Scalable Multi-core Sonar Beamforming with Computational Process Networks

John F. Bridgman, III, Gregory E. Allen and Brian L. Evans
Applied Research Laboratories and
Dept. of Electrical and Computer Engineering
The University of Texas at Austin, Austin, Texas

***Abstract*—This paper evaluates the scalability with respect to processor cores of a three-dimensional sonar beamforming kernel implemented on a multi-core workstation. Beamforming is an example of an extremely parallelizable problem. This implementation is instrumented with OpenMP to exploit multi-core computer systems. However, when executed on a 16-core machine, this kernel scales much less than expected. We implement this beamformer system within the scalable framework of Computational Process Networks to achieve additional performance and processor utilization for a larger number of cores. On our benchmark machine, the implementation with Computational Process Networks obtains a throughput speedup of more than two times over OpenMP with the default settings, and 13% improvement in throughput over OpenMP with optimized settings.**

I. Introduction

A beamformer is a spatial filter which focuses an array of sensors. The process of beamforming is a large computational load. Because of the computational load of beamforming, beamformers have traditionally been implemented in custom hardware in order to achieve real-time performance. To save cost, it is desirable to efficiently exploit the parallelism provided by modern commodity computer hardware. Use of the single instruction multiple data (SIMD) instructions in the x86 architecture also gives outstanding performance for digital signal processing algorithms. OpenMP [1] is an application programming interface to support shared memory parallelism provided with many popular compilers. The beamforming kernel used in this paper efficiently utilizes 2-core machines using OpenMP and SIMD. However, when executed on a machine with 16 cores, the kernel does not scale well.

OpenMP allows the addition of simple statements to automatically parallelize sequential code. The basic abstraction of OpenMP is the fork and join model. In this model, processing initially proceeds sequentially. When a parallel construct is encountered, threads are spawned and the work is distributed among the threads. When the parallel work is complete, the threads join together and the program continues sequentially. Typical implementations spawn threads only once and keep the threads around for the entire program lifetime as an optimization. A typical usage of OpenMP is to parallelize a loop which performs independent operations on each element of a large array of data. The fork and join model is very useful and can capture many forms of parallelism; but other forms, like pipelining, are difficult to capture. OpenMP is a useful tool, but its scalability is limited because of its fundamentally sequential underpinnings. Therefore, we turn to another model.

This work was supported by the Independent Research and Development Program at Applied Research Laboratories: The University of Texas at Austin.

Process networks [2] is a formal model of concurrency in which concurrent processes can communicate via one-way first-in first-out (FIFO) queues. A process network can be represented by a directed graph of nodes and edges, where each node represents a computational unit, and each edge represent a queue. A process may not examine a queue and must block if attempting to read from an empty queue. This model provides deterministic behavior, but may require infinite-length queues. Process networks are a subset of general message passing. Unlike general message passing, process networks are provably deterministic.

Computational Process Networks [3] (CPN) is a model and framework for high-throughput signal and image processing systems. CPN uses algorithms presented by [4] and [5] to execute the process network in bounded memory where possible. CPN also provides thresholds and zero copy queues to elide unnecessary copies [6]. This beamforming algorithm moves megabytes of data, so reducing unnecessary copying is important to maintaining performance. CPN can increase the beamforming algorithm's scalability by providing high level parallelism to the larger components. In addition, the CPN framework can be distributed across multiple machines by using a network for remote queue communication. This paper presents a modification of a beamformer implementation which uses CPN to gain additional parallelism and scalability. Effective usage of the memory and cache hierarchy are also critical to the algorithm's performance. The process network model is inherently more concurrent than the fork and join model, and is therefore more scalable. The contribution of this paper is to compare how the process network model can increase the performance of the presented beamforming algorithm on multi-core processors, as compared to OpenMP.

The following is an outline of the rest of this paper. Section II discusses the beamforming algorithm. Section III describes our algorithm implementations. Section IV presents benchmarking results. Section V concludes the paper.

II. Sonar Beamforming

A common beamforming algorithm is a weighted delay-and-sum of the sensors in an array. In order to specify the delays for the algorithm, the geometry of the array must be known. The

Fig. 1. The forward quarter of the sensor array as viewed from above.

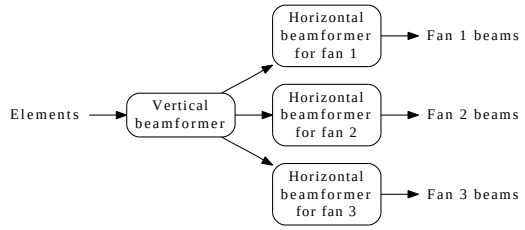

Fig. 2. Vertical and horizontal stages of the beamforming algorithm.

delays required to steer a beam in a particular direction are the signal propagation time from each of the sensors onto a plane that is orthogonal to the steering direction. The geometry used in this paper is a cylindrical array composed of 256 staves. Each stave is a vertical column of 12 elements. Each stave lies on one of 560 points equidistant around a circle. The staves are in groups of eight with one empty point between each in the group. Groups are separated by three or five points as can be seen in Fig. 1. The extra gaps between certain staves allow space for mechanical structure in the array.

The presented beamforming algorithm is composed of two stages: vertical and horizontal. The vertical beamformer forms three sets of stave outputs by steering and summing the 12 vertical elements in each stave. These three sets of stave outputs can be individually steered in the vertical dimension. The operations performed on each set of outputs is identical, but with different coefficients for weights and delays. After the vertical beamforming is complete, the staves are upsampled in space from 256 to 560 for the horizontal beamformer. Effectively, the horizontal beamformer sees 560 virtual staves on its input where 304 of them are always zero. This upsampling is done inside the horizontal beamformer to reduce unnecessary calculations. The horizontal beamformer forms a set of 560 beams which are equally spaced around the circle. The final output is three "fans" of beams.

For the rest of this discussion, the algorithms for a single vertical output are used and it is understood that each of these calculations is repeated. Three horizontal beamformers operate on the three sets of stave outputs from the vertical beamformer, as shown in Fig. 2. Because the staves lie on a circle, circular convolution beamforming [7] can be used to exploit geometric symmetry and significantly reduce the number of required calculations. This produces a set of beams that can be steered both vertically and horizontally.

For this system, sampled element data at the beamformer is assumed to already be frequency shifted to baseband and downsampled. The input to the vertical beamformer is a set of element data streams $x[s][e][n]$, for each stave s and vertical element e. The weighting and beamforming delay is implemented as a set of 4-tap fractional delay finite impulse response (FIR) filters, $h_v[e][n]$. The fractional delay filters are computed according to an algorithm presented in [8]. For the calculation of a vertical output set, each stave utilizes the same series of weights and delays. Because the algorithm performs delays at baseband, an additional complex correction factor $B[e]$ is also required. This correction factor could be combined into the filter, but leaving the correction factor separate makes the filter all real for the vertical beamformer. This reduces the number of operations for each multiplication in the convolution from six operations to two. The output for each stave is

$$y[s][k] = \sum_{e=0}^{E-1} \left[\sum_{l=0}^{3} x[s][e][k-l] h_v[e][l] \right] B[e]. \quad (1)$$

where $y[s][k]$ is the output stream of the vertical beamformer for stave s and time index k. There are 3072 total elements $(256 * 12)$ that must be filtered. For a sample rate of 75kHz, the vertical beamformer takes about 16.6 billion floating point operations per second for all three output sets.

The horizontal beamformer is similar to the vertical beamformer but uses circular convolution. A straightforward equation to produce horizontal beam m from a set of M virtual staves is

$$b[m][k] = \sum_{s=0}^{M-1} \sum_{l=0}^{L-1} y[s][k-l] h[m][s][l] \quad (2)$$

where L is the length of the fractional delay FIR filter, and the weighting and baseband correction is combined into the filter, h. For each horizontal beam to be produced, (2) would have to be computed. In this form, arbitrary beams can be formed from arbitrary staves.

However, we wish to form M horizontal beams that are equally spaced around the circle, and use the same weights, delays, and (geometrically relative) staves to form each of the beams. This gives us symmetry in the filter with regards to m and s. Specifically, for any integer n,

$$h[m][s][l] = h[(m+n) \bmod M][(s+n) \bmod M][l]. \quad (3)$$

We can therefore define a new filter $h_h[n][l]$ to exploit this symmetry. It is a subset of the original $h[m][s][l]$ where $n = (m-s) \bmod M$. Changing (2) to use the symmetric h_h the equation becomes

$$b[m][k] = \sum_{s=0}^{M-1} \sum_{l=0}^{L-1} y[s][k-l] h_h[(m-s) \bmod M][l]. \quad (4)$$

Swapping the summands we get

$$b[m][k] = \sum_{l=0}^{L-1} \sum_{s=0}^{M-1} y[s][k-l] h_h[(m-s) \bmod M][l], \quad (5)$$

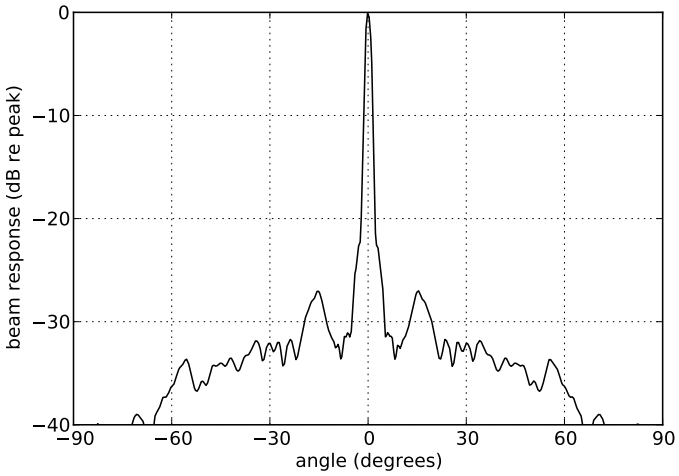

Fig. 3. Simulated beampattern as generated by the presented beamformer.

and note that this is a circular convolution in space around the array, followed by convolution in time. The inner circular convolution is now the same calculation for all beams. This significantly reduces the number of convolutions that must be done when calculating all the beams.

Circular convolution can be efficiently implemented with the Fast Fourier Transform (FFT), allowing all 560 beams to be calculated at once. The horizontal beamformer performs FFTs in both the (circular) spatial dimension and the time dimension. In the time dimension, an FIR filter is being implemented (in the frequency domain) to weight and delay the stave inputs. This reduces the algorithm from $O(LM^2)$ operations per sample to $O(LM \log_2 M)$ operations per sample. Matched filtering (replica correlation) is also performed in the horizontal beamformer simply by convolving the transmitted waveform replica with the beamforming filters. Overlap-and-save [9] is used to prevent time aliasing. The steps of the horizontal beamformer are:

1) 2-D FFT in time and space
2) Multiply by beamforming coefficients
3) Inverse FFT in space
4) Multiply by correlation coefficients
5) Inverse FFT in time

For this system, one horizontal beamforming step (operating on 8192 complex samples) is 908 million floating point operations. For a sample rate of 75kHz and an overlap of 2048, the computational load is 33.2 billion floating point operations per second for all three horizontal beamformers.

A beampattern is the response of a beamformed array versus angle. It is typically measured by rotating a point source around the array and taking the response of a beam at each angle. Because of the symmetry of this circular array and circular convolution beamformer, a single-beam pattern can quickly be approximated by the response of each beam to a single point source. Fig. 3 shows a simulated beampattern for the presented system using a point source at zero degrees. As expected, the beamformer gives a large response in the direction of the steered beam, while rejecting signals from other directions. In Fig. 3 there is more than a 25dB difference between the main lobe and the largest side lobe.

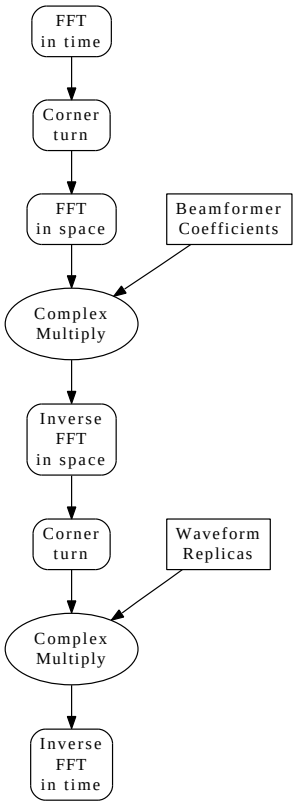

Fig. 4. Steps of the horizontal beamformer implementation.

III. Implementation

The algorithm implementations were written in C/C++ using the x86 streaming SIMD extension (SSE) intrinsics. The outer loops in the vertical and horizontal kernels were all given the necessary pragmas for OpenMP. The vertical beamformer produces all three outputs at once with a single pass over the input data. The horizontal beamformer implementation uses the Fastest Fourier Transform in the West [10] (FFTW) library to perform the FFTs. The FFTs that are not inside OpenMP loops use the thread support built into FFTW. The number of threads that the FFTW library uses is the number of online processors in the machine. The implementation uses corner turns to gain additional performance by keeping memory access patterns contiguous. The data sets used in these algorithms are large enough that the data does not fit into most processors' caches. The main memory used in most systems is optimized for sequential access patterns. The cost of doing the corner turn was measured to be less than the performance penalty from using a non-sequential access pattern for this algorithm.

The horizontal beamformer has eight steps in the implementation as seen in Fig. 4. First, an FFT in time is executed. Then,

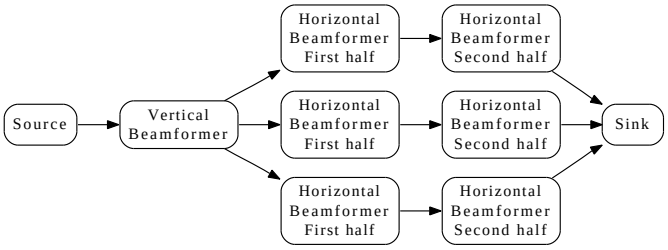

Fig. 5. The topology of the CPN beamformer implementation.

an optimized corner turn is done so that memory accesses will be sequential for the next three steps. An FFT in space is performed. Next, the coefficients are multiplied. An inverse FFT is done in space. Then, the data is corner turned again. The waveform replica coefficients are multiplied in. Finally, the inverse FFT in time is performed.

We have two implementations of the full beamformer system. The first implementation uses only OpenMP. The OpenMP implementation executes the vertical beamformer, then each horizontal beamformer sequentially for some number of samples. We experimented with how to place the OpenMP statements, and the best was chosen for our experiment. The second implementation uses CPN. The vertical and horizontal beamformers were placed in separate process network nodes. The horizontal beamformer was also divided into two nodes. The first half computes all operations up to multiplying the coefficients and the second half computes the remainder. A source node and a sink node were created. All the nodes were connected together with the CPN framework in the topology shown in Fig. 5.

Compiling and benchmarking was performed on RedHat Enterprise Linux 5 using the provided GCC 4.1.2 with GCC's OpenMP implementation. The default behavior of the OpenMP implementation was to use busy waiting for synchronization, which the OpenMP standard calls "active waiting policy". In our tests, we timed both the default waiting policy and the "passive waiting policy" which uses calls into the operating system synchronization routines. We explored these settings because busy waiting can adversely affect performance on many modern processors with features like automatic overclocking and hardware threads. For the "active waiting policy", the default spin count is to spin forever. Experimentation with setting the spin count gave the same results as setting the wait behavior to passive. This is because all of the parallel sections in this algorithm are longer than the operating system time slice. The default behavior always performed poorly on average in all of our tests.

IV. Results

For each implementation, the average throughput in samples per second was measured. Fig. 6 shows the measured results for several implementations versus the number of processors. Both implementations were run on a machine with two 4-core Nehalem processors [11] with Hyper-Threading. Hyper-Threading gives each core two hardware threads. This means

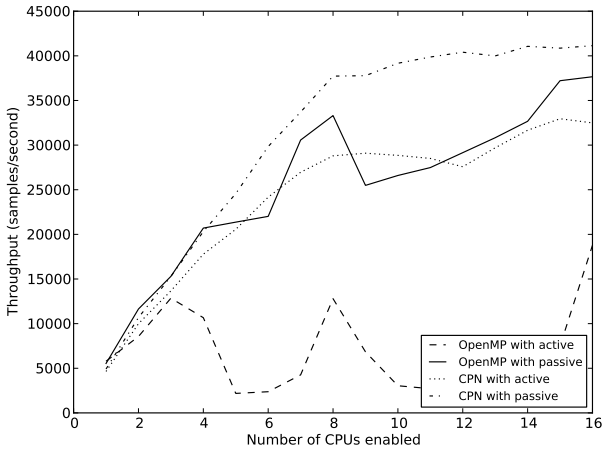

Fig. 6. Graph of throughput versus number of processors enabled for the CPN and OpenMP implementations.

that from the software's perspective there are 16-cores. When running the tests, the processors were turned on and off so that, first, only one hardware thread was active per hardware core. Then, only when all hardware cores had a hardware thread assigned, additional hardware threads were assigned to each hardware core. This is immediately apparent in Fig. 6, because there are only eight hardware cores and the graph starts to plateau at eight cores. Changing the order the cores are turned on, moves the plateaus in the graph. The small increase in the plateaus represent the increase in performance gained by adding a second hardware thread to a core. The CPN version continues to increase in throughput until all hardware cores have one hardware thread and then continues to increase much more slowly as all hardware threads are assigned. The OpenMP version increases much less than the CPN version. At eight cores, the CPN version is over twice as fast as the OpenMP with default settings. The CPN version with passive is 13% faster than the OpenMP with passive. When the second hardware thread in each core is enabled, the OpenMP version's performance starts to suffer.

Also, note that the versions that used the default active waiting policy have much lower performance with more than one hardware thread per core. This is because resources are shared between hardware threads and if one thread spins it uses functional units in the core which could be used by other threads to advance.

To take the measurement in Fig. 6, we ran the beamformer on 8192 samples 100 times, computing the throughput for each time and then computing the average throughput. For the measurements for CPN, the data generating node kept track of the time it took to enqueue 8192 data points 100 times, then computed the throughput based on this time. Each test was run with the default "active wait policy" for the OpenMP implementation and also with the policy set to passive. As can be seen from Fig. 6, the default OpenMP settings perform very poorly.

The theoretical maximum speed on the benchmark computer is 118.7kHz, when we consider only the floating point operations. This is approximately three times the maximum speed seen in the tests. Most of this time is spend not doing calculations, but moving data. The working set for one horizontal beamformer is approximately 32 MB. The cache size of the processor used is only 8 MB. This means the best that can be done without completely redesigning the algorithm is to have memory access patterns that ensure the most cache hits. A vertical kernel alone performed near the theoretical limit on the benchmark machine with an average throughput of 320 ksamples/s. A horizontal kernel alone performed near 24% of the theoretical maximum on the benchmark machine with an average throughput of 12.4 ksamples/s. The time spent in the horizontal kernel is dominated by the FFTW library.

V. Conclusion

Beamforming is a high computational load which can be described as embarrassingly parallel. We show how a simple implementation of a beamformer can be made much more scalable with CPN. The original implementation only used OpenMP and was only moderately scalable. We used process networks as an additional model of parallelism. The addition of a different model of parallelism increased the scalability of the implementation. Most of the additional parallelism captured with CPN is pipelining parallelism, which is difficult to capture with OpenMP. Pipelining parallelism can be easily added with the CPN framework. Also, CPN can easily distribute a process network across a cluster of machines. The CPN framework increased performance of the beamformer by 13% on the benchmark machine. We have shown how the process network model increased the performance and scalability of the beamforming algorithm on multi-core processors.

References

[1] (2008) OpenMP application program interface. [Online]. Available: http://www.openmp.org/mp-documents/spec30.pdf

[2] G. Kahn, "The semantics of a simple language for parallel programming," in *Proc. IFIP Congress on Information Processing*, J. L. Rosenfeld, Ed. New York, NY: North-Holland, 1974, pp. 471–475.

[3] (2010, Feb.) Computational process networks. [Online]. Available: http://webspace.utexas.edu/gallen/CPN/

[4] T. M. Parks, *Bounded scheduling of process networks*. Berkeley, CA, USA: University of California at Berkeley, 1995.

[5] G. E. Allen, P. E. Zucknick, and B. L. Evans, "A distributed deadlock detection and resolution algorithm for process networks," in *Proc. IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, vol. 2, 2007, pp. 33–36.

[6] G. Allen and B. Evans, "Real-time sonar beamforming on workstations using process networks and posix threads," *IEEE Transactions on Signal Processing*, vol. 48, no. 3, pp. 921–926, Mar. 2000.

[7] D. R. Farrier, T. S. Durrani, and J. M. Nightingale, "Fast beamforming techniques for circular arrays," *The Journal of the Acoustical Society of America*, vol. 58, no. 4, pp. 920–922, 1975. [Online]. Available: http://link.aip.org/link/?JAS/58/920/1

[8] S.-H. Yu and J.-S. Hu, "Optimal synthesis of a fractional delay FIR filter in a reproducing kernel Hilbert space," *IEEE Signal Processing Letters*, vol. 8, no. 6, pp. 160 –162, Jun. 2001.

[9] A. V. Oppenheim, R. W. Schafer, and J. R. Buck, *Discrete-time signal processing (2nd ed.)*. Prentice-Hall, Inc., 1999.

[10] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proc. of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[11] (2010) Nehalem (microarchitecture). [Online]. Available: http://en.wikipedia.org/wiki/Nehalem_(microarchitecture)