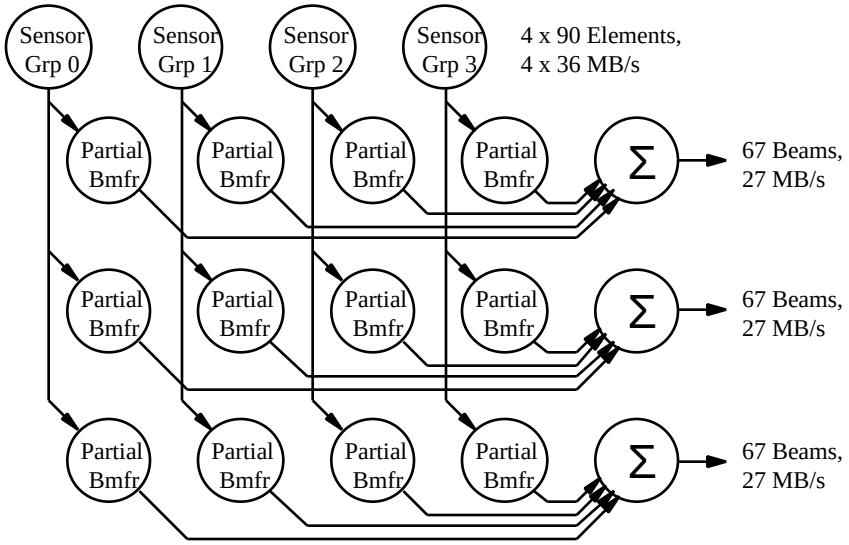

Fig. 5: Prototype beamformer block diagram.

Fig. 5 shows a dataflow block diagram of this beamforming system. The 12 partial beamformer nodes correspond to the 12 columns of Fig. 4. Although each partial beamformer node forms 67 partial beams from 90 sensors, the amount of computation performed at each node varies. The average partial beamformer node computation (using the one-pass method) is 1.75 GFLOPS, and the maximum is 2.4 GFLOPS (where all sensors contribute to all beams).

4. Implementation Frameworks

We implement the prototype digital interpolation beamformer of Fig. 5 using MPI, Computational Process Networks, and an MPI / Process Network hybrid. All implementations use the same high-performance beamforming kernel routines [6], so that we can compare the performance of the frameworks.

4.1. Message Passing Interface

MPI provides an easy-to-use, flexible standard for writing software for distributed memory environments. The code for a partial beamformer node is essentially the following: receive the sensor data, compute the partial beam results, send the results.

```
MPI_Recv( /* the incoming sensor data */ );
bmfr.BeamformBlock(srcSensors, dstBeams, numSamps);
MPI_Send(/* the outgoing partial beam data */);
```

Several MPI variations on this were attempted (including non-blocking and synchronous sends), but none were significantly faster than the “standard” case. Some were an order of magnitude slower. MPI is very powerful in that it can operate on a cluster of workstations. However, results were less than impressive on a single SMP workstation.

4.2. Computational Process Networks

Kahn Process Networks [7] are a formal model which naturally capture 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. Parks [8] developed a dynamic scheduling algorithm which allows execution in bounded memory.

Computational Process Networks extend this model by borrowing and extending the concept of a firing threshold from Computation Graphs [9]. Implementing the Computational Process Network model with POSIX Pthreads gives a low-overhead, high-performance, scalable framework [3]. Here, each processing node corresponds to a single thread on an SMP workstation. The queues in this implementation are designed for high-throughput data. By using pointers to operate directly in queue memory, we can avoid data copying. The code for a partial beamformer node is essentially the following: obtain pointers for incoming and outgoing data, compute the partial beam results in queue memory, release (dequeue) the incoming data, insert (enqueue) the resulting data.

```
float* srcSensors = sensorQ->GetDequeuePtr(numSensorsRequired);
float* dstBeams = beamQ->GetEnqueuePtr(numBeamsOut);
bmfr.BeamformBlock(srcSensors, dstBeams, numSamps);
sensorQ->Dequeue(numSensorsToDiscard);
beamQ->Enqueue(numBeamsOut);
```

Although the thread model (and therefore this implementation) is not capable of using clusters of workstations, it provides substantially improved performance on a single SMP workstation. However, because the queue class of this implementation was designed with virtual inheritance in C++, queues can be extended to use different transport mechanisms.

4.3. Computational Process Networks with MPI

We can extend the Computational Process Network queue class to allow transport over MPI. Therefore, the framework can execute across distributed memory systems. To do this, we implement an MPI queue class which provides the above Computational Process Network interface.

Design of a sending queue is straightforward. We must simply keep a buffer that `GetEnqueuePtr` returns a pointer to, and make `Enqueue` call `MPI_Send`. To overlap computation and communication, we can use two buffers. The design of a receiving queue is more complex, because we wish to support algorithms on overlapping data (such as filters and beamformers). Our solution here is to reuse the underlying queue implementation from Section 4.2. Now data received from MPI is simply inserted into the queue, and `GetDequeuePtr` and `Dequeue` operate similarly to before (while servicing some non-blocking `MPI_Irecv` calls).

A difficulty encountered in this hybrid of Process Networks and MPI was the use of C++ templates in the queue class. The fact that queues use templates makes them flexible, because they can contain any type of data. Unfortunately, this does not work well with the MPI method of typing (`MPI_FLOAT`, etc.) because C++ has no construct to perform the following:

```
switch (typeof(T)) {
  case float: mpiType = MPI_FLOAT; break;
  ...
}
```

The simple solution is to always use `MPI_BYTE`, and multiply all operations by `sizeof(T)`. While this does work correctly, it bypasses all of MPI's powerful typing on heterogeneous hardware.

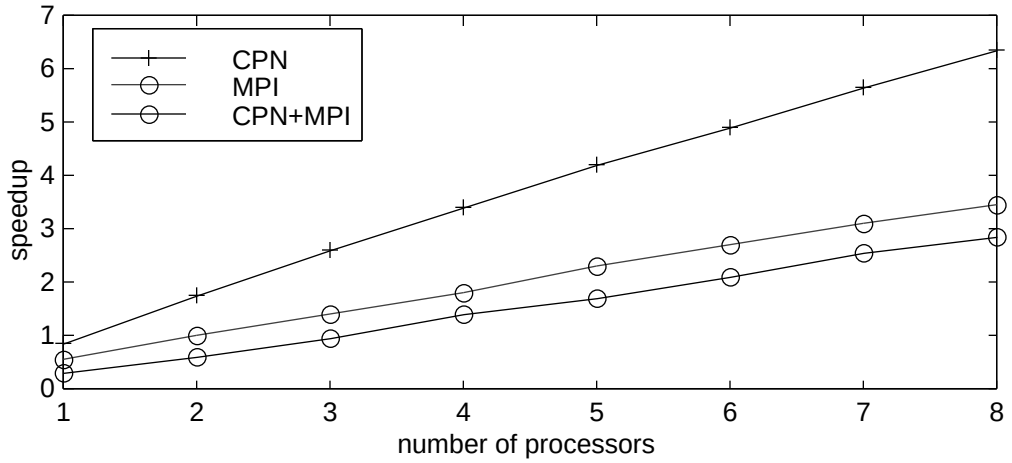

Fig. 6: Beamforming speedup results.

5. Results

For the prototype beamforming system depicted in Fig. 5, we evaluate multiple parallel implementations versus a sequential version. These benchmarks were performed on a Sun Ultra Enterprise 4000 with eight 336-MHz UltraSPARC-II processors and 2 GB of RAM, running the Sun Solaris 2.6 operating system. We used Sun's MPI implementation, a component of Sun HPC Software 2.0. All results are calculated as the average execution time over 50 trials to calculate 64K samples of data. Because this is intended to model a real-time system, only the actual beamforming time is measured – allocating memory and spawning processes (or threads) is not included.

The sequential version of this beamformer runs in just under 30 seconds, delivering 480 MFLOPS using a single processor (with 8 active). This is about 70% of peak operation, and very nearly demonstrates the raw beamforming kernel performance. Fig. 6 displays the beamformer benchmark results, where "MPI" uses the implementation of Section 4.1, "CPN" uses Computational Process Networks from Section 4.2, and "CPN+MPI" uses the hybrid implementation of Section 4.3. This plot was generated by disabling individual processors, and benchmarking the same executables. The CPN implementation substantially outperforms both MPI implementations. On a single processor, the Process Network implementation exhibits a slowdown of 15.7%, compared to MPI's slowdown of 83.9%. On eight processors, the CPN implementation shows a

speedup of nearly 6.5, while the MPI speedup is about 3.5. The CPN implementation operates at 3.1 GFLOPS. It would be possible to speed the CPN further by implementing a shared-memory fork – currently the producer node copies the same data into multiple queues.

The hybrid has even poorer performance than the MPI implementation. This is perhaps because of the extra data copying performed, but is more prominent than expected. Although a major advantage of MPI is its ability to operate across multiple networked workstations, preliminary measurements have shown 10-times slowdowns when run across two workstations networked with 100mb ethernet, making the sequential case faster than the 12-processor networked case! Clearly further study is required.

6. Conclusion and Commentary

We have implemented a prototype 20-GFLOP digital interpolation beamformer with multiple frameworks for parallelism: MPI, Computational Process Networks, and a hybrid Process Network over MPI. While MPI is somewhat portable and easy-to-use, results indicate that performance is less than desirable.

The development of standards such as MPI is important, and I feel that MPI is "a step in the right direction." However, it seems that MPI is not optimized for high-throughput applications such as real-time signal processing or multimedia (such as streaming video). Indeed, the use of MPI is largely absent from the embedded real-time signal processing community. It seems that MPI is more suitable for applications which require less data movement per computation, such as prime factorization or password cracking.

I feel that MPI's choice of a process as the smallest unit of computation will become outdated. The shared memory model is becoming more and more widely used with the growth of SMP machines, making threads a more common mechanism for parallelism. As distributed shared memory systems (already being implemented by several vendors) become commonplace, this will

become even more true. Threads are easier to coordinate than multiple processes, and much more efficient. (Sun's hefty per-CPU license for MPI is also discouraging.)

I also feel that parallel programming will come to necessitate formal software models. Debugging sequential programs is difficult enough; debugging parallel programs, with their deadlocks and nondeterministic behaviors can be *extremely* difficult. MPI makes no attempt to implement any formal models.

References

[1] 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.

[2] The IEEE Portable Applications Standards Committee Web Page: `http://www.pasc.org/`

[3] G. E. Allen and B. L. Evans, "Real-Time Sonar Beamforming on a Unix Workstation Using Process Networks and POSIX Threads", IEEE Transactions on Signal Processing, to appear, March 1999. `http://www.ece.utexas.edu/~allen/Journal/`

[4] The Message Passing Interface Forum Web Page: `http://www.mpi-forum.org/`

[5] 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.

[6] 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*, Pacific Grove, CA, Oct., 1999.

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

[8] 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.

[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.