

High Throughput-Rate Parallel Algorithms for Space Time Adaptive Processing *

Myungho Lee and Viktor K. Prasanna

Department of EE-Systems
University of Southern California
Los Angeles, CA 90089-2562
{mlee + prasanna}@halcyon.usc.edu
<http://www.usc.edu/dept/ceng/prasanna/home.html>

Abstract

Space Time Adaptive Processing (STAP) techniques that are being developed for the next generation radar systems require very high computing power. In order to meet the real-time requirements in STAP applications, High Performance Computing (HPC) systems are used. In this paper, we present a methodology to design high throughput-rate parallel algorithms for STAP on HPC platforms. Data remapping is used between successive computation steps of STAP. Task level parallelism is used for these steps. In order to achieve high throughput performance, we regard the HPC platform as executing in a pipelined fashion. For each pipeline stage, we schedule the computation and the communication, and assign a number of processors based on the computation requirements and associated communication time. A simple and accurate communication model is used to characterize the communication time. A technique called *replication* of pipeline stages is employed to effectively reduce the communication cost and the period of each pipeline stage. As an example, our methodology is applied to a general STAP technique called *Beam space post-Doppler* STAP [12]. Experimental results conducted on the IBM SP2 show that our methodology leads to high throughput performance. Our code is written using C and MPI, and is portable across HPC platforms.

1 Introduction

Space Time Adaptive Processing (STAP) [12] techniques are being developed for the next generation radar systems. They demand very high computing power. *High Performance Computing* (HPC) platforms composed of *Commercial-Off-The-Shelf* (COTS) components interconnected by a high speed network are becoming increasingly popular for embedded signal processing applications such as STAP. In order to achieve real-time performance for STAP on HPC platforms, scalable parallel algorithms are essential.

Two performance measures are used to evaluate parallel algorithms and platforms for STAP: *latency* and *throughput*. Traditionally, latency has been emphasized. Latency is defined as the length of time, $T_{latency}$, to process a data stream corresponding to an instance of the problem by the system. Throughput is defined as the inverse of a periodic time interval T_{period} . T_{period} is the average time interval between two successive outputs of the system when one input data is received

* This research was supported in part by DARPA Embeddable Systems Program under contract DABT63-95-C-0092.

every T_{input} seconds. The goal of throughput-oriented paradigm is to minimize T_{period} , whereas that of the latency-oriented paradigm is to minimize $T_{latency}$. In STAP, a stream of input data is received continuously: each of N -antenna elements receives one input data every T_{input} seconds. Under steady state, one output must be produced every T_{period} to keep pace with data input rate. Therefore, throughput is an important performance measure for STAP. However, traditional HPC applications have not paid much attention to this performance measure.

Parallel algorithms for STAP have been designed previously to optimize latency. In STAP, a sequence of computations is performed on an input data cube. This data cube is accumulated by a number of antenna elements in one time interval called *Coherent Processing Interval* (CPI). The dimension along which each computation step accesses the data cube is different in each step. In [9], a parallel algorithm for *Higher Order Post-Doppler* (HOPD) STAP was developed by assuming a fixed data distribution throughout the complete computation. Thus, fine-grain parallelism is exploited throughout the weight computation step of HOPD STAP. In [7], task level parallelism was exploited for each computation step. The data is remapped between successive steps. Analysis in [7] shows that the total communication time of [9] increases as the number of processors increases. This is because exploiting fine grain parallelism in the weight computation step generates many short messages and incurs significant communication overhead. However, the communication cost of [7] decreases linearly as the number of processors increases. This leads to scalable performance for HOPD STAP. A benchmarking study of IBM SP2 for real-time signal processing applications is reported in [6]. They observed that the time delay of a single communication operation in SP2 is equivalent to thousands of floating point operations. Based on this, they claimed that only coarse grain parallelism must be exploited. However, these approaches do not address the issue of optimizing throughput performance.

In this paper, we present a methodology to design parallel algorithms for achieving high throughput performance on HPC platforms. We exploit task level parallelism for each computation step. Data remapping is used between successive steps. To obtain high throughput performance, we regard the HPC platform as executing in a pipelined fashion. A group of processors is assigned to each pipeline stage. For this, we estimate both the computational requirements of each computation step and the communication time between adjacent steps. A simple and accurate communication model [11] is used to characterize the communication time. Our mapping technique allows the same processor group, which is used for an earlier pipeline stage, to be reused for a later stage. Choosing the type of processors for each stage is also allowed, when there are alternatives. Furthermore, our technique employs replication [5] of pipeline stages. A computationally demanding stage requires a large number of processors to be assigned to it. Instead of assigning all the processors to one instance of the stage, we partition the processors into groups and assign each group to an instance of the stage. Therefore, the stage is replicated. Without replication, the communication time usually increases as more processors are assigned to the stage. Therefore, the communication time and the computation time of each instance is amortized over a number of instances. This can reduce T_{period} compared with no replication.

The rest of the paper is organized as follows. In Section 2, *Element space* and *Beam space* STAPs are introduced. Section 3 describes our model of HPC platforms that includes the cost of communication. Section 4 shows our parallel algorithm design methodology. This is followed by an example algorithm and experimental results for *Beam space post-Doppler* (BSPD) STAP.

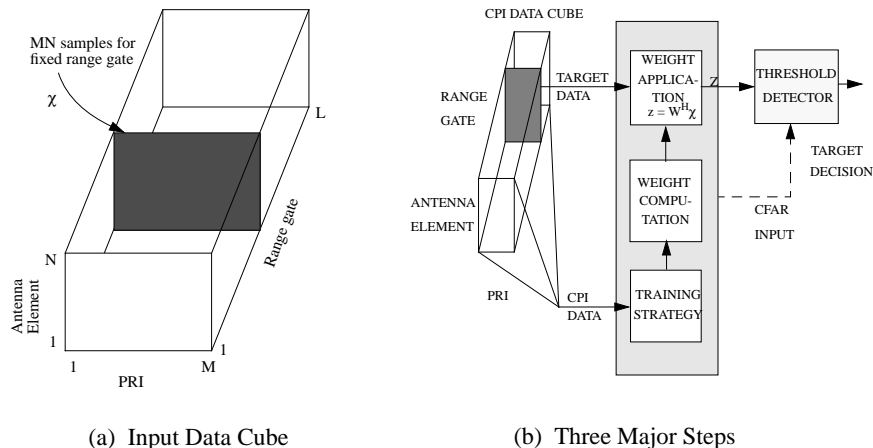


Figure 1: General view of STAP processing

2 Space Time Adaptive Processing Algorithms

STAP consists of three major steps. First, a set of rules called training strategy is used to estimate the interference. The second step is weight computation. Based on the training data, the adaptive weight vector is computed. Generally, weight computation requires the solution of a linear system of equations. This step is therefore a very computation intensive portion. Weight computations are performed with each training data. Finally, the computed weight vector is applied to obtain the scalar output or test statistic. Weight application involves an inner product operation. The scalar output is then compared to a threshold to determine if a target is present at a specified angle and Doppler. The output of the processor is a separate scalar for each range, angle, and velocity at which target presence is to be queried. Figure 1 shows the general idea of STAP processing.

A space-time processor that computes and applies a separate adaptive weight to every element and pulse is said to be fully adaptive. This requires the solution of MN -dimensional linear system of equations, where N = number of antenna elements and M = number of pulses transmitted in one time interval called Coherent Processing Interval (CPI). Loose ranges for N and M are from 10 to several hundreds. L (range gate) ranges from 500 to 50,000. It is mostly determined by the radar instantaneous bandwidth and the *Pulse Repetition Frequency* (PRF) [13]. For many radar systems, the product MN is likely to range from several hundreds to several thousands. Due to the large amount of computations in the fully adaptive approach, many partially adaptive approaches have been developed, which reduce the dimension by transform or other techniques. They break prohibitively large problems arising in fully adaptive STAP algorithms down into a number of smaller, more manageable adaptive problems while achieving near-optimum performance.

A convenient classification for various types of STAP processors, in terms of four basic categories, is shown in Figure 2 [12]. Each box represents the data for a single range gate after applying a different type of transform. For example, with no transformation, a plane (defined by antenna element axis and PRI axis) of data composes a space-time snapshot. Reducing the problem dimension is possible by utilizing all elements but only a small number of PRIs. Algorithms of this type are referred to as *element-space pre-Doppler* (ESPrD), because full element adaptivity is retained and Doppler processing is done after adaptation. Spatial filtering (beam-forming) may be performed on the element outputs of each PRI prior to adaptation. In this case, the space-time snapshot is

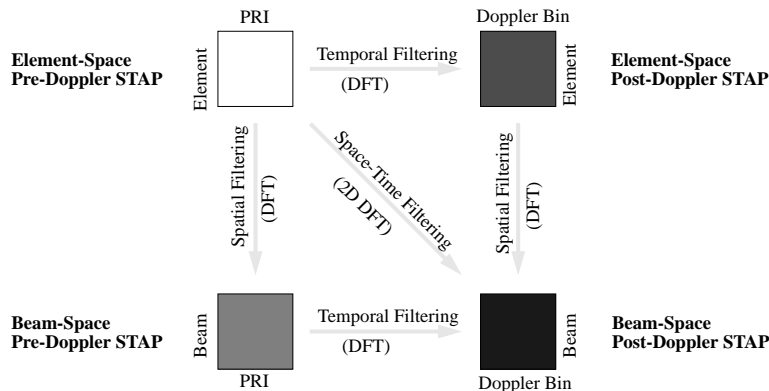


Figure 2: Classification of partially adaptive STAP approaches

transformed into a new snapshot composed of a plane (defined by angle beam axis and PRI axis) of data. Algorithms that perform beam-forming prior to adaptation are referred to as *beam-space* algorithms. STAP approaches that adapt with beam-formed data from a small subset of PRIs are termed *beam-space pre-Doppler* (BSPrD) algorithms as shown by the lower left block of Figure 2. In an analogous fashion, temporal filtering (Doppler processing) may be performed on the data from each array element prior to adaptive processing. This operation transforms the space-time snapshot into a snapshot of Doppler bin and element data. STAP algorithms that operate on a subset of this data are termed *element-space post-Doppler* (ESPD) algorithms, as adaptation occurs after Doppler processing. Post-Doppler algorithms require solving a separate adaptive problem for each Doppler bin. Finally, in the *beam-space post-Doppler* (BSPD) algorithms, both spatial and temporal preprocessing of the data is done prior to adaptation. This processing can be thought of as cascading a beam-former on each PRI with a Doppler filter bank on each beam.

3 Embedded HPC Systems

Most of the modern HPC platforms are coarse grained parallel systems, having a “similar” high level architecture. Examples of such systems are the Intel Paragon, IBM SP-2, and Cray T3D/T3E, among others. Their architecture consists of three main components: (a) *Powerful compute nodes*, often based on the same processors as those used in current-generation uniprocessor workstations. (b) A *low latency high bandwidth network* that interconnects the compute nodes. The network typically consists of high speed point-to-point links and routing switches. (c) *Network interfaces* that couple each processing node with the network links. In these machines, the overheads of performing communication are very high compared with the speed of processors. Therefore, these machines are suitable for coarse grain parallel computations.

A simple and accurate analytic communication model for general HPC platforms is essential to design efficient algorithms. *General purpose Distributed Memory* (GDM) model [11] is used in this paper. The model represents the communication cost of a message passing operation with two parameters: the *startup time* T_s and the *unit transmission time* τ_d . The startup time includes the time for buffer copy, system call invocation, and communication protocol processing. It is incurred once for each transmitted message. The unit transmission time is the cost of transferring a message of unit length over the network. The total transmission time for a message is proportional to the

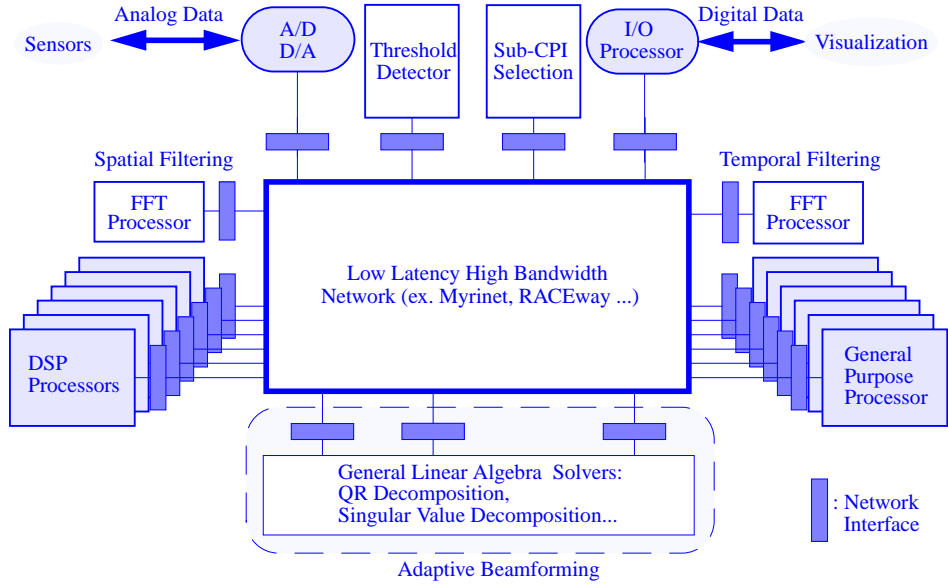


Figure 3: Architecture of a typical Scalable Heterogeneous High Performance Embedded (SHHiPE) system for STAP processing

message size. The total communication time for sending a message of size D data units from one processor to another is $T_s + D \times \tau_d$. T_s is usually much larger than τ_d . For example, the values of T_s and τ_d for the IBM SP2 are $39 \mu\text{sec}$ and $0.035 \mu\text{sec}$ per byte respectively. Thus, it is important to reduce the number of communication steps to minimize the communication time. The model also states that a node can send at most one message and receive at most one message at a time.

When mapping our application (STAP) onto our HPC platform, we assume that for each computation step of STAP a group of processors is dedicated. P_0 processors are used to store the input data received by the antenna elements, P_1 processors for *Beamforming* (BF) and *Doppler Processing* (DP) (FFT operations), and P_2 processors for *Weight Computations* (WC) and *Weight Application* (WA) (solving linear system of equations using *QR* decomposition and computing vector inner product). All communication among the processors in different groups or among the processors within the same group are performed over an interconnection network. The entire system executes in a pipelined fashion. This models the execution of STAP algorithm on systems such as *Scalable Heterogeneous High Performance Embedded* (SHHiPE) system (Figure 3).

4 Parallel Algorithms

In this section, design of high throughput-rate parallel algorithms for STAP is discussed. First, an algorithm design methodology is presented. The methodology is illustrated using an example STAP technique, BSPD STAP. Experimental results on the IBM SP2 are also discussed.

4.1 A Design Methodology

Given an embedded signal processing application, such as STAP, our algorithm design methodology considers the following issues to optimize the throughput performance (Figure 4):

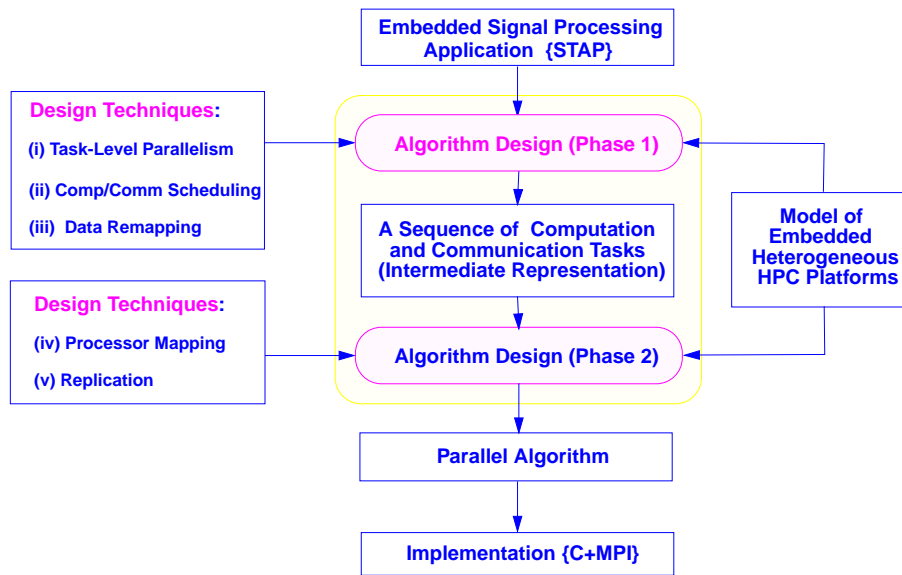


Figure 4: Our Parallel Algorithm Design Methodology

- Granularity of parallelism to be exploited: exploiting fine-grain parallelism or coarse-grain parallelism.
- Scheduling of computations and communications.
- Data remapping: using a fixed data distribution throughout the complete computation steps or remapping data between different computation steps as needed.
- Mapping the application onto the processors for pipelined execution.
- Replication of pipeline stages

Some of the above issues are interrelated with one another.

We exploit coarse-grain, task-level parallelism, because, as indicated in Section 3, the current generation HPC platforms are suitable for coarse grain computations due to their high communication overhead. This requires data remapping between successive computation steps to localize the data needed for each computation step. Thus, the communication phase is separated from the computation phase. We use efficient data remapping algorithms [8, 10] that have been developed by our group for STAP and Synthetic Aperture Radar (SAR).

After applying the above algorithm design techniques, the application is represented as a sequence of computation and communication tasks. We then consider how to map this sequence of tasks onto processors to optimize the throughput performance. We refer to this as processor mapping. We regard the HPC platform as executing in a pipelined fashion. A group of processors is assigned to each pipeline stage. The processors in a group receive data from processors in another group, perform some computation, and send the result to the next group of processors. Communication is performed over an interconnection network. The entire system processes multiple data sets at the same time. An output is produced every T_{period} seconds. For each pipeline stage, computation and communication tasks need to be mapped. The number of processors to

be assigned to each stage must be also determined. For this, the computational requirements of each computation task and the time for each communication task need to be characterized. The computational requirement is estimated by counting the number of floating point operations. The communication time is characterized using the GDM model. Note that our processor mapping technique allows the same group of processors, used for an earlier stage, to be reused for a later stage.

Replication technique is also employed in our methodology. To maximize throughput, computationally demanding stages must be assigned more processors. However, the period of each pipeline stage is determined not only by the computation time but also by the communication time. In STAP, the computation time decreases linearly over a range of processors as more processors are used, because task level parallelism is exploited. However, the communication time is determined by the number of processors assigned to the stage and those assigned to the adjacent stages. For example, assume stage-2 is computationally more demanding than stage-1. Initially, both stages are assumed to be assigned P processors, respectively. As we increase the number of processors assigned to stage-2, the computation time of the stage-2 decreases. However, since the data is distributed from a small number of processors to a large number of processors, the stage-1 to stage-2 communication time increases due to an increase in the number of communication steps.

Instead of increasing the number of processors in stage-2 by r times to process an input data set, we replicate stage-2 r times. In this way, r data sets are processed concurrently by r instances of stage-2. Stage-1 sends the first data set to the first instance of stage-2, the second data set to the second instance, and so on, until the r -th data set is sent to the r -th instance. Note that the communication from stage-1 to stage-2 is restructured such that only as many as stage-1 nodes are used in stage-2 for each instance of the stage. Let $T_{comp}^P(\text{stage-2})$ be the computation time of stage-2 and $T_{comm}^P(\text{stage-2})$ be the communication time of stage-2 with adjacent stages when P processors are used in stage-2. Assuming that stage-2 is the most computationally demanding stage, the periodic time interval of the entire system (T_{period}) is determined by this stage: $T_{period} = T_{comp}^P(\text{stage-2}) + T_{comm}^P(\text{stage-2})$. By replicating stage-2 r times, T_{period} is reduced to $\frac{T_{comp}^P(\text{stage-2}) + T_{comm}^P(\text{stage-2})}{r}$. Therefore, T_{period} is reduced by a factor of r . Without replication, $T_{period} = T_{comp}^{r \times P}(\text{stage-2}) + T_{comm}^{r \times P}(\text{stage-2})$. Compared with $T_{comp}^P(\text{stage-2})$, $T_{comp}^{r \times P}(\text{stage-2})$ can be smaller by r times. However, $T_{comm}^{r \times P}(\text{stage-2})$ is larger compared with $T_{comm}^P(\text{stage-2})$. Replication results in shorter T_{period} , thus, better throughput than no replication.

Consider a two stage pipeline (See Figure 5 (a)). Assume that each of the two stages is assigned P processors. Processors assigned to stage-1 do not perform any computation. They are only involved in the communication with the processors assigned to stage-2. Processors in stage-2 perform some computation. The T_{period} of this pipeline is $T_{comp}^P(\text{stage-2}) + T_{comm}^P(\text{stage-2})$. Replicating stage-2 two times leads to $T_{period} = \frac{T_{comp}^P(\text{stage-2}) + T_{comm}^P(\text{stage-2})}{2}$ (See Figure 5 (b)). T_{period} is further reduced when stage-2 is replicated four times (See Figure 5 (c)). The benefit of replication is greater when the replication factor is higher. However, replicating stage-2 more than four times idles some processors as indicated in Figure 5 (d). In this case, stage-2 is replicated six times. When the processors in the first instance of stage-2 complete their computations, they are ready to receive the next set of data from the processors in stage-1. However, at that moment, stage-1 processors start to send a set of data to processors in the fifth instance of stage-2. This leads to processors assigned to the first instance idle until stage-1 processors complete sending a set of data to the processors in the sixth instance. Thus, for the example considered, the maximum benefit of replication is achieved when stage-2 is replicated four times. In this case, T_{period} is $T_{comm}^P(\text{stage-2})$.

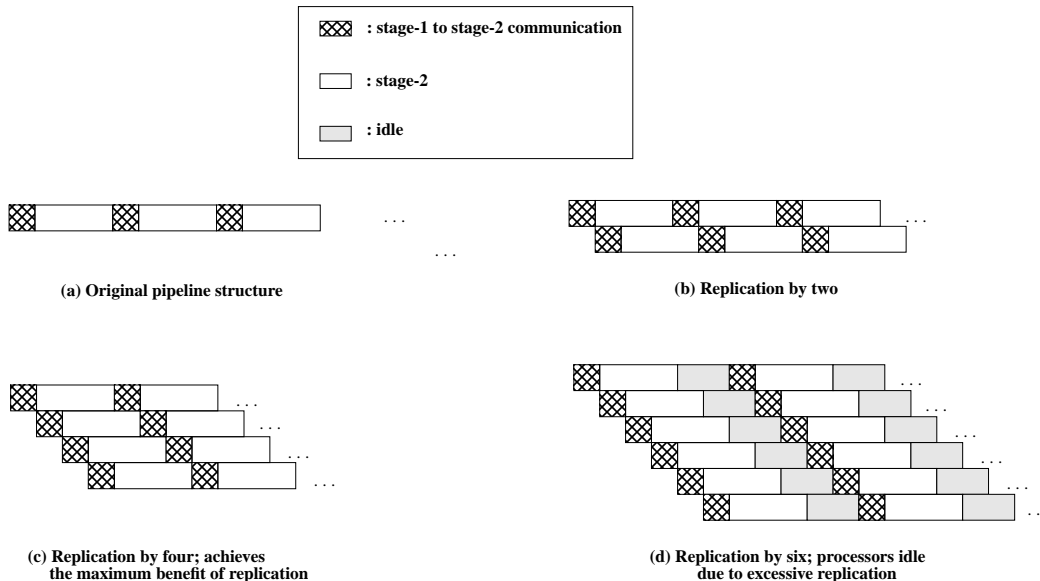


Figure 5: Examples illustrating the benefit of replication and the effect of excessive replication

An extreme case of the parallel algorithm design is as follows: given a total of P processors, assign a single processor to perform all the computation steps corresponding to an input data set. Thus, after the initial transient period, P input data sets are processed concurrently. T_{period} , in this case, is $\frac{T_{comp}^1 + T_{comm}^1}{P}$, where T_{comm}^1 is the time to send a set of input data from a number of sensor nodes to a single processor. The minimum possible value for T_{period} is T_{comm}^1 . However, this design requires a large amount of memory for each processor to store one complete set of input data. This is impractical for embedded signal processing applications. Furthermore, the minimum T_{period} that this design can achieve (T_{comm}^1) can be significantly larger than when a number of processors are used to process each data input. This is because a set of data collected at a number of sensor nodes must be sent to a single processor. Also, notice that $T_{latency}$ of this design is high.

Replication is also a viable technique to improve throughput performance in cases where the amount of available task level parallelism in a stage is limited. In such cases, increasing the number of processors to reduce the period can actually increase the period of the stage due to interprocessor communication within a stage in executing a task over several processors.

4.2 An Example: Parallel Algorithms for BSPD STAP

Using the methodology described above, we present a parallel algorithm for BSPD STAP. For data input, we assume that each antenna element receives a plane (defined by range gate axis and PRI axis) of data in each CPI. Then, a sequence of computation steps are performed on the input data. They are Beamforming (BF), Doppler Processing (DP), Weight Computation (WC), and Weight Application (WA). The BF step involves FFT operations along the spatial dimension (antenna elements). Since there are N antenna elements, we need M N -point FFT's for each space-time snapshot. $O(N \log(N))$ time is needed on a uniprocessor for each N point FFT. Therefore, we need $O(MN \log(N))$ computations per snapshot. Thus, the total complexity is $O(LMN \log(N))$. The DP step involves FFT operations along the temporal dimension (PRI). In all, $O(LNM \log(M))$

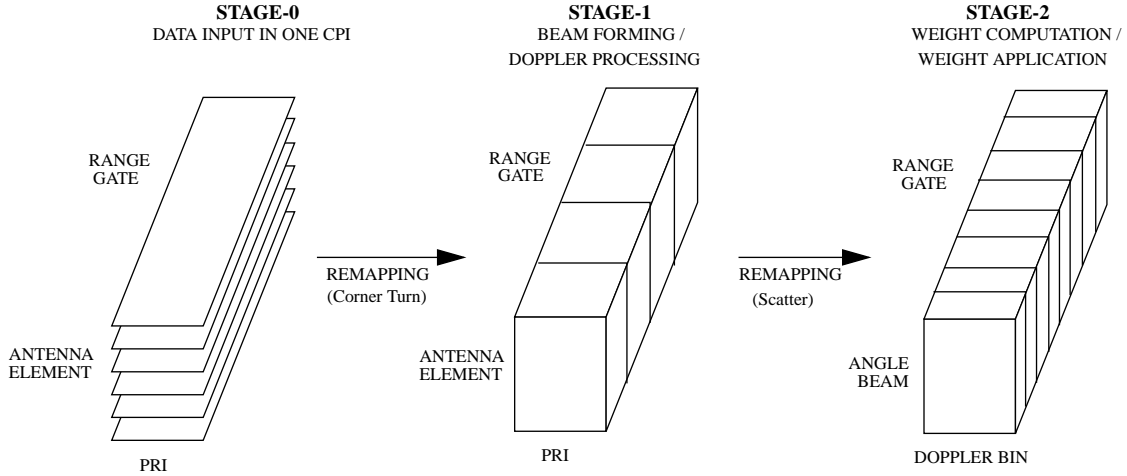


Figure 6: Data access pattern for each step of BSPD STAP

computations are required for DP. After the BF and DP, the space-time snapshot is transformed into a new snapshot composed of a plane of data defined by angle beam axis and Doppler bin axis, as explained in Section 2.

The WC step selects a subplane from the new snapshot for computing the weights. For each Doppler bin, K_s beam outputs and K_t adjacent Doppler bins (including itself) are selected to form a K by K matrix, where $K = K_s \times K_t$, $K_s \ll N$, and $K_t \ll M$. Typical values for K_s and K_t are 2 to 3. K_t vectors of K_s elements are concatenated to form a vector V of length $K_s \times K_t$. An outer product of the vector V with itself is computed to form a K by K matrix. This matrix is used for computing the weights. It involves solving a linear system of equations using QR decomposition. QR decomposition of an M by N matrix requires $O(MN^2)$ computations. Since we compute the weights for $(M - K_t + 1)$ Doppler bins, $O((M - K_t + 1)K^3)$ computations are needed for each snapshot. Therefore, in all, $O(L(M - K_t + 1)K^3)$ computations are needed. In the WA step, computed weights are applied to the collected sample points. It involves performing vector inner products. Thus, the resulting values are scalars. These scalar values are used in the next step (target detection). This step is not computationally demanding. We ignore this in our algorithm design.

Task level parallelism is exploited for each computation step. Data remapping is needed between successive computation steps, if successive steps have different data access patterns. In the data input step, each antenna element receives a plane of data defined by PRI axis and range gate axis. The BF step accesses data along antenna element axis. The DP step accesses data along PRI axis. The WC and WA steps access a subplane of data defined by angle beam axis and Doppler bin axis. Therefore, for BF, DP, WC, and WA, we allocate a data subcube of size $N \times M \times L/P_i$, where P_i is the number of processors used in each step. We regard the HPC platform as executing in a pipelined fashion. We need to schedule the computations and communications, and assign a group of processors for each stage (See Figure 6). Data input by each antenna element composes one pipeline stage (stage-0). We assign $P_0 (= N)$ processors to stage-0. BF and DP are FFT operations. Therefore, they are combined into one pipeline stage and are assigned a group of processors. This is denoted as stage-1. P_1 processors are assigned to stage-1. WC and WA are also

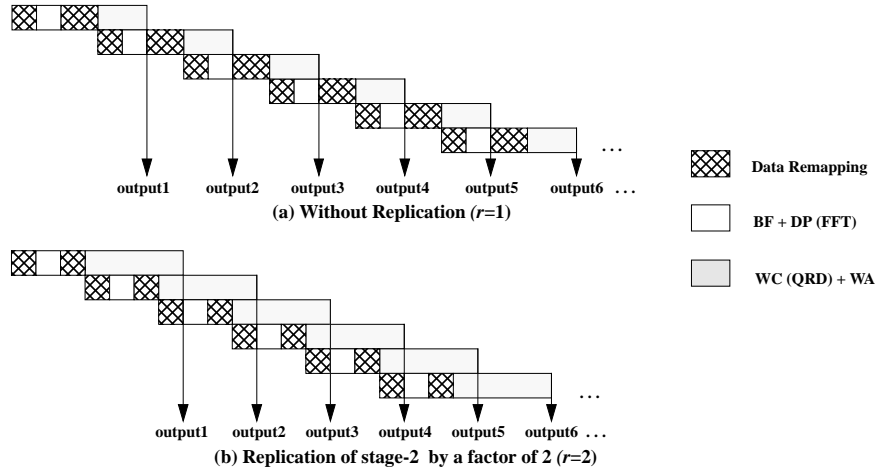


Figure 7: Illustration of pipelined execution with and without replication

combined into one stage (stage-2) and are assigned the same group of processors. Since WC using QR decomposition is the most computationally demanding step, larger number of processors are needed. Let P_2 denote the number of processors assigned to this stage.

Data is communicated between successive stages. Between stage-0 and stage-1, a data remapping which is a *Corner Turn* is needed. An efficient algorithm was developed [10] for this remapping problem, where the number of receivers \gg the number of senders. However, since we don't need a much larger number of processors for stage-1 than for stage-0, we can use a simpler algorithm. As more processors are assigned to stage-2 than stage-1, data assigned to a single processor of stage-1 needs to be scattered to multiple processors in stage-2. We use a tree-based algorithm to perform this data movement.

Next, we consider replication to improve the throughput performance. Increasing the number of processors assigned to stage-2 from P_2 to P'_2 will decrease the computation time almost linearly, because task level parallelism is exploited. Communication time must also decrease linearly to achieve scalable performance. However, the communication time increases. Therefore, T_{period} does not decrease linearly. Replicating stage-2 r times, where $r = P'_2/P_2$, rather than involving all of P'_2 processors in a single instance of stage-2 can effectively decrease T_{period} . By using replication, $T_{period} = \frac{T_{comp}^P(\text{stage-2}) + T_{comm}^P(\text{stage-2})}{P'_2/P_2}$. Figure 7 illustrates the pipelined execution of BSPD STAP with and without replication.

4.3 Experimental Results

Our algorithm described in Section 4.2 was implemented on the IBM SP2 at the Maui High Performance Computing Center. We used up to 228 processors in our experiments. We measured the time period, T_{period} , of successive outputs, as the number of processors was increased. Based on this, the number of outputs per second was computed. Our code was written using C and MPI. Therefore, it is portable across various HPC platforms.

We assume that a data cube of size 12 (antenna elements) \times 64 (PRIs) \times 576 (range gates) is received in T_{period} seconds. This data cube size is chosen to reflect a real situation. For example, in the MITRE RT-STAP benchmark [3], a data cube size of 16 \times 64 \times 630 is assumed. Our data cube

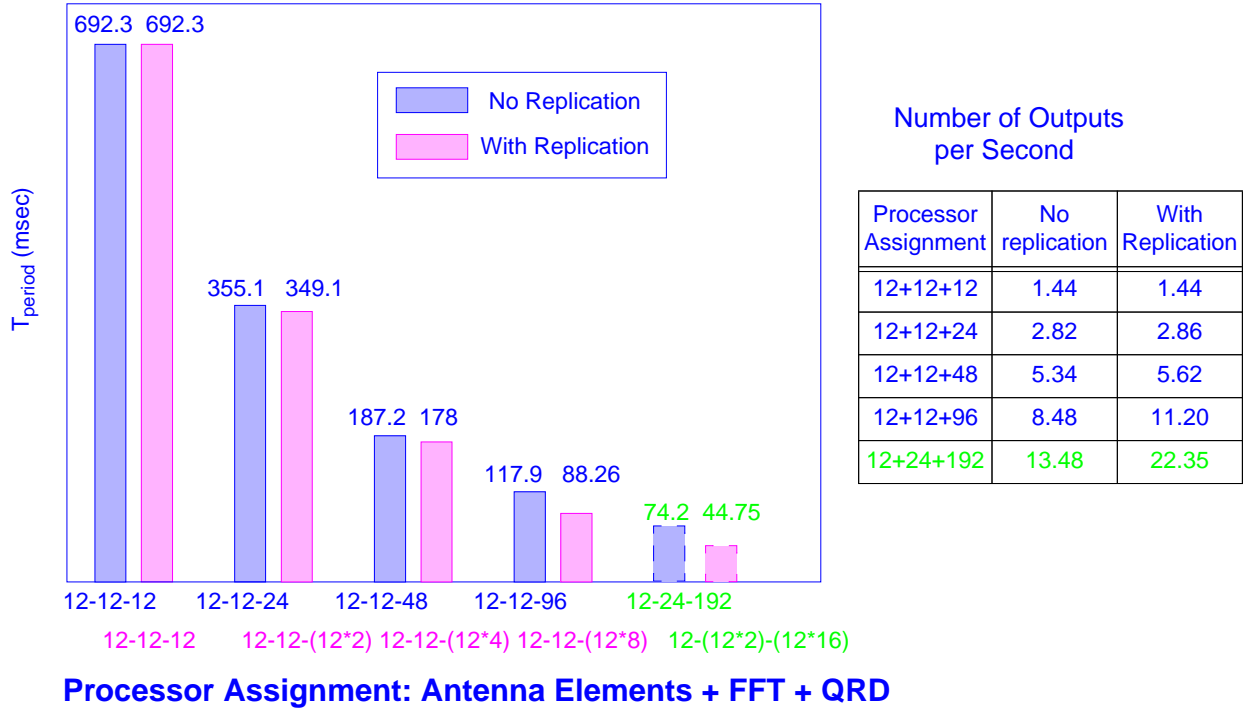


Figure 8: Experimental results on SP2

size is decided by considering this real situation and for efficient use of the available processors. Since 12 antenna elements are assumed, we assign 12 processors to stage-0. To simulate the data input from each antenna element, we assume that the data is accumulated in the memory of the 12 processors. We assign 12 processors to stage-1 for BF and DP. The rest of the processors are assigned to stage-2. The numbers of processors assigned to stage-2 are 12, 24, 48, and 96. Moving some of the processors from stage-1 to stage-2 improves the computation time for stage-2 marginally. However, it increases the communication time between stage-1 and stage-2 (and also the time between stage-0 and stage-1). This cancels out the improvements in computation time. A simple data remapping algorithm is used between stage-0 and stage-1. It involves $(12 - 1 = 11)$ communication steps. A data scatter is performed between stage-1 and stage-2. Replication is also used. For P_2 processors assigned to stage-2, they are partitioned into groups of 12 processors. Therefore, stage-2 is replicated $P_2/12$ times.

Figure 8 shows the implementation results for IBM SP2. Using T_{period} , the number of outputs per second is calculated. Stage-2 has the highest computational requirement. Thus, T_{period} is determined by the computation and the communication time taken by stage-2. T_{period} decreases linearly up to 72 (12+12+48) processors. Therefore, our methodology is very effective in this range. However, without replication, T_{period} decreases only by 59%, when 120 (12+12+96) processors are used compared with when 72 (12+12+48) processors are used. With replication, it decreases linearly with P_2 . In case of 120 processors, replication results in 32% improvement in T_{period} over no replication. To further improve T_{period} using additional processors, we assign more processors to stage-1 (24) and to stage-2 (192). A total of 228 processors were used. In this case, stage-1 is also replicated. Without replication, T_{period} decreases by only 59%, when 228 (12+24+192)

processors are used compared with the case when 120 (12+12+96) processors are used. With replication, T_{period} decreases linearly with P_2 . Replication results in 65% improvement in T_{period} over no replication.

5 Conclusion

We have presented a methodology to design parallel algorithms for throughput-oriented processing of STAP on HPC platforms. Throughput is one of the key requirements of STAP processing, where the input data arrives in a continuous fashion. We used a model of HPC platforms that includes the cost of communication. The algorithm design methodology exploits task level parallelism. Data remapping is used between successive computation steps. We regard the HPC platform as executing in a pipelined fashion. We assigned a group of processors to each pipeline stage. The computational requirements of each computation task and the time for each communication task were first estimated. Based on these, we mapped computation and communication tasks onto each pipeline stage. To further improve the performance, replication technique was employed. In this technique, the available processors in a stage were partitioned into sets. Each set of processors was assigned to process an input data sets. Experimental results on the IBM SP2 show that the resulting parallel algorithm linearly decreases the time period of the pipeline as we increase the number of processors. Our code is written using C and MPI, and is portable across HPC platforms.

6 Acknowledgment

The implementations reported in this paper were performed on the IBM SP2 at the Maui High Performance Computing Center (MHPCC). Research at MHPCC is sponsored in part by the Phillips Laboratory, Air Force Material Command, USAF, under cooperative agreement number F29601-93-2-0001.

References

- [1] P. B. Bhat, Y. W. Lim, and V. K. Prasanna, "Issues in using Heterogeneous HPC Systems for Embedded Real Time Signal Processing Applications", Proc. of 2nd Intl. Workshop on Real-Time Computing Systems and Applications, Tokyo, Japan, October 1995.
- [2] R. A. Games, "Benchmarking Methodology for Real-Time Embedded Scalable High Performance Computing," MITRE Technical Report MTR 96B0000010, March 1996.
- [3] R.A. Games, J.A. Torres, and R.T. Williams, "RT_STAP: Real-Time Space-Time Adaptive Processing Benchmark", MITRE Corporation, June 1996.
- [4] W. Gropp, E. Lusk, and A. Skjellum, "Using MPI: Portable Parallel Programming with the Message Passing Interface", MIT Press, 1994.
- [5] J. Hennessy and D. Patterson, "Computer Architecture: A Quantitative Approach", Morgan Kaufman Publishers, Second Edition, 1996.
- [6] K. Hwang, Z. Xu, and M. Arakawa, "Benchmark Evaluation of IBM SP2 for Parallel Signal Processing", IEEE Transactions on Parallel and Distributed Systems, June, 1996.

- [7] Y. W. Lim and V. K. Prasanna, “Scalable Portable Implementations of Space-Time Adaptive Processing”, 10th High Performance Computers, Ottawa, Canada, 1996.
- [8] Y. W. Lim, P. B. Bhat, and V. K. Prasanna, “Efficient Data Remapping Algorithms for Embedded Signal Processing Applications”, 10th High Performance Computers, Ottawa, Canada, 1996.
- [9] S.J. Olszanskyj, J.M. Lebek, and A.W. Bojanczyk, “Parallel Algorithms for Space-Time Adaptive Processing”, International Parallel Processing Symposium '95, pp. 77-81, Apr. 1995.
- [10] J. Suh and V.K. Prasanna, “Scalable and Portable Implementations of Real-Time Signal Processing Benchmarks on HPC Platforms”, submitted for publication.
- [11] C.-L. Wang, “High Performance Computing for Vision on Distributed Memory Machines”, Ph.D. Thesis, University of Southern California, August 1995.
- [12] J. Ward, “Space-Time Adaptive Processing for Airborne Radar”, MIT Lincoln Lab. Technical Report 1015, Dec. 1994.
- [13] J. Ward, Private Communication.