

Design, Implementation and Evaluation of Parallel Pipelined STAP on Parallel Computers

ALOK CHOUDHARY

WEI-KENG LIAO
Northwestern University

DONALD WEINER, Life Fellow, IEEE

PRAMOD VARSHNEY, Fellow, IEEE
Syracuse University

RICHARD LINDERMAN, Senior Member, IEEE

MARK LINDERMAN, Member, IEEE

RUSSELL BROWN, Fellow, IEEE
Air Force Research Laboratory

Performance results are presented for the design and implementation of parallel pipelined space-time adaptive processing (STAP) algorithms on parallel computers. In particular, the issues involved in parallelization, our approach to parallelization, and performance results on an Intel Paragon are described. The process of developing software for such an application on parallel computers when latency and throughput are both considered together is discussed and tradeoffs considered with respect to inter and intratask communication and data redistribution are presented. 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. Results are presented for up to 236 compute nodes (limited by the machine size available to us). Another interesting observation made from the implementation results is that performance improvement due to the assignment of additional processors to one task can improve the performance of other tasks without any increase in the number of processors assigned to them. Normally, this cannot be predicted by theoretical analysis.

Manuscript received January 29, 1999; revised June 9 and December 6, 1999; released for publication December 11, 1999.

IEEE Log No. T-AES/36/2/05228.

Refereeing of this contribution was handled by W. D. Blair.

This work was supported by Air Force Material Command under Contract F30602-97-C-0026.

Authors' addresses: A. Choudhary and W. Liao, Electrical and Computer Engineering Dept., Northwestern University, Evanston, IL 60208; D. Weiner and P. Varshney, Electrical Engineering and Computer Science Dept., Syracuse University, 121 Link Hall, Syracuse, NY 13244; R. Linderman and M. Linderman, AFRL/IFTC, 26 Electronics Parkway, Air Force Research Laboratory, Rome, NY 13441; R. Brown, AFRL/SNRT, 26 Electronics Parkway, Air Force Research Laboratory, Rome, NY 13441.

0018-9251/00/\$10.00 © 2000 IEEE

I. INTRODUCTION

Space-time adaptive processing (STAP) is a well-known technique in the area of airborne surveillance radars used to detect weak target returns embedded in strong ground clutter, interference, and receiver noise. STAP is a 2-dimensional adaptive filtering algorithm that attenuates unwanted signals by placing nulls in their directions of arrival and Doppler frequencies. Most STAP applications are computationally intensive and must 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 processing steps is challenging and is the subject of this paper.

Described here is our innovative parallel pipelined implementation of a pulse repetition interval (PRI)-staggered post-Doppler STAP algorithm on the Intel Paragon at the Air Force Research Laboratory (AFRL), Rome, NY. For a detailed description of the STAP algorithm implemented in this work, the reader is referred to [1, 2]. AFRL successfully installed their implementation of the STAP algorithm onboard an airborne platform and performed four flight experiments in May and June 1996 [3]. These experiments were performed as part of the Real-Time Multi-Channel Airborne Radar Measurements (RTMCARM) program. The RTMCARM system block diagram is shown in Fig. 1. In that real-time demonstration, live data from a phased-array radar was processed by the onboard Intel Paragon and results showed that high-performance computers can deliver a significant performance gain. However, this implementation used compute nodes of the machine only as independent resources in a round robin fashion to run different instances of STAP (rather than speeding up each instance of STAP.) Using this approach, the throughput may be improved, but the latency is limited by what can be achieved using one compute node.

Parallel computers, organized with a large set (several hundreds) of processors linked by a specialized high speed interconnection network, offer an attractive solution to many computationally intensive applications, such as image processing, simulation of particle reactions, and so forth. Parallel processing splits an application problem into several subproblems which are solved on multiple processors simultaneously. To learn more about parallel computing, the reader is referred to [4–8]. For our parallel implementation of this real application we have designed a model of the parallel pipeline system

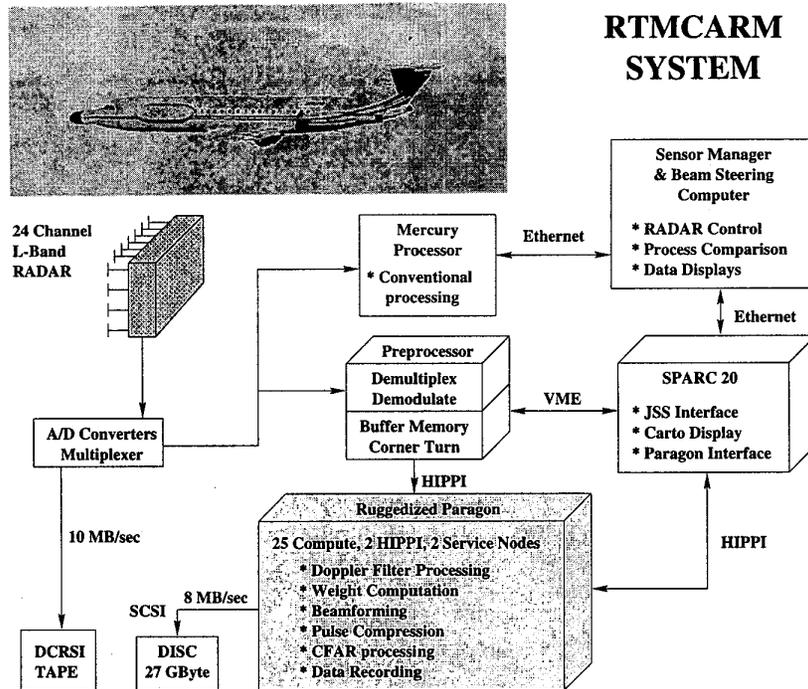


Fig. 1. RTMCARM system block diagram.

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.

This paper discusses both the parallelization process and performance results. In addition, design considerations for portability, task mapping, parallel data redistribution, parallel pipelining as well as system-level and task-level performance measurement are presented. Finally, the performance and scalability of the implementation for a large number of processors is demonstrated. Performance results are given for the Intel Paragon at AFRL.

The paper is organized as follows. In Section II we discuss the related work. An overview of the implemented algorithm is given in Section III. In Section IV we present the parallel pipeline system model and discuss some parallelization issues and approaches for implementation of STAP algorithms. Section V presents specific details of STAP implementation. Software development is presented in Section VI. Performance results and conclusions are presented in Sections VII and VIII, respectively.

II. RELATED WORK

The RTMCARM experiments were performed using a BAC 1-11 aircraft. The radar was a phased-array L-Band radar with 32 elements organized into two rows of 16 each. Only the data

from the upper 16 elements were processed with STAP. This data was derived from a 1.25 MHz IF signal that was 4 : 1 oversampled at 5 MHz. The number representation at IF was 14 bits, 2s complement and was converted to 16 bit baseband real and imaginary numbers. Special interface boards were used to digitally demodulate IF signals to baseband. The signal data formed a raw 3-dimensional data cube, called the coherent processing interval (CPI) data cube, comprised of 128 pulses, 512 range gates (32.8 mi), and 16 channels. These special interface boards were also used to corner turn the data cube so that the CPI is unit stride along pulses. This speeds the subsequent Doppler processing on the high performance computing (HPC) systems. Live CPI data from a phased-array radar were processed by a ruggedized version of the Paragon computer.

The ruggedized version of the Intel Paragon system used for the RTMCARM experiments consists of 25 compute nodes running the SUNMOS operating system. Fig. 2 depicts the system implementation. Each compute node has three i860 processors accessing the common memory of size 64M bytes as a shared resource. The CPI data sets were sent to the 25 compute nodes in a round robin manner and all three processors worked on each CPI data set as a shared-memory machine. The system processed up to 10 CPIs/s (throughput) and achieved a latency of 2.35 s/CPI. This implementation used compute nodes of the machine as independent resources to run different instances of CPI data sets. No communication

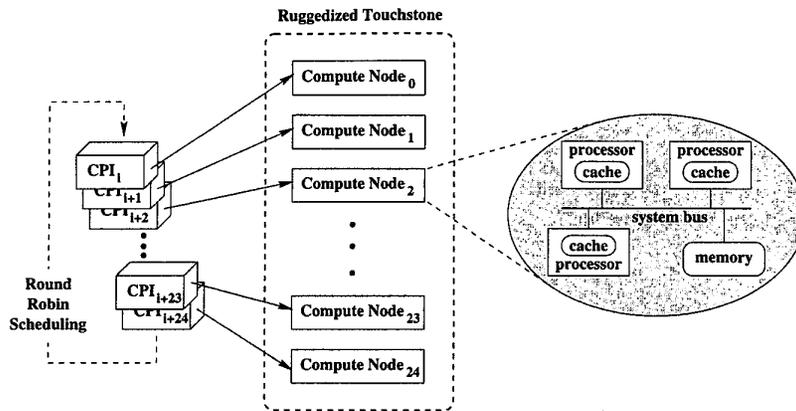


Fig. 2. Implementation of ruggedized version of Intel Paragon System in RTMCARM experiments.

among compute nodes was needed. This approach can achieve desired throughput by using as many nodes as needed, but the latency is limited by what can be achieved using the three processors in one compute node. More information on the overall system configuration and performance results can be found in [1, 3].

Other related work [9–12] parallelized high-order post-Doppler STAP algorithms by partitioning the computational workload among all processors allocated for the applications. In [9, 10], the work focused on the design of parallel versions of subroutines for fast Fourier transform (FFT) and QR decomposition. In [11, 12], the implementations optimized the data redistribution between processing steps in the STAP algorithms while using sequential versions of the FFT and QR decomposition subroutines. A multistage approach was employed in [13] which was an extension of [11, 12]. A beam space post-Doppler STAP was divided into three stages and each stage was parallelized on a group of processors. A technique called replication of pipeline stages was used to replicate the computational intensive stages such that a different data instance is run on a different replicated stage. Their effort focused on increasing the throughput while keeping the latency fixed. For other related work, the reader is referred to [14–16].

III. ALGORITHM OVERVIEW

The adaptive algorithm, which cancels Doppler-shifted clutter returns as seen by the airborne radar system, is based on a least squares solution to the weight vector problem. This approach has traditionally yielded high clutter rejection but suffers from severe distortions in the adapted mainbeam pattern and resulting loss of gain on the target. Our approach, which is described in greater detail in the Appendix, introduces a set of constraint equations into the least squares problem which can be weighted

proportionally to preserve mainbeam shape. The algorithm is structured so that multiple receive beams may be formed without changing the matrix of training data. Thus, the adaptive problem can be solved once for all beams which lie within the transmit illumination region. The airborne radar system was programmed to transmit five beams, each 25 deg in width, spaced 20 deg apart. Within each transmit beam, six receive beams were formed by the processor.

A MATLAB version of the code which was parallelized is presented in the Appendix. The algorithm consists of the following steps.

- 1) Doppler filter processing.
- 2) Weight computation.
- 3) Beamforming.
- 4) Pulse compression.
- 5) CFAR processing.

Doppler filtering is performed on each receive channel using weighted FFTs. The analog portion of the receiver compensates the received clutter frequency to center the clutter frequency at zero regardless of the transmit beam position. This simplifies indexing of Doppler bins for classification as “easy” or “hard” depending on their proximity to mainbeam clutter returns. For the hard cases, Doppler processing is performed on two 125-pulse windows of data separated by three pulses (a STAP technique known as “PRI-stagger”). Both sets of Doppler processed data are adaptively weighted in the beamforming process for improved clutter rejection. In the easy case, only a single Doppler spectrum is computed. This simpler technique has been termed post-Doppler adaptive beamforming and is quite effective at a fraction of the computational cost when the Doppler bin is well separated from mainbeam clutter. In these situations, an angular null placed in the direction of the competing ground clutter provides excellent rejection. Selectable window functions are applied to the data prior to

the Doppler FFTs to control sidelobe levels. The selection of a window is a key parameter in that it impacts the leakage of clutter returns across Doppler bins, traded off against the width of the clutter passband.

An efficient method of beamforming using recursive weight updates is made possible by a block update form of the QR decomposition algorithm. This is especially significant in the hard Doppler regions, which are computed using separate weights for six consecutive range intervals. The recursive algorithm requires substantially less training data (sample support) for accurate weight computation, as well as providing improved efficiency. Since the hard regions have one-sixth the range extent from which to draw data, this approach dealt with the paucity of data by using past looks at the same azimuth, exponentially forgotten, as independent, identically distributed estimates of the clutter to be canceled. This assumes a reasonable revisit time for each azimuth beam position. During the flight experiments, the five 25 deg transmit beam positions were revisited at a 1–2 Hz rate (5–10 CPIs/s).

The training data for the easy Doppler regions was selected using a more traditional approach. Here, the entire range extent was available for sample support, so the entire training set was drawn from three preceding CPIs for application to the next CPI in this azimuth beam position. In this case, a regular (nonrecursive) QR decomposition is performed on the training data, followed by block update to add in the beam shape constraints.

Pulse compression is a compute intensive task, especially if applied to each receive channel independently. In general, this approach is required for adaptive algorithms which compute different weight sets as a function of radar range. Our algorithm, however, with its mainbeam constraint, preserves phase across range. In fact, the phase of the solution is independent of the clutter nulling equations, and appears only in the constraint equations. The adapted target phase is preserved across range, even though the clutter and adaptive weights may vary with range. Thus, pulse compression may be performed on the beamformed output of the receive channels providing a substantial savings in computations.

In the sections to follow, we present the process of parallelization and software design considerations including those for portability, task mapping, parallel data redistribution, parallel pipelining and issues involved in measuring performance in implementations when not only the performance of individual tasks is important, but overall performance of the integrated system is critical. We demonstrate the performance and scalability for a large number of processors.

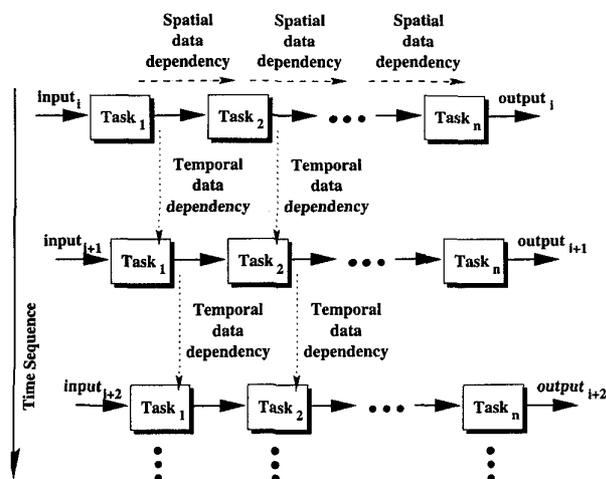


Fig. 3. Model of parallel pipeline system. (Note that Task_{*i*} for all input instances is executed on same number of processors, but that the number of processors may differ from one task to another.)

IV. MODEL OF PARALLEL PIPELINED SYSTEM

The system model for the type of STAP applications considered in this work is shown in Fig. 3. A pipeline is a collection of tasks which are executed sequentially. The input to the first task is obtained normally from sensors or other input devices with the inputs to the remaining tasks coming from outputs of 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 task that is parallelized on multiple (different number of) processors. That is, each task is decomposed into subtasks to be performed in parallel. Therefore, each pipeline is a collection of parallel tasks.

In such a system, there exist both spatial and temporal parallelism that result in two types of data dependencies and flows, namely, spatial data dependency and temporal data dependency [17–19]. Spatial data dependency can be classified into intertask data dependency and intratask data dependency. Intratask data dependencies arise when a set of subtasks needs to exchange intermediate results during the execution of a parallel task in a pipeline. Intertask data dependency is due to the transfer and reorganization of data passed onto the next parallel task in the pipeline. Inter-task communication can be communication from the subtasks of the current task to the subtasks of the next task, or collection and reorganization of output data of the current task and then redistribution of the data to the next task. The choice depends on the underlying architecture, mapping of algorithms and input-output relationship between consecutive tasks. 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. STAP is an

interesting parallelization problem because it exhibits both types of data dependency.

A. Parallelization Issues and Approaches

A STAP algorithm involves multiple algorithms (or processing steps), each of which performs particular functions, to be executed in a pipelined fashion. Multiple pipelines need to be executed in a staggered manner to satisfy the throughput requirements. Each task needs to be parallelized for the required performance, which, in turn, requires addressing the issue of data distribution on the subset of processors on which a task is parallelized to obtain good efficiency and incur minimal communication overhead. Given that each task is parallelized, data flow among multiple processors of two or more tasks is required and, therefore, communication scheduling techniques become critical.

1) *Intertask Data Redistribution*: In an integrated system, data redistribution is required to feed data from one parallel task to another, because the way data is distributed in one task may not be the most appropriate distribution for the next task for algorithmic or efficiency reasons. For example, the FFTs in the Doppler filter processing task perform optimally when the data is unit-stride in pulse, while the next stage, beamforming, performs optimally when the data is unit stride in channel. To ensure efficiency and continuity of memory access, data reorganization and redistribution are required in the inter-task communication phase. Data redistribution also allows concentration of communication at the beginning and the end of each task.

We have developed runtime functions and strategies that perform efficient data redistribution [20]. These techniques reduce the communication time by minimizing contention on the communication links as well as by minimizing the overhead of processing for redistribution (which adds to the latency of sending messages). We take advantage of lessons learned from these techniques to implement the parallel pipelined STAP application.

2) *Task Scheduling and Processor Assignment*: An important factor in the performance of a parallel system is how the computational load is mapped onto the processors in the system. Ideally, to achieve maximum parallelism, the load must be evenly distributed across the processors. The problem of statically mapping the workload of a parallel algorithm to processors in a distributed memory system has been studied under different problem models, such as [21, 22]. The mapping policies are adequate when an application consists of a single task, and the computational load can be determined statically. These static mapping policies do not model applications consisting of a sequence of tasks

(algorithms) where the output of one task becomes the input to the next task in the sequence.

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 [23]. When several parallel tasks need to be executed in a pipelined fashion, tradeoffs exist between assigning processors to maximize the overall throughput and assigning processors to minimize the response time (or latency) of a single data set. 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.

To reduce the latency, each parallel task must be allocated more processors to reduce its execution time, and consequently, the overall execution time of the integrated system. But it is well known that the efficiency of parallel programs usually decreases as the number of processors is increased. Therefore, the gains in this approach may be incremental. On the other hand, throughput can be increased by increasing the latency of individual tasks by assigning them fewer processors and, therefore, increasing efficiency, but at the same time having multiple streams active concurrently in a staggered manner to satisfy the input-data rate requirements. We next present these tradeoffs and discuss various implementation issues.

V. DESIGN AND IMPLEMENTATION

The design of the parallel pipelined STAP algorithm is shown in Fig. 4. The parallel pipeline system consists of seven basic tasks. We refer to the parallel pipeline as simply a pipeline in the rest of this paper. The input data set for the pipeline is obtained from a phased-array radar and is formed in terms of a CPI. Each CPI data set is a 3-dimensional complex data cube comprised of K range cells, J channels, and N pulses. The output of the pipeline is a report on the detection of possible targets. The arrows shown in Fig. 4 indicate data transfer between tasks. Although a single arrow is shown, note that each represents multiple processors in one task communicating with multiple processors in another task. Each task i is parallelized by evenly partitioning its work load among P_i processors. The execution time associated with task i , T_i , consists of the time to receive data from the previous task, computation time, and time to send results to the next task.

The calculation of weights is the most computationally intensive part of the STAP algorithm. For the computation of the weight vectors for the current CPI data cube, data cubes from previous

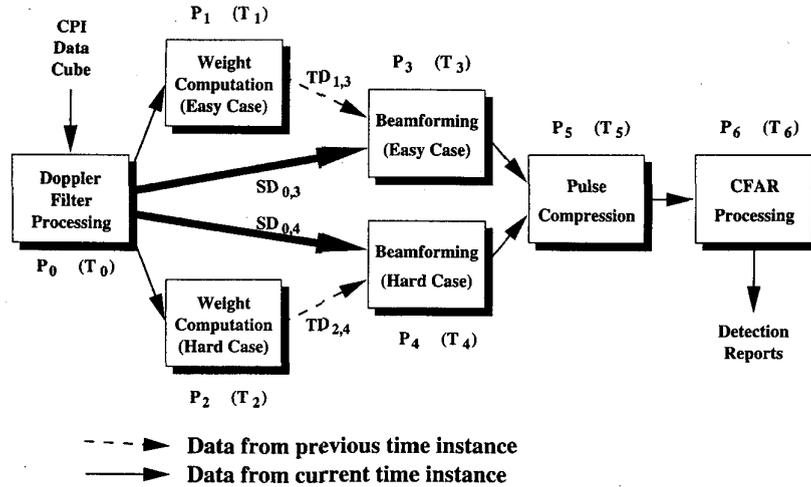


Fig. 4. Implementation of parallel pipelined STAP. Arrows connecting task blocks represent data transfer between tasks.

CPIs are used as input data. This introduces temporal data dependency. For example, suppose that a set of CPI data cubes entering the pipeline sequentially are denoted by CPI_i , $i = 0, 1, \dots$. At any time instance i , the Doppler filtering task is processing CPI_i and the beamforming task is processing CPI_{i-1} . In the meanwhile, the weight computation task is using past CPIs in the same azimuthal direction to calculate the weight vectors for CPI_i , as described below. The computed weight vectors will be applied to CPI_i in the beamforming task at the next time instance ($i + 1$). Thus, temporal data dependencies exist and are represented by arrows with dashed lines, $TD_{1,3}$ and $TD_{2,4}$, in Fig. 4 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 in Fig. 4 by arrows with solid lines.

Throughput and latency are two important measures for performance evaluation on a pipeline system. The throughput of our pipeline system is the inverse of the maximum execution time among all tasks, i.e.,

$$\text{throughput} = \frac{1}{\max_{0 \leq i < 7} T_i}. \quad (1)$$

To maximize the throughput, the maximum value of T_i should be minimized. In other words, no task should have an extremely large execution time. With a limited number of processors, the processor assignment to different tasks must be made in such a way that the execution time of the task with highest computation time is reduced.

The latency of this pipeline system is the time between the arrival of the CPI data cube at the system input and the time at which the detection report is available at the system output. Therefore, the latency for processing one CPI is the sum of the execution times of all the tasks except weight computation tasks,

i.e.,

$$\text{latency} = T_0 + \max(T_3, T_4) + T_5 + T_6. \quad (2)$$

Equation (2) does not contain T_1 and T_2 . The temporal data dependency does not affect the latency because weight computation tasks use data from the previous instance of CPI data rather than the current CPI. The filtered CPI data cube sent to the beamforming tasks does not wait for the completion of its weight computation but rather for the completion of the weight computation of the previous CPI. For example, when the Doppler filter processing task is processing CPI_i , the weight computation tasks use the filtered CPI data, CPI_{i-1} , to calculate the weight vectors for CPI_i . At the same time, the beamforming tasks are working on CPI_{i-1} using the data received from the Doppler filter processing and weight computation tasks. The beamforming tasks do not wait for the completion of the weight computation task when processing CPI_{i-1} data. The overall system latency can be reduced by reducing the execution times of the parallel tasks, e.g., T_0 , T_3 , T_4 , T_5 , and T_6 in our system.

Next, we briefly describe each task and its parallel implementation. A detailed description of the STAP algorithm we used can be found in [1, 2].

A. Doppler Filter Processing

The input to the Doppler filter processing task is one CPI complex data cube received from a phased-array radar. The computation in this task involves performing range correction for each range cell and the application of a windowing function (e.g. Hanning or Hamming) followed by an N -point FFT for every range cell and channel. The output of the Doppler filter processing task is a 3-dimensional complex data cube of size $K \times 2J \times N$ which is

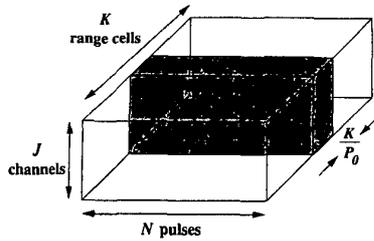


Fig. 5. Partitioning strategy for Doppler filter processing task. CPI data cube is partitioned among P_0 processors across dimension K .

referred to as staggered CPI data. In Fig. 4, we can see that this output is sent to the weight computation task as well as to the beamforming task.

Both the weight computation and the beamforming tasks are divided into easy and hard parts. These two parts use different portions of staggered CPI data and the associated amounts of computation are also different. The easy weight computation task uses range samples only from the first half of the staggered CPI data while the hard weight computation task uses range samples from the entire staggered CPI data. On the other hand, easy and hard beamforming tasks use all range cells rather than some of them. Therefore, the size of data to be transferred to the weight computation tasks is different from the size of data to be sent to the beamforming tasks. In Fig. 4, thicker arrows connected from the Doppler filter processing task to the beamforming tasks indicate that the amount of data sent to the beamforming tasks is more than the amount of data sent to the weight tasks.

The basic parallelization technique employed in the Doppler filtering processing task is to partition the CPI data cube across the range cells, that is, if P_0 processors are allocated to this task, then each processor is responsible for K/P_0 range cells. The reason for partitioning the CPI data cube along dimension K is that it maintains an efficient accessing mechanism for contiguous memory space. A total of $K \cdot 2J$ N -point FFTs are performed and the best

performance is achieved when every N -point FFT accesses its N data sets from a contiguous memory space. Fig. 5 illustrates the parallelization of this step. The intertask communication from the Doppler filter processing task to weight computation tasks is explained in Fig. 6(b). Since only subsets of range cells are needed in weight computation tasks, data collection has to be performed on the output data before passing it to the next tasks. Data collection is performed to avoid sending redundant data and hence reduces the communication costs.

B. Weight Computation

The second step in this pipeline is the computation of weights that will be applied to the next CPI. This computation for N pulses is divided into two parts, namely, easy and hard Doppler bins, as shown in Fig. 6(a). The hard Doppler bins (pulses), N_{hard} , are those in which significant ground clutter is expected. The remaining bins are easy Doppler bins, N_{easy} . The main difference between the two is the amount of data used and the amount of computation required. Not all range cells in the staggered CPI are used in weight calculation and different subsets of range samples are used in easy Doppler bins and hard Doppler bins.

To gather range samples for easy Doppler bins to calculate the weight vectors for the current CPI, data is drawn from three preceding CPIs by evenly spacing out over the first one third of K range cells of each of the three CPIs. The easy weight computation task involves N_{easy} QR factorizations, block updates, and back substitutions. In the easy weight calculation, only range samples in the first half of the staggered CPI data are used while hard weight computation employs range samples from the entire staggered CPI. Furthermore, the range extent for hard Doppler bins is split into six independent segments to further improve clutter cancellation. To calculate weight vectors for the current CPI, the range samples used in hard Doppler bins are taken from the immediately preceding staggered CPI combined with older, exponentially

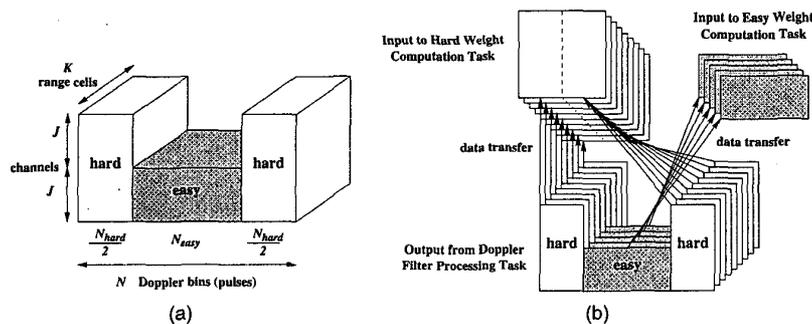


Fig. 6. (a) Staggered CPI data partitioned into easy and hard weight computation tasks. (b) Parallel intertask communication from Doppler filter processing task to easy and hard weight computation tasks requires different sets of range samples. Data collection needs to be performed before communication. This can be viewed as irregular data redistribution.

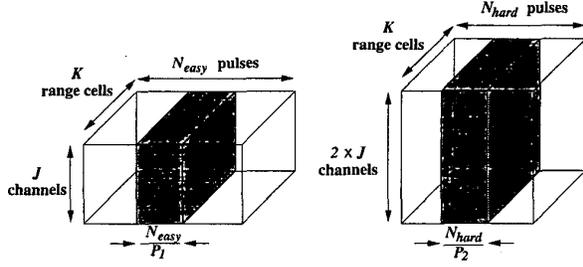


Fig. 7. Partitioning strategy for easy and hard weight computation tasks. Data cube is partitioned across dimension N .

forgotten, data from CPIs in the same direction. This is done for each of the six range segments. The hard weight computation task involves $6N_{\text{hard}}$ recursive QR updates, block updates, and back substitutions. The easy and hard weight computation tasks process sets of 2-dimensional matrices of different sizes.

Temporal data dependency exists in the weight computation task because both easy and hard Doppler bins use data from previous CPIs to compute the weights for the current CPI. The outputs of this step, the weight vectors, are two 3-dimensional complex data cubes of size $N_{\text{easy}} \times J \times M$ and $N_{\text{hard}} \times 2J \times M$ for the easy and hard weight computation tasks, respectively, where M is the number of receive beams. These two weight vectors are to be applied to the current CPI in the beamforming task. Because of the different sizes of easy and hard weight vectors, the beamforming task is also divided into easy and hard parts to handle different amounts of computation.

Given the uneven nature of the weight computations, different sets of processors are allocated to the easy and hard tasks. In Fig. 4, P_1 processors are allocated to easy weight computation and P_2 processors to hard weight computation. Since weight vectors are computed for each pulse (Doppler bin), the parallelization in this step involves partitioning of the data along dimension N , that is, each processor in easy weight computation task is responsible for N_{easy}/P_1 pulses while each processor in hard weight computation task is responsible for N_{hard}/P_2 pulses, as shown in Fig. 7.

Notice that the Doppler filter processing and weight computation tasks employ different data partitioning strategies (along different dimensions.) Due to different partitioning strategies, an all-to-all personalized communication scheme is required for data redistribution from the Doppler filter processing task to the weight computation task. That is, each of the P_1 and P_2 processors needs to communicate with all P_0 processors allocated to the Doppler filter processing task to receive CPI data. Since only subsets of the output of the Doppler filter processing task are used in the weight computation task, data collection is performed

before intertask communication. Although data collection reduces intertask communication cost, it also involves data copying from noncontiguous memory space to contiguous buffers. Sometimes the cost of data collection may become extremely large due to hardware limitations (e.g., high cache miss ratio.) When sending data to the beamforming task, the weight vectors have already been partitioned along dimension N which is the same as the data partitioning strategy for the beamforming task. Therefore, no data collection is needed when transferring data to the beamforming task.

C. Beamforming

The third step in this pipeline (which is actually the second step for the current CPI because the result of the weight task is only used in the subsequent time step) is beamforming. The inputs of this task are received from both the Doppler filter processing and weight computation tasks, as shown in Fig. 4. The easy weight vector received from the easy weight computation task is applied to the easy Doppler bins of the received CPI data while the hard weight vector is applied to the hard Doppler bins. The application of weights to CPI data requires matrix-matrix multiplications on two received data sets. Due to different matrix sizes for the multiplications in the easy and hard beamforming tasks, uneven computational load results. The beamforming task is also divided into easy and hard parts for parallelization purposes. This is because the easy and hard beamforming tasks require different amounts and portions of CPI data, and involve different computational loads. The inputs for the easy beamforming task are two 3-dimensional complex data cubes. One data cube, which is received from the easy weight computation task, is of size $N_{\text{easy}} \times M \times J$. The other is from the Doppler filter processing task and its size is $N_{\text{easy}} \times J \times K$. A total of N_{easy} matrix-matrix multiplications are performed where each multiplication involves two matrices of size $M \times J$ and $J \times K$, respectively. The hard beamforming task also has two input data cubes which are received from the Doppler filter processing and hard weight computation tasks. The data cube of size $6N_{\text{hard}} \times M \times 2J$ is received from the hard weight computation task and the Doppler filtered CPI data cube is of size $N_{\text{hard}} \times 2J \times K$. Since range cells are divided into 6 range segments, there are a total of $6N_{\text{hard}}$ matrix-matrix multiplications in hard beamforming. The results of the beamforming task are two 3-dimensional complex data cubes of size $N_{\text{easy}} \times M \times K$ and $N_{\text{hard}} \times M \times K$ corresponding to the easy and hard parts, respectively.

In a manner similar to the weight computation task, parallelization in this step also involves partitioning of data across the N dimension (Doppler

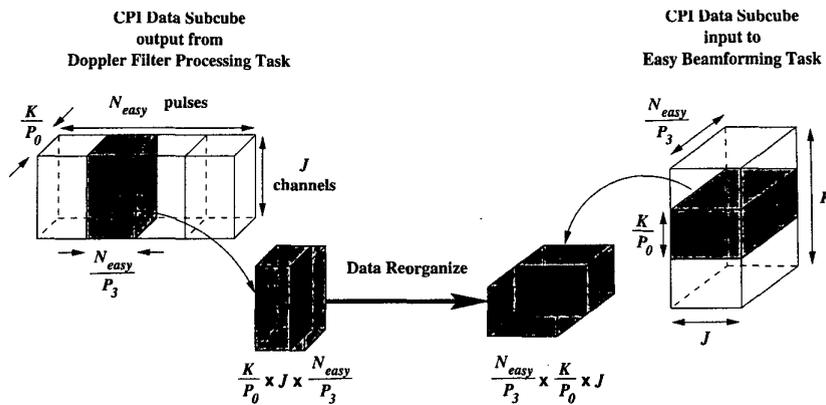


Fig. 8. Data redistribution from Doppler filter processing task to easy beamforming task. CPI data subcube of size $(K/P_0) \times J \times N_{\text{easy}}/P_3$ is reorganized to subcube of size $(N_{\text{easy}}/P_3) \times (K/P_0) \times J$ before sending from one processor in Doppler filtering processing task to another in easy beamforming task.

bins.) Different sets of processors are allocated to the easy and hard beamforming tasks. Since the cost of matrix multiplications can be determined accurately, the computations are equally divided among the allocated processors for this task. As seen from Fig. 4, this task requires data to be communicated from the first as well as the second task. Because data is partitioned along different dimensions, an all-to-all personalized communication is required for data redistribution between the Doppler filter processing and beamforming tasks. The output of the Doppler filter processing task is a data cube of size $K \times 2J \times N$ which is redistributed to the beamforming task after data reorganization in the order of $N \times K \times 2J$. Data reorganization has to be done before the intertask communication between the two tasks takes place, as shown in Fig. 8.

Data reorganization involves data copying from noncontiguous memory space and its cost may become extremely large due to cache misses. For example, two Doppler bins in the same range cell and the same channel are stored in contiguous memory space. After data reorganization, they are $K/P_0 \cdot J$ element distance apart. Therefore, if P_0 is small and the size of the CPI data subcube partitioned in each processor is large, then it is quite likely that expensive data reorganization will be needed which becomes a major part of the communication overhead. The algorithms which perform data collection and reorganization are crucial to exploit the available parallelism. Note that receiving data from the weight computation tasks does not involve data reorganization or data collection because they have the same partitioning strategy (along dimension N .)

D. Pulse Compression

The input to the pulse compression task is a 3-dimensional complex data cube of size $N \times M \times K$,

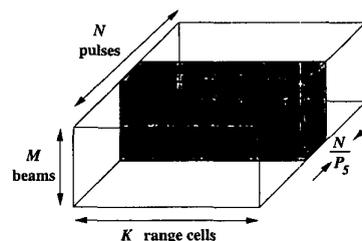


Fig. 9. Partitioning strategy for pulse compression task. Data cube is partitioned across dimension N into P_5 processors.

as shown in Fig. 9. This data cube consists of two subcubes of size $N_{\text{easy}} \times M \times K$ and $N_{\text{hard}} \times M \times K$ which are received from the easy and hard beamforming tasks, respectively. Pulse compression involves convolution of the received signal with a replica of the transmit pulse waveform. This is accomplished by first performing K -point FFTs on the two inputs, point-wise multiplication of the intermediate result, and then computing the inverse FFT. The output of this step is a 3-dimensional real data cube of size $N \times M \times K$. The parallelization of this step is straightforward and involves the partitioning of the data cube across the N dimension. Each of the FFTs could be performed on an individual processor and, hence, each processor in this task gets an equal amount of computation. Partitioning along the N dimension also results in an efficient accessing mechanism for contiguous memory space when running FFTs. Since both the beamforming and pulse compression tasks use the same data partitioning strategy (along dimension N), no data collection or reorganization is needed prior to communication between these two tasks. After pulse compression, the square of the magnitude of the complex data is computed to move to the real power domain. This cuts the data set size in half and eliminates the computation of the square root.

E. CFAR Processing

The input to this task is an $N \times M \times K$ real data cube received from the pulse compression task. The sliding window constant false alarm rate (CFAR) processing compares the value of a test cell at a given range to the average of a set of reference cells around it times a probability of false alarm factor. This step involves summing up a number of range cells on each side of the cell under test, multiplying the sum by a constant, and comparing the product to the value of the cell under test. The output of this task, which appears at the pipeline output, is a list of targets at specified ranges, Doppler frequencies, and look directions. The parallelization strategy for this step is the same as for the pulse compression task. Both tasks partition the data cube along the N dimension. Also, no data collection or reorganization is needed in the pulse compression task before sending data to this task.

VI. SOFTWARE DEVELOPMENT AND SYSTEM PLATFORM

All the parallel program development and their integration was performed using ANSI C language and message passing interface (MPI) [24]. This permits easy portability across various platforms which support C language and MPI. Since MPI is becoming a de facto standard for high-performance systems, we believe the software is portable.

The implementation of the STAP application based on our parallel pipeline system model has been done on the Intel Paragon at the Air Force Research Laboratory, Rome, NY. This machine contains 321 compute nodes interconnected in a two-dimensional mesh. The Paragon runs Intel's standard Open Software Foundation (OSF) UNIX operating system. Each compute node consists of three i860 RISC processors which are connected by a system bus and share a 64M byte memory. The speed of an i860 RISC processor is 40 MHz and its peak performance is 100M floating point operations per second. The interconnection network has a message startup time of 35.3 μ s and a data transfer time of 6.53 ns/byte for point-to-point communication.

In our implementation, a double buffering strategy was used both in the receive and send phases. During the execution loops, this strategy employs two buffers alternatively such that one buffer can be processed during the communication phase while the other buffer is processed during the compute phase. Together with the double buffering implementation, asynchronous send and receive calls were employed in order to maximize the overlap of communication and computation. Asynchronous communication means that the program executing the send/receive does not wait until the send/receive is complete. This type of

```

n      : number of CPIs
inBuf[2] : input data buffer
outBuf[2] : output data buffer

1 for i ← 0 to n - 1
2   prev ← (i - 1) mod 2
3   cur ← i mod 2
4   next ← (i + 1) mod 2
5   t0 ← read timer
6   post async receives for inBuf[next]
7   wait for completion of previous receives for inBuf[cur]
8   data unpacking on inBuf[cur]
9   t1 ← read timer
10  computation on inBuf[cur] and result in outBuf[cur]
11  t2 ← read timer
12  data packing for outgoing message on outBuf[cur]
13  post async sends for outBuf[cur] to next task
14  wait for completion of sends for outBuf[prev]
15  t3 ← read timer

```

Fig. 10. Implementation of timing computation and communication for each task. Double buffering strategy is used to overlap communication with computation. Receive time = $t_1 - t_0$, compute time = $t_2 - t_1$, send time = $t_3 - t_2$.

communication is also referred to as non-blocking communication. The other option is synchronous communication which blocks the send/receive operation until the message has been sent/received. The general execution flow and the approach to measure the timing for each part of computation and communication is given in Fig. 10. We used MPI timer, MPI_Wtime(), because this function is portable with high resolution.

VII. PERFORMANCE RESULTS

We specified the parameters that were used in our experiments as follows:

range cells (K) = 512,
channels (J) = 16,
pulses (N) = 128,
receive beams (M) = 6,
easy Doppler bins (N_{easy}) = 72,
hard Doppler bins (N_{hard}) = 56.

Given these values of parameters, the total number of floating point operations (flops) required for each CPI data to be processed throughout this STAP algorithm is 403,552,528. Table I shows the number of flops required for each task. A total of 25 CPI complex data cubes were generated as inputs to the parallel pipeline system. Each task in the pipeline contains three major parts: receiving data from the previous task, main computation, and sending results to the next task. Performance results are measured separately for these three parts, namely receiving time, computation time, and sending time. In each task timing results for processing one CPI data were obtained by accumulating the execution time for the middle 20 CPIs and then averaging it. Timing results presented here do not include the effect of the initial setup (first 3 CPIs) and final iterations (last 2 CPIs).

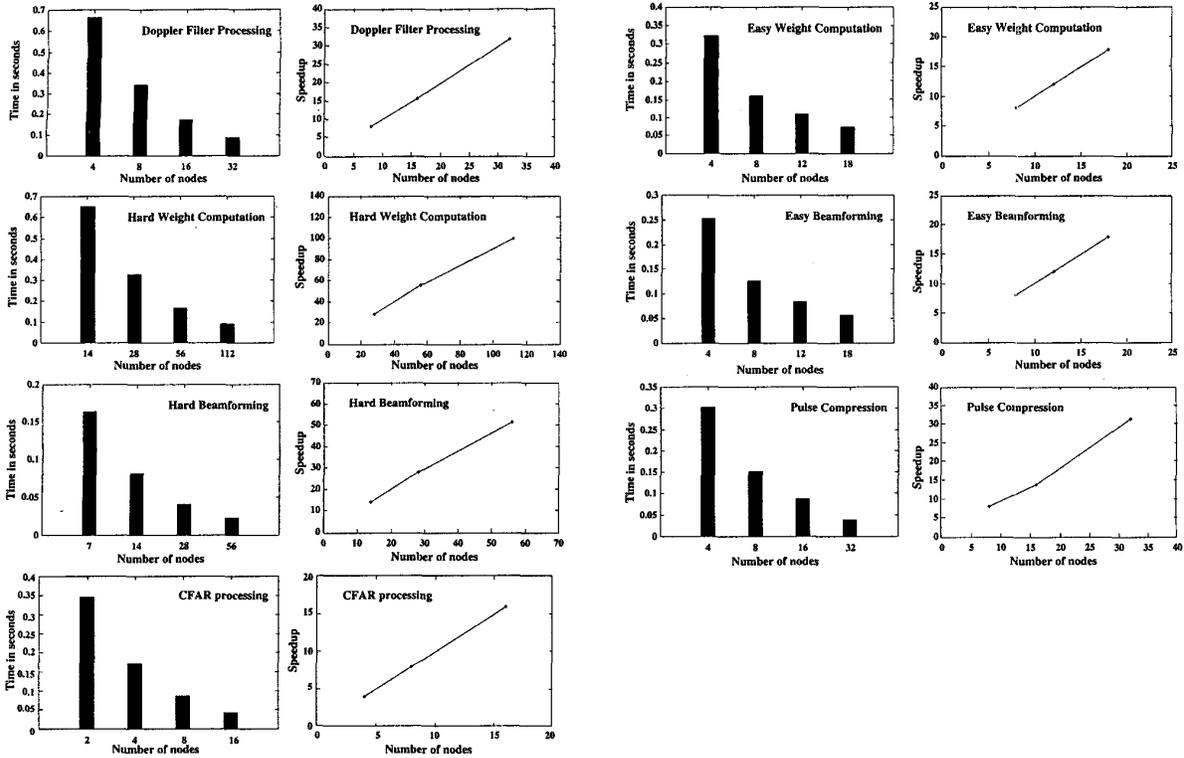


Fig. 11. Performance and speedup of computation time as function of number of compute nodes for all tasks.

TABLE I
Number of Floating Point Operations for PRI-Staggered Post-Doppler STAP Algorithm to Process One CPI Data

Task	number of floating point operations
Doppler filter processing	79,691,776
hard weight computation	197,038,464
easy weight computation	13,851,792
easy beamforming	28,311,552
hard beamforming	44,040,192
pulse compression	38,928,384
CFAR processing	1,690,368
Total	403,552,528

A. Computation Costs

The task of computing hard weights is the most computationally demanding task. The Doppler filter processing task is the second most demanding task. Naturally, more compute nodes are assigned to these two tasks in order to obtain a good performance. For each task in the STAP algorithm, parallelization was done by evenly dividing the computational load across the compute nodes assigned. Since there is no intratask data dependency, no inter-processor communication occurs within any single task in the pipeline. Another way to view this is that intratask communication is moved to the beginning of each task within the data redistribution step. Fig. 11 gives the

computation performance results as functions of the numbers of nodes and the corresponding speedup on the AFRL Intel Paragon. For each task, we obtained linear speedups.

B. Intertask Communication

Intertask communication refers to the communication between the sending and receiving (distinct and parallel) tasks. This communication cost depends on both the processor assignment for each task as well as on the volume and extent of data reorganization. Tables II–VI present the intertask communication timing results. Each table considers pairs of tasks where the number of compute nodes for both tasks are varied. In some cases timing results shown in the tables contain idle time for waiting for the corresponding task to complete. This happens when the receiving task's computation part of the receiving task completes before the sending task has generated data to send.

From most of the results (Tables II–VI) the following important observations can be made. First, when the number of nodes is unbalanced (e.g., sending task has a small number of nodes while the receiving task has a large number of nodes), the communication performance is not very good. Second, as the number of nodes is increased in the sending and receiving tasks, communication scales

TABLE II
Timing Results of Intertask Communication From Doppler Filter Processing Task to Its Successor Tasks

	# nodes	easy weight		hard weight				easy BF		hard BF	
		16		56		112		16		16	
		send	recv	send	recv	send	recv	send	recv	send	recv
Doppler filter	8	.1332	.4339	.1332	.3603	.1332	.4441	.1332	.4509	.1332	.4395
	16	.0679	.1780	.0679	.1048	.0679	.1837	.0679	.1955	.0679	.1843
	32	.0340	.0511	.0332	.0034	.0340	.0563	.0340	.0646	.0340	.0519

Note: Time in seconds.

TABLE III
Timing Results of Intertask Communication From Easy Weight Computation Task to Easy Beamforming Task

	# nodes	easy beamforming			
		8		16	
		send	recv	send	recv
easy weight	4	.0005	.1956	.0007	.2570
	8	.0088	.0883	.0004	.0905
	16	.0768	.0807	.0003	.0660

Note: Time in seconds.

TABLE VI
Timing Results of Intertask Communication From Pulse Compression Task to CFAR Processing Task

	# nodes	CFAR processing			
		4		8	
		send	recv	send	recv
pulse compr	4	.0099	.3351	.0098	.3348
	8	.0053	.0662	.0051	.1750
	16	.1256	.0435	.0028	.1783

Note: Time in seconds.

TABLE IV
Timing Results of Intertask Communication From Hard Weight Computation Task to Hard Beamforming Task

	# nodes	hard beamforming			
		8		16	
		send	recv	send	recv
hard weight	28	.0007	.1798	.0007	.2485
	56	.0100	.1468	.0065	.0765
	112	.1824	.1398	.0005	.0543

Note: Time in seconds.

TABLE V
Timing Results of Intertask Communication From Easy and Hard Beamforming Tasks to Pulse Compression Task

	# nodes	pulse compression			
		8		16	
		send	recv	send	recv
easy BF	4	.0069	.5016	.0069	.5714
	8	.0036	.1379	.0036	.2090
	16	.0580	.0771	.0022	.0569
hard BF	4	.0054	.5016	.0054	.5714
	8	.0029	.1379	.0030	.2090
	16	.1159	.0771	.0017	.0569

Note: Time in seconds.

tremendously. This happens for two reasons. One, each node has less data to reorganize, pack and send and each node has less data to receive; and two, contention at the sending and receiving nodes is reduced. For example, Table II shows that when the number of nodes of the sending task is increased from 8 to 32, the communication times improve in a superlinear fashion. Thus, it is not sufficient

to improve the computation times for such parallel pipelined applications to improve throughput and latency.

In Fig. 10 the receiving time for each loop is given by subtracting t_1 from t_0 . Since computation has to be performed only after the input data has been received, receiving time may contain the waiting time for the input, shown in line 4. Sending time, $t_3 - t_2$, measures the time containing data packing (collection and reorganization) and posting sending requests. Because of the asynchronous send used in the implementation, the results shown here are the visible sending time and the actual sending action may occur in other portions of the task. Similar to the receiving time, the sending time may also contain the waiting time for the completion of sending requests in the previous loop, shown in line 8. Especially in the cases when two communicating tasks have an uneven partitioned parallel computation load, this effect becomes more apparent. With a large number of nodes, there is tremendous scaling in performance of communicating data as the number of nodes is increased. This is because the amount of processing for communication per node is decreased (as it handles less amount of data), the amount of data per node to be communicated is decreased and the traffic on links going in and out of each node is reduced. This model scales well for both computation and communication.

C. Integrated System Performance

Integrated system refers to the evaluation of performance when all the tasks are considered together. Throughput and latency are the two most

TABLE VII
Performance Results for 3 Cases With Different Node Assignments

case 1: total number of nodes = 236

	# nodes	rcv	comp	send	total
Doppler filter	32	.0055	.0874	.0348	.1276
easy weight	16	.0493	.0913	.0003	.1408
hard weight	112	.0555	.0831	.0005	.1390
easy BF	16	.0658	.0708	.0021	.1387
hard BF	28	.0936	.0414	.0010	.1361
pulse compr	16	.0551	.0776	.0028	.1355
CFAR	16	.0910	.0434	-	.1344
throughput					7.2659
latency					0.3622

case 2: total number of nodes = 118

	# nodes	rcv	comp	send	total
Doppler filter	16	.0110	.1714	.0668	.2492
easy weight	8	.0998	.1636	.0003	.2637
hard weight	56	.0979	.1636	.0005	.2621
easy BF	8	.1302	.1267	.0036	.2605
hard BF	14	.1782	.0822	.0017	.2622
pulse compr	8	.1027	.1543	.0051	.2621
CFAR	8	.1742	.0864	-	.2606
throughput					3.7959
latency					0.6805

case 3: total number of nodes = 59

	# nodes	rcv	comp	send	total
Doppler filter	8	.0219	.3509	.1296	.5024
easy weight	4	.1796	.3254	.0003	.5053
hard weight	28	.1779	.3265	.0006	.5050
easy BF	4	.2439	.2529	.0068	.5037
hard BF	7	.3370	.1636	.0032	.5039
pulse compr	4	.1806	.3067	.0097	.4970
CFAR	4	.3240	.1723	-	.4963
throughput					1.9898
latency					1.3530

Note: Time in seconds.

important measures for performance evaluation in addition to individual task computation time and intertask communication time. Table VII gives timing results for three different cases with different node assignments.

In Section V, (1) and (2) provide the throughput and latency for one CPI data set. The measured throughput is obtained by placing a timer at the end of the last task and recording the time difference between every loop (that is between two successive completions of the pipeline.) The inverse of this measure provides the throughput. On the other hand, it is more difficult to measure latency because it requires synchronizing clocks at the node of the first

TABLE VIII
Throughput and Latency for the 3 Cases of Table VII

# of nodes		236	118	59
throughput	equation	7.1019	3.7919	1.9791
	real	7.2659	3.7959	1.9898
latency	equation	0.5362	1.0346	1.9996
	real	0.3622	0.6805	1.3530

Note: Real results obtained from experiments while equation results obtained from applying individual tasks' timing to equations (1) and (2). Unit of throughput is number of CPIs per second. Unit of latency is second.

and last task. Thus, to obtain the measured latency, the timing measurement should be made by first reading time at both the first task and last task when the first task is ready to read a new input data. This can be done by sending a signal from the first task to the last task when the first task is ready for reading the new input data. Then the timer for the last task can be started.

In fact, the latency given in (2) represents an **upper bound** because the way we time tasks contains the time of waiting for input from the previous task. This waiting time portion overlaps with the computation time in the previous tasks and should be excluded from the latency. Thus, the latency results are conservative values and the real latency is expected to be smaller than this value. However, the latency given from (2) indicates the worst case performance for our implementation. The real latency equation, therefore, becomes

$$\text{real latency} = T_0 + \max(T'_3, T'_4) + T'_5 + T'_6 \quad (3)$$

where $T'_i = T_i$ -idle time at receiving, $i = 3, 4, 5$, and 6 .

Table VIII gives the throughput and latency results for the 3 cases shown in Table VII. From these 3 cases, it is clear that even for the latency and throughput measures we obtain linear speedups from our experiments. Given that this scale-up is up to compute 236 nodes (we were limited to these number of nodes due to the size of the machine), we believe these are very good results.

As discussed in Section IV, tradeoffs exist between assigning nodes to maximize throughput and to minimize latency, given limited resources. Using two examples, we illustrate how further performance improvements may (or may not) be achieved if few extra nodes are available. We now take case 2 from Table VII as an example and add some extra nodes to tasks to analyze its affect to the throughput and latency. Suppose that case 2 has fulfilled the minimum throughput requirement and more nodes can be added. Table IX shows that adding 4 more nodes to the Doppler filter processing task not only increases the throughput but also reduces the latency. This is because the communication amount for each send and

TABLE IX
Performance Results for Adding 4 More Nodes to Doppler Filter Processing Task to Case 2 in Table VII

total number of nodes = 122

	# nodes	recv	comp	send	total
Doppler filter	20	.0090	.1395	.0540	.2024
easy weight	8	.0519	.1633	.0003	.2155
hard weight	56	.0486	.1644	.0005	.2135
easy BF	8	.0815	.1272	.0037	.2124
hard BF	14	.1232	.0823	.0018	.2073
pulse compr	8	.0519	.1543	.0051	.2113
CFAR	8	.1240	.0864	-	.2105
throughput					5.0213
latency					0.5498

Note: Time in seconds.

receive between the Doppler filter processing task to weight computation and to beamforming tasks is reduced (Table IX). So, clearly, adding nodes to one task not only affects the performance of that task but has a measurable effect on the performance of other tasks. By increasing the number of nodes 3%, the improvement in throughput is 32% and in latency is 19%. *Such effects are very difficult to capture in purely theoretical models because of the secondary effects.*

Since the parallel computation load may be different among tasks, bottleneck problems arise when some tasks in the pipeline do not have the proper numbers of nodes assigned. If the number of nodes assigned to one task with a heavy work load is not enough to catch up the input data rate, this task becomes a bottleneck in the pipeline system. Hence, it is important to maintain approximately the same computation time among tasks in the pipeline system to maximize the throughput and, also, achieve higher processor utilization. One bottleneck task can be seen when its computation time is relatively much larger than the rest of the tasks. The performance of the entire system degrades because the rest of the tasks have to wait for the completion of the bottleneck task to send/receive data to/from it no matter how many more nodes assigned to them and how fast they can complete their jobs. Therefore, poor task scheduling and processor assignment will cause a significant portion of idle time in the resulted communication costs. In Table X we added a total of 16 more nodes to the pulse compression and CFAR processing tasks to the case in Table IX. Comparing to case 2 in Table VII, we can see that the throughput increased. However, the throughput did not improve compared with the results in Table IX, even though this assignment has 16 more nodes. In this case, the weight tasks are the bottleneck tasks because their computation costs are relatively higher than other tasks. We can see that the receiving time of the rest of the tasks are much larger than their computation

TABLE X
Performance Results for Adding 16 More Nodes to Pulse Compression and CFAR Processing Tasks to the Case in Table IX

total number of nodes = 138

	# nodes	recv	comp	send	total
Doppler filter	20	.0091	.1395	.0541	.2027
easy weight	8	.0516	.1633	.0003	.2152
hard weight	56	.0488	.1644	.0005	.2137
easy BF	8	.0819	.1273	.0037	.2129
hard BF	14	.1301	.0823	.0018	.2142
pulse compr	16	.1337	.0775	.0028	.2140
CFAR	16	.1701	.0434	-	.2135
throughput					4.9052
latency					0.4247

Note: Time in seconds.

time. A significant portion of idle time waiting for the completion of weight tasks is in the receiving time. On the other hand, we observe 23% improvement in the latency. This is because the computation time is reduced in the last two tasks with more nodes assigned. From (3), the execution time of these two tasks, T'_5 and T'_6 , decreases and, therefore, the latency is reduced.

VIII. CONCLUSIONS

In this paper we presented performance results for a PRI-staggered post-Doppler STAP algorithm implementation on the Intel Paragon machine at the Air Force Research Laboratory, Rome, NY. The results indicate that our approach of parallel pipelined implementation scales well both in terms of communication and computation. For the integrated pipeline system, the throughput and latency also demonstrate the linear scalability of our design. Linear speedups were obtained for up to 236 compute nodes. When more than 236 nodes are used, the speedup curves for the results of throughput and latency may saturate. This is because the communication costs will become significant with respect to the computation costs.

Almost all radar applications have real-time constraints. Hence, a well-designed system should be able to handle any changes in the requirements on the response time by dynamically allocating or reallocating processors among tasks. Our design and implementation not only shows tradeoffs in parallelization, processor assignment, and various overheads in inter and intratask communication etc., but it also shows that accurate performance measurement of these systems is very important. Consideration of issues such as cache performance when data is packed and unpacked, and impact of the parallelization and processor assignment for one

$$M = \begin{bmatrix} \text{data vector 1} \\ \text{data vector 2} \\ \vdots \\ \text{data vector } n \\ w_s^H \end{bmatrix}$$

w = column vector of element weights
 w_s = steering vector (column)
 Find w = least square error solution of: $Mw = [0 \ 0 \ \dots \ 0 \ 1]^T$

Fig. 12. Conventional least squares processing.

task on another task are crucial. This is normally not easily captured in theoretical models. In the future we plan to incorporate further optimizations including multithreading, multiple pipelines, and multiple processors on each compute node.

APPENDIX A. SPACE-TIME ADAPTIVE PROCESSING WITH MAINBEAM CONSTRAINT

The STAP problem can be formulated as a least squares minimization of the clutter response. This approach is desirable from a computational standpoint, as it is not necessary to produce an estimate of the clutter covariance matrix, which is an order n^3 operation. In the least squares approach, a matrix M is constructed from snapshots of the array data after Doppler processing, and a weight vector w is computed which minimizes the norm of the product vector Mw . The snapshots are samples of data from each array element taken at range cells adjacent to the test cell, and also from multiple CPIs which are decorrelated across time. Typically a beam constraint, such as a requirement for unit response in the direction of the desired target, is added to rule out the trivial solution, $w = 0$. As illustrated in Fig. 12, the weight vector is computed by multiplying the pseudoinverse of M times a unit vector.

While assuring a non-zero solution for the weights, the conventional beam constraint placed on the least squares problem as formulated above often produces an adapted pattern with a highly distorted main beam with a peak response far removed from the target of interest. The algorithm that was formulated and implemented here is a constrained version of the least squares problem. Given a steering vector w_s , we seek a weight vector w that minimizes the clutter response while maintaining a close similarity between w and w_s . This condition is specified by augmenting the data matrix M with an identity matrix as depicted in Fig. 13. The product of the identity matrix and the solution vector w is set to a scalar multiple of the steering vector w_s . The least squares solution is a compromise between clutter rejection

$$M = \begin{bmatrix} \text{data vector 1} \\ \text{data vector 2} \\ \vdots \\ \text{data vector } n \\ \text{Identity Matrix} * k \end{bmatrix}$$

w = column vector of element weights
 w_s = steering vector (column)
 k = beam constraint weight
 Find w = least square error solution of: $Mw = [0 \ 0 \ \dots \ 0 \ w_s^H]^H$

Fig. 13. Beam constrained least squares processing.

and preservation of mainbeam shape. In practice, only slight modifications of the weight vector are required to move spatial nulls into the clutter region, for clutter returns that are outside of the mainbeam. Thus, preservation of mainbeam shape requires only a slight reduction of clutter rejection performance, and is often offset by an increase in array gain on the desired target. As shown in Fig. 13, the preservation of main beam shape is controlled by scalar k . The choice of k directs the least squares solution for w to adhere more closely to the steering vector when k is large, and emphasize clutter cancellation at the expense of beam shape when k is small. Since k is variable depending on operating requirements, we normalize the resulting weight vector to unit length.

There is a computational advantage of the constrained technique of Fig. 13 over that of Fig. 12 for systems that utilize multiple beam steering. Since the steering vector w_s appears only on the right side of the equation, and matrix M is independent of the mainbeam pointing angle, the QR factorization of M needs be performed only once for a given data set. Multiple weight vectors can be computed for different steering vector choices by multiplying the same matrix pseudoinverse or QR factorization by several choices of constraint vectors.

APPENDIX B. MATLAB VERSION OF RT-MCARM PROCESSING ALGORITHM

```

function [detections] = process_CPI(CPI_data, N)
    % N is the CPI number

    num_channels      = 16;
    num_range         = 512;
    num_pulses        = 128;
    num_doppler       = num_pulses;
    numHardDop        = 56;
    stagger            = 3;    % PRI-stagger pulses
    BeamConstraintWt  = 0.5;
    FreqConstraintWt  = 0.5;
    DopplerWindow     = hanning(num_pulses-stagger);
    range_Segment_Boundaries = [0 75 150 225 300 375 512];
    
```

```

% Doppler Filter Processing
doppler_data = rawToFFT(CPI_data,DopplerWindow,stagger);

% Easy Weight Computation and Beamforming
beamformed_data(numHardDop/2+1:num_doppler-numHardDop/2) = easy_wts.' * doppler_data;
easy_wts = zeros(num_doppler,num_channels,num_beams);
for idop = numHardDop/2+1 : num_doppler - numHardDop/2,
    [easy_wts(idop,,:), previous_doppler_data(idop,,:)] = computeEasyWts(idop,BeamConstraintWt, ...
        Steering_vectors,previous_doppler_data(idop,,:),doppler_data(idop,,:));
end;

% Hard Weight Computation and Beamforming
for rangeSeg = 1:num_range_segments,
    startR = range_Segment_Boundaries(rangeSeg)+1;
    endR = range_Segment_Boundaries(rangeSeg+1);

    beamformed_data(1:numHardDop/2,.,startR:endR) = hard_wts(rangeSeg,1:numHardDop/2,.,:).' * ...
        doppler_data(1:numHardDop/2,.,startR:endR);
    beamformed_data(num_doppler-numHardDop/2+1:num_doppler,.,startR:endR) = ...
        hard_wts(rangeSeg,num_doppler-numHardDop/2+1:num_doppler,.,:).' * ...
        doppler_data(num_doppler-numHardDop/2+1:num_doppler,.,startR:endR);

    hard_wts = zeros(num_range_segments,num_doppler,num_channels,num_beams);
    for idop = 1:numHardDop/2,
        [wts(rangeSeg,idop,,:), new_r(idop,,:)] = computeRecurHardWts(idop,startR,endR, ...
            FreqConstraintWt,BeamConstraintWt,Steering_Vectors,doppler_data(idop,,:), ...
            new_r(idop,,:),stagger);
    end; % idop
    for idop = num_doppler-numHardDop/2+1:num_doppler,
        [wts(rangeSeg,idop,,:), new_r(idop,,:)] = computeRecurHardWts(idop,startR,endR, ...
            FreqConstraintWt,BeamConstraintWt,Steering_Vectors,doppler_data(idop,,:), ...
            new_r(idop,,:),stagger);
    end; % idop
end; % range_segments

% Pulse Compression and CFAR processing
pulsecompression= pulseCompression(beamformed,pc_filter_freq);
detections      = CFAR(pulsecompression);
end; % function process_CPI

function [Doppler_data] = rawToFFT(CPI_data>window,stagger)
% input: CPI_data(num_pulses,num_range,num_channels)
% output Doppler_data(num_doppler,num_channels,num_range)

Doppler_data = zeros(num_doppler,num_channel,num_range);

padded_CPI_data(1:num_pulses-stagger,,:) = CPI_data(1:num_pulses-stagger,,:) .* window);
padded_CPI_data(num_pulses-stagger+1:num_pulses,,:) = zeros(stagger,num_channels,num_range);
Doppler_data(1:num_doppler,,:) = fft_pulses(padded_CPI_data);

padded_CPI_data(stagger+1:num_pulses,,:) = CPI_data(1:num_pulses-stagger,,:) .* window);
padded_CPI_data(1:stagger:num_pulses,,:) = zeros(stagger,num_channels,num_range);
Doppler_data(num_doppler+1:2*num_doppler,,:) = fft_pulses(padded_CPI_data);
end; % function rawToFFT

function [wts, updated_doppler_data] = computeEasyWts(doppler,diagWts, Steering_vectors, ...
    prev_doppler_data,new_doppler_data)
% computes adaptive weights directly for the easy doppler bins from data from three previous CPIs

```

```

% shift data from previous two CPIs N-1 and N-2 up, overwriting data from CPI N-3
updated_doppler_data(1:Total_easy_Samples * 2/3,:) = ...
    prev_doppler_data(Total_easy_Samples * 1/3:Total_easy_Samples,:);

updated_doppler_data(Total_easy_Samples * 2/3,Total_easy_Samples,:) = ...
    Select_Range_Samples(Total_easy_Samples * 1/3,new_doppler_data);

avg = average(updated_doppler_data) * diagWts ;

for beam=1:num_beams,
    work = updated_doppler_data;
    work(updated_doppler_data+1:updated_doppler_data+num_channels,:) = avg * eye(num_channels);

    rhs = zeros(updated_doppler_data+num_channels,1);
    rhs(updated_doppler_data+1:updated_doppler_data+num_channels,1) = Steering_vectors(:,beam);

    wts(:,beam) = work\rhs;
    wts(:,beam) = wts(:,beam)/(sqrt(wts(:,beam)' * wts(:,beam)));
end;
end; % function computeEasyWts

function [wts, new_r] = computeRecurHardWts(doppler,startRangeSeg,endRangeSeg,spatialWt,freqWt, ...
    Steering_vectors,doppler_data,old_r,stagger, CPI_Num)
% computes adaptive weights recursively for the hard doppler bins.

forgettingFactor = 0.6;
qr_x = zeros(2*num_channels + num_hard_samples,2*num_channels);
qr_x(1:2*num_channels,:) = forgettingFactor * old_r;
qr_x(2*num_channels+1:2*num_channels+num_hard_samples,:) = ...
    Select_Range_Samples(num_hard_samples,doppler_data);
avg = average(qr_x);
half_channels = num_channels/2;
if (CPI_Num mod 2 = 1) then colOffset = 0;
else colOffset = half_channels;
end;
% constrain half of the columns
qr_x(num_hard_samples + 2*num_channels+1,1+colOffset:half_channels+colOffset) = ...
    [avg * eye(half_channels)]; % spatial constraints
qr_x(num_hard_samples + 2*num_channels+1,
    num_channels+1+colOffset:num_channels+half_channels+colOffset) = ...
    [avg * eye(half_channels) * exp(-j * 2 * pi * (doppler-1) * stagger / num_doppler)];
[q new_r] = qr(qr_x(1:num_hard_samples + 2*num_channels + half_channels,:),0);

for beam=1:num_beams,
    work = new_r;
    % freq constraints scaled by e(-jk/n)
    work(2*num_channels+1:3*num_channels,1:num_channels) = [avg * eye(num_channels)];

    rhs = zeros(3*num_channels,1);
    rhs(2*num_channels+1:3*num_channels,1) = Steering_vectors(:,beam);

    [q2 r2] = qr([work rhs],0);
    matrhs(:,beam) = r2(1:2*num_channels,2*num_channels+1);

    wts(:,beam) = work\rhs;
    wts(:,beam) = wts(:,beam)/(sqrt(wts(:,beam)' * wts(:,beam)));
end;
end; % function computeRecurHardWts

```

ACKNOWLEDGMENTS

We acknowledge the use of the Intel Paragon at Caltech for the initial development.

REFERENCES

- [1] Linderman, M., and Linderman, R. (1998)
Real-time STAP demonstration on an embedded high performance computer.
IEEE AES Systems Magazine (Mar. 1998), 15–21.
- [2] Brown, R., and Linderman, R. (1997)
Algorithm development for an airborne real-time STAP demonstration.
In *Proceedings of the IEEE National Radar Conference*, 1997.
- [3] Little, M., and Berry, W. (1997)
Real-time multi-channel airborne radar measurements.
In *Proceedings of the IEEE National Radar Conference*, 1997.
- [4] Kumar, V., Grama, A., Gupta, A., and Karypis, G. (1994)
Introduction to Parallel Computing: Design and Analysis of Algorithms.
Benjamin-Cummings, 1994.
- [5] Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., and Walker, D. (1990)
Solving Problems on Concurrent Processors.
Englewood Cliffs, NJ: Prentice-Hall, 1990.
- [6] Hwang, K. (1993)
Advanced Computer Architecture: Parallelism, Scalability, Programmability.
New York: McGraw-Hill, 1993.
- [7] Golub, G., and Ortega, J. (1993)
Scientific Computing: An Introduction with Parallel Computing.
Boston: Academic Press, 1993.
- [8] Xavier, C., and Iyengar, S. (1998)
Introduction to Parallel Algorithms.
New York: Wiley, 1998.
- [9] Lebak, J., Durie, R., and Bojanczyk, A. (1996)
Toward a portable parallel library for space-time adaptive methods.
Technical report CTC96TR242, Cornell Theory Center, June 1996.
- [10] Olszanskyj, S., Lebak, J., and Bojanczyk, A. (1995)
Parallel algorithms for space-time adaptive processing.
In *Proceedings of the International Parallel Processing Symposium* (Apr. 1995), 77–81.
- [11] Lim, Y., and Prasanna, V. (1996)
Scalable portable implementations of space-time adaptive processing.
In *Proceedings of the 10th International Conference on High Performance Computing* (June 1996).
- [12] Bhat, P., Lim, Y., and Prasanna, V. (1995)
Issues in using heterogeneous HPC systems for embedded real time signal processing applications.
In *Proceedings of the 2nd International Workshop on Real-Time Computing Systems and Applications* (Oct. 1995).
- [13] Lee, M., and Prasanna, V. (1997)
High throughput-rate parallel algorithms for space time adaptive processing.
Presented at the 2nd International Workshop on Embedded Systems and Applications, Apr. 1997.
- [14] Martinez, D. (1999)
Application of parallel processors to real-time sensor array processing.
Presented at the International Parallel Processing Symposium, Apr. 1999.
- [15] Cain, K., Torres, J., and Williams, R. (1997)
RT-STAP: Real-time space-time adaptive processing benchmark.
Technical report 96B0000021, MITRE Corporation, Feb. 1997.
- [16] Brown, C., Flanzbaum, M., Games, R., and Ramsdell, J. (1994)
Real-time embedded high performance computing: Application benchmarks.
Technical report MTR94B145, MITRE Corporation, Oct. 1994.
- [17] Choudhary, A., and Patel, J. (1990)
Parallel Architectures and Parallel Algorithms for Integrated Vision Systems.
Boston: Kluwer Academic Publishers, 1990.
- [18] Choudhary, A., and Ponnusamy, R. (1992)
Run-time data decomposition for parallel implementation of image processing and computer vision tasks.
Journal of Concurrency, Practice and Experience, 4, 4 (June 1992), 313–334.
- [19] Choudhary, A., and Ponnusamy, R. (1992)
Parallel implementation and evaluation of a motion estimation system algorithm using several data decomposition strategies.
Journal of Parallel and Distributed Computing, 14 (Jan. 1992), 50–65.
- [20] Thakur, R., Choudhary, A., and Ramanujam, J. (1996)
Efficient algorithms for array redistribution.
IEEE Transactions on Parallel and Distributed Systems, 6, 7 (June 1996), 587–594.
- [21] Berger, M., and Bokhari, S. (1987)
A partitioning strategy for nonuniform problems on multiprocessors.
IEEE Transactions on Computers, C-36, 5 (May 1987), 570–580.
- [22] Berman, F., and Snyder, L. (1987)
On mapping parallel algorithms into parallel architectures.
Journal of Parallel and Distributed Computing, 4 (1987), 439–458.
- [23] Choudhary, A., Narahari, B., Nicol, D., and Simha, R. (1994)
Optimal processor assignment for pipeline computations.
IEEE Transactions on Parallel and Distributed Systems (Apr. 1994).
- [24] Snir, M., et al. (1995)
MPI the Complete Reference.
Cambridge, MA: The MIT Press, 1995.



Alok Choudhary received his Ph.D. from University of Illinois, Urbana-Champaign, in electrical and computer engineering, in 1989, M.S. from University of Massachusetts, Amherst, in 1986 and B.E. (Hons.) from Birla Institute of Technology and Science, Pilani, India in 1982.

He has been an associate professor in the Electrical and Computer Engineering Department at Northwestern University since September 1996. From 1989 to 1996 he was a faculty member in the ECE Department at Syracuse University. He has worked in industry for computer consultants prior to 1984. His main research interests are in high-performance computing and communication systems and their applications in many domains of information processing, datamining, and scientific computing. In particular, his interests lie in the design and evaluation of architectures and software systems, high-performance servers, and input-output.

Dr. Choudhary has served as program chair and general chair for several conferences in parallel and high-performance computing areas. He received the National Science Foundation's Young Investigator Award in 1993 (1993-1999). He also received an IEEE Engineering Foundation award, an IBM Faculty Development award and an Intel Research Council award.



Wei-keng Liao received a Ph.D. in computer and information science from Syracuse University in 1999.

He is a Research Assistant Professor in the Electrical and Computer Engineering Department at Northwestern University. His research interests are in the area of high-performance computing, parallel I/O, and data management for large-scale scientific applications.

Donald D. Weiner (S'54—M'60—SM'90—F'94—LF'97) received the S.B. and S.M. degrees from the Massachusetts Institute of Technology, Cambridge, MA, in 1956 and 1958, respectively, and the Ph.D. from Purdue University, West Lafayette, IN, in 1964, all in electrical engineering.

In 1964 he joined the Department of Electrical and Computer Engineering at Syracuse University as an Assistant Professor where he was promoted to Associated Professor in 1968 and Professor in 1974. He served as Department Chair from January 1993 through June 1996. Dr. Weiner became an Emeritus Professor in 1998 and continues to serve the Department as a Research Professor. His present research deals with signal processing in an interference environment including non-Gaussian noise. He has performed extensive work on the application of nonlinear systems analysis techniques and communications theory to electromagnetic compatibility problems.



Dr. Weiner has authored or coauthored over 40 journal articles, has presented over 80 papers at various conferences and has delivered many invited lectures. He coauthored a book with J. Spina on *The Sinusoidal Analysis and Modeling of Weakly Nonlinear Circuits*, published by Van Nostrand Reinhold Company in 1980. He also produced several films for the National Committee for Electrical Engineering Films released by the Education Development Center in 1988. In 1994 he was elected to the grade of Fellow in the IEEE for contributions to the analysis of nonlinear effects in electromagnetic compatibility. He received the Best Paper Award of the *IEEE Transactions on Electromagnetic Compatibility* in 1979, the General Electric Company Teaching Excellence Award in 1980, and the IEEE Electromagnetic Compatibility Society Certificate of Achievement for outstanding theoretical contributions to the analysis of nonlinear systems in 1985.

Pramod K. Varshney (S'72—M'77—SM'82—F'97) was born in Allahabad, India on July 1, 1952. He received the B.S. degree in electrical engineering and computer science (with highest honors), and the M.S. and Ph.D. degrees in electrical engineering from the University of Illinois at Urbana-Champaign in 1972, 1974, and 1976, respectively.

During 1972–1976, he held teaching and research assistantships at the University of Illinois. Since 1976 he has been with Syracuse University, Syracuse, NY where he is currently a professor of Electrical Engineering and Computer Science. He served as the Associate Chair of the department during 1993–1996. His current research interests are in distributed sensor networks and data fusion, detection and estimation theory, wireless communications, image processing, radar signal processing, and parallel algorithms.



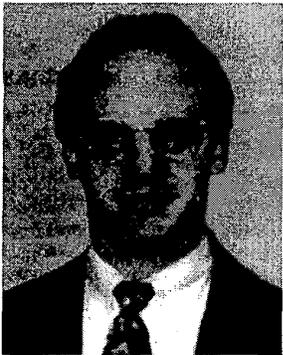
Dr. Varshney has authored and coauthored over sixty journal papers and over one hundred thirty conference papers. He is the author of *Distributed Detection and Data Fusion*, published by Springer-Verlag in 1997. He has consulted for General Electric, Hughes, SCEEE, Kaman Sciences Corp., Andro Consulting, Stiefvater Consulting and Digicomp Research Corp. While at the University of Illinois, Dr. Varshney was a James Scholar, a Bronze Tablet Senior, and a Fellow. He is a member of Tau Beta Pi and is the recipient of the 1981 ASEE Dow Outstanding Young Faculty Award. He was elected to the grade of Fellow of the IEEE in 1997 for his contributions in the area of distributed detection and data fusion. He was the guest editor of the special issue on data fusion of the *Proceedings of the IEEE*, January 1997. He is on the editorial boards of *Cluster Computing and Information Fusion*. He is listed in *Who's Who in Technology Today* and *Outstanding Young Men of America*.



Richard W. Linderman (S'81—M'84—SM'90) received his B.S.E.E. degree from Cornell University, Ithaca, NY, in 1980; and his M.Eng. (EE) degree and Ph.D. degrees from Cornell University, Ithaca, NY, in 1981 and 1984, respectively.

He served as Chair, Signal Processing Technology at the U.S. Air Force Research Laboratory (Rome Research Site) from 1988 to 1997. From 1984 to 1988, he served on the graduate faculty of the Air Force Institute of Technology teaching and conducting research in advanced computer architectures. His research interests include high performance computing architectures, VLSI architecture, advanced packaging, and digital signal processing applications.

Dr. Linderman is an AFRL Fellow.



Mark H. Linderman (M'95) received his B.S.E.E. from the University of Delaware, Newark, in 1986 and his M.Eng. in electrical engineering and Ph.D. degrees in 1990 and 1994, respectively from Cornell University, Ithaca, NY.

He has worked as an engineer at HRB Singer in State College, PA. He is currently the Technical Director at Air Force Research Laboratory, Rome Research Site, IFTC Branch. He also is the Director of the High Performance Computing Facility at Rome Research Site. His research interests include computer aided-design tools for VLSI and computer architecture for signal processing.

Dr. Linderman is a member of Eta Kappa Nu.



Russell D. Brown (SM'92—F'99) was born in Washington, DC on Sept. 5, 1949. He received a B.S.E.E. degree in 1972, and M.S.E.E. in 1973, from the University of Maryland, College Park, and Ph.D. in electrical engineering in 1995 from Syracuse University, Syracuse, NY.

From 1973 to 1978 he served as a Communications Electronics Officer for the U.S. Air Force, and since 1978 has performed research in radar signal processing at the USAF Research Laboratory in Rome, NY.