Appears in International Parallel Processing Symposium 1999

Multi-Threaded Design and Implementation of Parallel Pipelined STAP on Parallel Computers with SMP Nodes

Wei-keng Liao†, Alok Choudhary‡, Donald Weiner†, and Pramod Varshney†

† EECS Department
Syracuse University
Syracuse, NY 13244

‡ ECE Department
Northwestern University
Evanston, IL 60208

Abstract

This paper presents performance results for the multi-threaded design and implementation of a parallel pipelined Space-Time Adaptive Processing (STAP) algorithm on parallel computers with Symmetrical Multiple Processor (SMP) nodes. In particular, the paper describes our approach to parallelization and multi-threaded implementation on an Intel Paragon MP system. Our goal is to determine how much more performance can be enhanced using small SMPs on each node of a large parallel computer for such an application. The paper also discusses the process of developing software for such an application on parallel computers when latency and throughput are both considered together and presents their tradeoffs. The results show that not only scalable performance was achieved for individual component tasks of STAP but linear speedups were obtained for the integrated task performance, both for latency as well as throughput.

1 Introduction

Space-time adaptive processing (STAP) is a well known technique in the area of airborne surveillance radars, which is used to detect weak target returns embedded in strong ground clutter, interference, and receiver noise. Most STAP applications consume great amounts of computational resources and are also required to operate in real time. High performance computers are becoming mainstream due to the progress made in hardware as well as software support in the last few years. They can satisfy the STAP computational requirements of real-time applications while increasing the flexibility, affordability, and scalability of radar signal processing systems. However, efficient parallelization of a STAP algorithm which has embedded in it different algorithms, is challenging and requires several optimizations.

In our previous work [3], we described the parallel pipelined implementation of a PRI-staggered post-Doppler STAP algorithm. In this paper, we focus on the multi-threaded design and implementation on the parallel computers with SMP nodes. This STAP algorithm consists of five steps: 1)Doppler filter processing, 2)weight computation, 3)beamforming, 4)pulse compression, and 5)CFAR processing. For our implementation of this real application we designed a model of the parallel pipeline system where each pipeline is a collection of tasks and each task itself is parallelized. This parallel pipeline model was applied to the STAP algorithm with each step as a task in a pipeline. This permits us to significantly improve latency as well as throughput. Performance results presented in this paper were obtained on the Intel Paragon at the Air Force Research Laboratory (AFRL), Rome, New York.

The Intel Paragon at the AFRL is an MP system which has three processors on each compute node board. In this paper, we focus on the design of the parallel pipeline system and its implementation using multi-threading on this system. We demonstrate the performance and scalability on different numbers of compute nodes for both threaded and non-threaded implementations. The improvement of threaded implementation over non-threaded implementation is provided.

The rest of the paper is organized as follows: in Section 2, we present the parallel pipeline system model and discuss some parallelization issues. Section 3 describes the multi-threaded programming environment on the Intel Paragon MP system. Section 4 presents the implementation. Performance results and conclusions are given in Section 5 and Section 6 respectively.

2 Model of the parallel pipeline system

The system model for the type of STAP applications considered in this work is shown in Figure 1. This model is suitable for the computational characteristics found in these applications. A pipeline is a collection of tasks which are

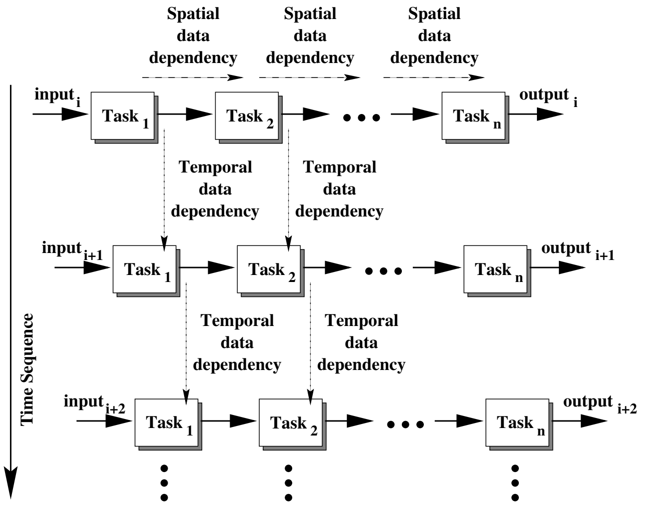

Figure 1. Model of the parallel pipeline system. The set of pipelines indicates that the same pipeline is repeated on subsequent input data sets. Each task for all input instances is executed on the same number of compute nodes.

executed sequentially. The input to the first task is obtained normally from sensors or other input devices and the inputs to the rest of the tasks in the pipeline are the outputs of their previous tasks. The set of pipelines shown in the figure indicates that the same pipeline is repeated on subsequent input data sets. Each block in a pipeline represents one parallel task, which itself is parallelized on multiple (different number of) compute nodes.

From a single task point of view, the execution flow consists of three phases: receive, compute, and send phases. In the receive and send phases, communication involves data transfer between two different groups of compute nodes. In the compute phase, work load is evenly partitioned among all compute nodes assigned in each task to achieve the maximum efficiency. For the parallel systems with SMP nodes, multi-threading technique can be employed to further improve the computation performance.

2.1 Data dependency

In such a parallel pipeline system, there exist both spatial and temporal parallelism that result in two types of data dependencies, namely, spatial data dependency and temporal data dependency [2, 5]. Spatial data dependency can be classified into inter-task data dependency and intra-task data dependency. Intra-task data dependencies arise when a set of subtasks needs to exchange intermediate results during the execution of a parallel task in a pipeline. Inter-task data dependency is due to the transfer and reorganization of data passed onto the next parallel task in the pipeline. Temporal data dependency occurs when some form of output generated by the tasks executed on the previous data set are needed by tasks executing the current data set. We will later see that STAP has both types of data dependencies.

2.2 Compute node assignment

Optimal use of resources is particularly important in high-performance embedded applications due to limited resources and other constraints such as desired latency or throughput [4]. When several parallel tasks need to be executed in a pipelined fashion, tradeoffs exist between the assignment of processors for the maximization of overall throughput as opposed to the minimization of a single data set's response time (or latency.) The throughput requirement says that when allocating processors to tasks, it should be guaranteed that all the input data sets will be handled in a timely manner. That is, the processing rate should not fall behind the input data rate. The response time criteria, on the other hand, require minimizing the latency of computation on a particular set of data input.

3 Multi-threads on Paragon

We implemented our parallel pipeline model of the STAP algorithm on the Intel Paragon XP/S parallel computer located at AFRL. The compute partition of this machine consists of 232 MP nodes, each has three i860 processors on its compute node board. By running UNIX OSF/1 operating system, the three processors are configured with two processors as general application processors and one processor as message coprocessor which is dedicated to message passing. Multi-threaded programming environment is supported on a Paragon system and the threads are implemented as *POSIX threads* [6].

4 Design and implementation

The STAP algorithm we implemented is a PRI-staggered post-Doppler STAP algorithm [1, 7]. The design of the parallel pipelined STAP algorithm is shown in Figure 2. The parallel pipeline system consists of seven tasks. Both the weight computation and the beamforming tasks are divided into two parts, namely, "easy" and "hard" Doppler bins. The hard Doppler bins are those in which significant ground clutter is expected and the remaining bins are easy Doppler bins. The main difference between the two is the amount of data used and the amount of computation required. The input data set for the pipeline is obtained from a phased array radar and is formed in terms of a coherent processing

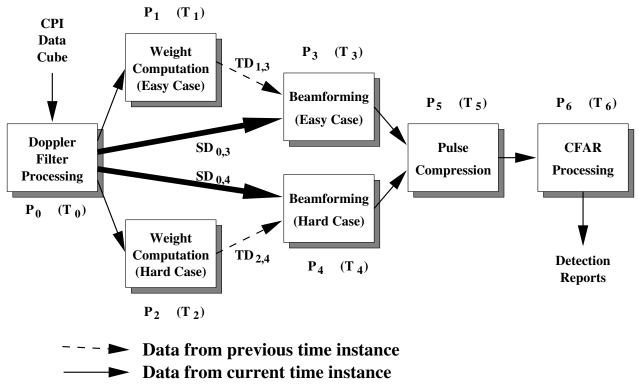

Figure 2. Implementation of parallel pipelined STAP. Arrows connecting task blocks represent data transfer between tasks.

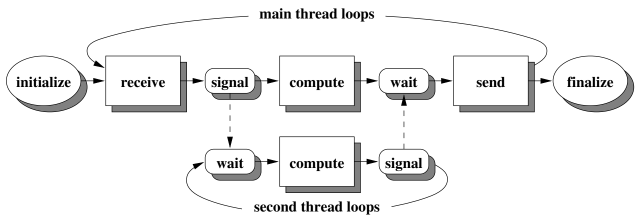

Figure 3. Implementation of two threads in the compute phase. The main thread signals the second thread to perform its computation. After completion of its computation, the second thread signals back to the main thread.

interval (CPI). Each CPI data set is a 3-dimensional complex data cube. The output of the pipeline is a report on the detection of possible targets. Each task i, $0 \le i < 7$, is parallelized by evenly partitioning its work load among P_i compute nodes. The execution time associated with task i is T_i. For the computation of the weight vectors for the current CPI data cube, data cubes from previous CPIs are used as input data. This introduces temporal data dependency. Temporal data dependencies are represented by arrows with dashed lines in Figure 2 where $TD_{i,j}$ represents temporal data dependency of task j on data from task i. In a similar manner, spatial data dependencies $SD_{i,j}$ can be defined and are indicated by arrows with solid lines.

Throughput and latency are two important measures for performance evaluation on a pipeline system.

$$throughput = \frac{1}{\max\limits_{0 \le i \le 6} T_i}. \tag{1}$$

$$latency = T_0 + \max_{i=3,4} T_i + T_5 + T_6. \tag{2}$$

The temporal data dependency does not affect the latency because weight computation tasks use data from the previous time instance rather than the current CPI. The filtered CPI data cube sent to the beamforming task does not wait for the completion of its weight computation. This explains why equation (2) does not contain T_1 and T_2. A detailed description of the STAP algorithm we used can be found in [1, 7].

4.1 Threads in compute phases

In the Intel Paragon MP system, two out of the three processors in one compute node are configured as general processors to run application code while the third as a message coprocessor which is dedicated to message passing. With this configuration, only compute phase for each task in our parallel pipeline system is implemented with threads. The reason for not implementing threads in communication phases is that the Paragon message-passing library is not thread-safe. Since there are only two application processors in each compute node, each compute phase in every task will have two threads implemented. Figure 3 gives the execution flows of two threads in the compute phase.

5 Performance results

The implementation of the STAP application based on our parallel pipeline system model was done on the Intel Paragon at AFRL. Each CPI complex data cube is a $512 \times 16 \times 128$ three-dimensional array. A total of 27 CPIs were generated as inputs to the parallel pipeline system.

5.1 Compute time

For each task in the STAP algorithm, parallelization was done by evenly dividing computational load across compute nodes assigned to the task. Figure 4 gives the performance results of compute phases for different tasks. For each task, we obtained linear speedups on both implementations using two threads as well as using single thread.

Assuming that the execution time of a non-threaded implementation of a task is t_1 and the execution time of its threaded implementation is t_2, we define the threading speedup for threaded over non-threaded implementation as $s = \frac{t_1}{t_2}$. Since two processors are employed in the threaded implementation, we have $\frac{t_1}{2} \le t_2 \le t_1$ and, therefore, $1 \le s \le 2$. The threading speedups for compute phases of all tasks are also given in Figure 4. By running on two processors at the same time, the two-threaded STAP code ideally can have a threading speedup of 2. However, in most cases, the actual threading speedups do not approach this ideal value. This may be caused by the limitation of implementation of operating system, OSF/1, and the implemen-

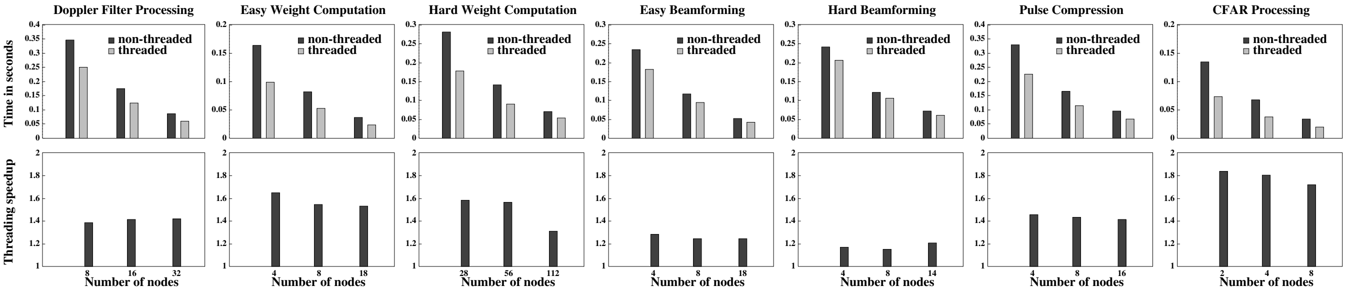

Figure 4. Performance of compute phases as a function of number of compute nodes.

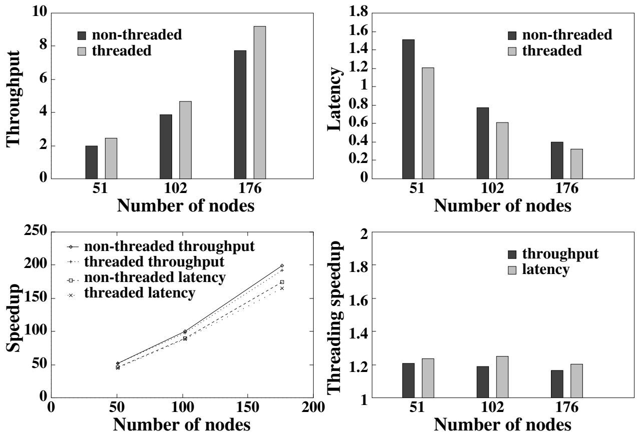

Figure 5. Integrated performance results for threaded and non-threaded implementations.

tation of linked thread-safe libraries. On an Intel Paragon MP system, scheduling of threads is handled by the operating system kernel. Users cannot have control over or get information about which processor runs which thread.

5.2 Integrated system performance evaluation

Integrated system performance evaluation refers to the evaluation of performance when all the tasks in the pipeline are considered together. Throughput (number of CPIs per second) and latency (seconds per CPI) are the two most important measures for performance evaluation on the parallel pipeline system. Figure 5 shows the speedups and threading speedups achieved by the threaded implementation for both latency and throughput for three cases of different compute node assignments with 51, 102 and 176 nodes. From these experiments, it is clear that for latency and throughput measures we obtain linear speedups for both threaded and non-threaded implementations. Given that this scale up is up to 176 compute nodes (we were limited to this number of nodes due to the size of the machine), we believe these are very good results.

5.3 Tradeoff between throughput and latency

Using an example, we illustrate how further performance improvements may (or may not) be achieved if a few additional compute nodes are available. We now take the case with 102 nodes from Figure 5 as an example and add some nodes to the pipeline to analyze its effect on the throughput and latency. Compute nodes were added to each task in increments of two nodes at a time. The resulting throughput and latency are plotted in Figure 6.

When nodes were added to the Doppler filter processing task, the throughput increased and latency reduced. From Equations (1) and (2), this improvement was obtained because the execution time, T_0, is reduced. However, when the number of nodes added is more than 8, both throughput and latency degrade. This is because the Doppler filter processing task finishes its computation on the new CPI so fast that the actual send operations for the previous CPI have not been carried out yet. The waiting time increases Doppler filter processing task's execution time, T_0, and therefore degrades the throughput and latency.

When compute nodes are added to easy and hard weight computation tasks, the resulting throughput and latency have no significant changes. This is because the latency does not contain the execution time of weight computations, as indicated in Equation (2). However, when extra compute nodes are added to either the beamforming or the pulse compression task, we observe that the latency is reduced. This is because the execution times T_3, T_4, and T_5 reduce in Equation (2). The throughput, on the other hand, is not improved because the Doppler filter processing task is the task with the maximum execution time among all tasks.

Figure 6 presents the tradeoffs between increasing the throughput and reducing the latency, when assigning nodes to the tasks in the pipeline. We observed that only the addition of nodes to the Doppler filter processing task can increase the throughput. Similarly, only beamforming and pulse compression tasks are candidates for the addition of more compute nodes to reduce the latency.

Compute node assignment can also be made in such a way that both throughput and latency are improved simul-

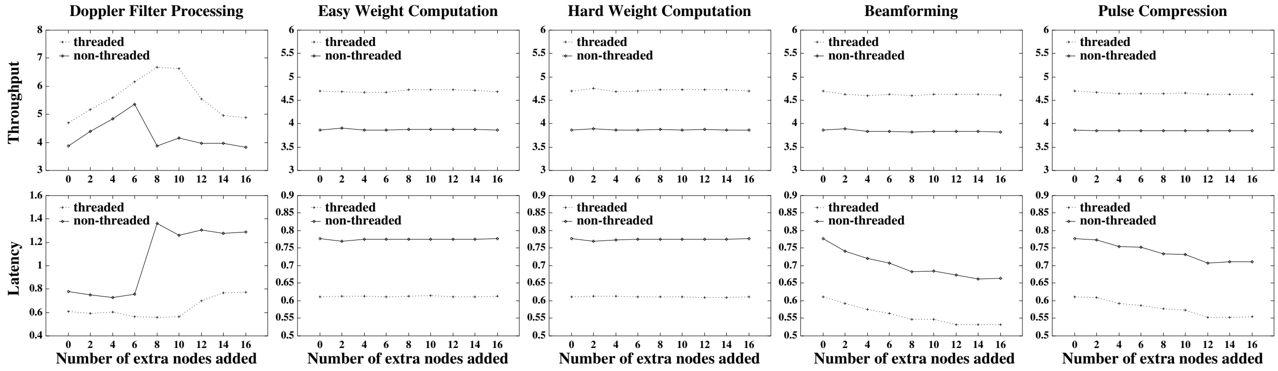

Figure 6. Throughput and latency results by adding 2 compute nodes at a time to each task.

Table 1. Performance results when 4 nodes to the Doppler processing task and 4 nodes to the pulse compression task are added to the implementation with 102 nodes.

	non-threaded		threaded	
# nodes	102	110	102	110
throughput	3.8677	4.8368	4.6916	5.6137
latency	0.7767	0.6650	0.6108	0.5458

throughput: CPIs/sec   latency: sec/CPI

taneously. We now add 4 nodes to the Doppler filter processing task and 4 nodes to the pulse compression task. By increasing the number of compute nodes by 7.8%, the improvement in throughput is 25.1% and in latency it is 14.4% for the non-threaded implementation. Meanwhile, the threaded implementation shows 19.7% improvement in throughput and 10.6% improvement in latency. From these experimented results, we can draw the following conclusions. Extra compute nodes can be assigned to the task that has the maximum execution time among all tasks. In this way, the execution time of this task is reduced and according to Equation (1), the throughput is increased. Extra compute nodes can be added to those tasks which benefit the most, that is, the tasks with greatest reduced execution time when more nodes are assigned. The sum of these tasks can be reduced the most and therefore it minimizes the latency.

6 Conclusions

In this paper we presented performance results for a PRI-staggered post-Doppler STAP algorithm implementation on the Intel Paragon machine at Air Force Research Laboratory, Rome, New York. This Paragon machine has three processors on each compute node board. By taking advantage of the SMP architecture, a multi-threaded implementation is was designed and compared to the non-threaded implementation. Performance results indicate that our approach of parallel pipelined implementation scales well both in terms of throughput and latency whether the multi-threaded technique is used or not. Our design and implementation not only shows tradeoffs in parallelization, compute node assignment, and various overheads in inter-task communication etc., but it also shows that accurate performance measurement of these systems is very important.

7 Acknowledgments

This work was supported by Air Force Materials Command under contract F30602-97-C-0026. We are grateful to Russell Brown, Mark Linderman, Richard Linderman, and Zen Pryk for their help, support, and encouragement during the course of this work.

References

[1] R. Brown and R. Linderman. "Algorithm Development for an Airborne Real-Time STAP Demonstration,". *IEEE National Radar Conference*, 1997.

[2] A. Choudhary. *Parallel Architectures and Parallel Algorithms for Integrated Vision Systems*. Kluwer Academic Publisher, Boston, MA, 1990.

[3] A. Choudhary, W. Liao, D. Weiner, P. Varshney, R. Linderman, and M. Linderman. "Design, Implementation and Evaluation of Parallel Pipelined STAP on Parallel Computers,". *International Parallel Processing Symposium*, 1998.

[4] A. Choudhary, B. Narahari, D. Nicol, and R. Simha. "Optimal Processor Assignment for Pipeline Computations,". *IEEE Trans. on Parallel and Distributed Systems*, Apr. 1994.

[5] A. Choudhary and R. Ponnusamy. "Parallel Implementation and Evaluation of a Motion Estimation System Algorithm using Several Data Decomposition Strategies,". *Journal of Parallel and Distributed Computing*, Jan. 1992.

[6] Intel Corporation. *Paragon System User's Guide*, Apr. 1996.

[7] M. Linderman and R. Linderman. "Real-Time STAP Demonstration on an Embedded High Performance Computer,". *IEEE National Radar Conference*, 1997.