Design and Performance Evaluation of a Portable Parallel Library for Space-Time Adaptive Processing

James M. Lebak, *Member*, *IEEE*, and Adam W. Bojanczyk

Abstract—Space-time adaptive processing (STAP) refers to a class of methods for detecting targets using an array of sensors. Various STAP methods use similar operations on different data or in different orders. We have developed a portable, parallel library of subroutines for prototyping STAP methods. The subroutines work on the IBM SP2 and the Intel Paragon under three different operating systems and three different communication libraries, and can also be configured for other systems. We provide execution-time models for predicting the performance of each subroutine. Using the library routines, we created a parallel version of element-space pre-Doppler processing, three parallel versions of higher-order post-Doppler processing, and two versions of PRI-staggered post-Doppler processing. We implemented a fourth version of higher-order post-Doppler processing, the hybrid method, which uses a combination of fine-grain and coarse-grain parallelism to reduce execution time. The hybrid method can be used to improve performance when a large number of processors is available. Our execution time models generally predict the best method and predict execution times to within 10 percent or better for large test cases.

Index Terms—Space-time adaptive processing, portable software, library development, execution-time modeling, fine-grain parallelism.

1 Introduction

Recently, there has been significant interest in benchmarking and evaluating the use of massively parallel computers for space-time adaptive processing (STAP) methods. These methods operate on data collected by radar antennae. Targets detected after processing this data must be reported within a specified time interval. Parallel processing is often used to help meet this real-time deadline.

The goal of our research has been to design a library of subroutines to allow easy implementation of parallel methods for space-time adaptive processing. This paper describes a library for STAP algorithms which makes five major contributions toward this goal. First, we identify parallel modules from which many different STAP algorithms may be built. Second, we demonstrate a layered design for the modules which makes them portable between different communication libraries and operating systems on different parallel machines. Third, we implement routines which allow the user to easily specify the number of processors that work on particular subproblems. Fourth, we demonstrate how the modules may be used to build different parallel variations of specific STAP heuristic methods. Finally, we develop execution-time models of the subroutines on the different machines which may be used to predict the fastest method for a particular scenario, thus saving time that would otherwise be spent implementing and testing different methods.

Many other groups are conducting research into parallel processing and STAP. Research at the University of Southern California [1], [2] has made important contributions in this area. Rome Laboratory has developed a graphical user interface for experimenting with different STAP algorithms and evaluating results. MITRE Corporation has also produced a real-time STAP benchmark for parallel computers [3]. An object-oriented software library for STAP is being developed [4].

In this paper, we demonstrate the use of our parallel library in building benchmark STAP codes. Section 2 discusses features of the library design, Section 3 gives models of library routines, and in Section 4, we give results from parallel STAP programs built using the library. Details of the parallel library and complete results are presented in [5].

- *J.M. Lebak is with the MIT Lincoln Laboratory, Lexington, MA 02420. E-mail: jlebak@ll.mit.edu.*
- *A.W. Bojanczyk is with the Cornell University School of Electrical Engineering, Ithaca, NY 14853. E-mail: adamb@ee.cornell.edu.*

Manuscript received 21 July 1998; accepted 28 June 1999.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 107172.

2 Radar System Processing

Consider data consisting of the returns from M radar pulses, sampled for L different range increments, and recorded by a radar array with N sensor elements. This data may be considered as a series of data matrices A_j, $A_j \in \mathcal{C}^{L\times N}$, $j = 1, 2, \ldots, M$. We refer to $A = [A_1, A_2, ..., A_M]$, the concatenation of the matrices A_j, as the *data matrix*. This

1045-9219/00/$10.00 © 2000 IEEE

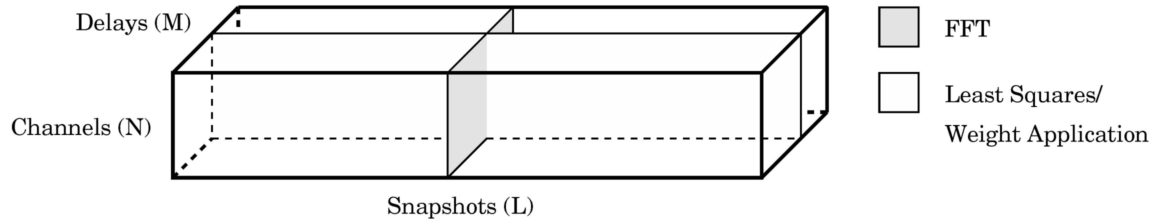

Fig. 1. STAP data cube showing the orientation of operations to be performed.

data matrix may also be considered as a three-dimensional *data cube* of M matrices of size $L \times N$.

The goal of radar processing is to detect targets in the data. Each target has a direction and a Doppler shift, specified by a *steering vector* v, and a range, corresponding to one of the L range increments in the data cube. Given v, a weight vector w is computed by calculating a QR factorization of the data matrix and solving a triangular system of linear equations. This is often referred to as *sample matrix inversion*, even though explicit matrix inversion is not performed. Matrix-vector multiplication between the data matrix and the weight vector is used to obtain the output vector y corresponding to each particular range and direction. Each such output is compared to a threshold to determine whether a target is present at the corresponding range. This process is described more fully elsewhere [6].

When the weight calculation and output vector formation steps are applied to the entire data matrix A, the algorithm is referred to as *joint-domain optimum* or *fully adaptive* space-time processing [7], [8], [9]. This method is expensive, and full statistical confidence in the results may require more data than is available. Therefore, various *partially adaptive* methods of space-time processing are used instead: an overview of these methods was given by Ward [6]. These methods reduce computation by transforming the data cube into a collection of smaller cubes prior to sample matrix inversion.

An important tool in partially adaptive STAP methods is the Doppler processing operation, which may be viewed as a weighted fast Fourier transform (FFT) operation. Ward [6] classifies space-time adaptive algorithms into four categories, depending on the type of transformation used and when Doppler processing is performed.

In an *element-space pre-Doppler* method, the data is broken into smaller problems, each involving a subset of the total number of pulses M. After the problems are solved, Doppler processing is performed to integrate the results. Doppler processing may also be performed prior to weight computation, in which case the method is termed an *element-space post-Doppler* algorithm. DiPietro [8] describes *higher-order post-Doppler processing*, where after Doppler processing, the computation is broken into smaller problems, the number of which is related to the order of the method. Another possible post-Doppler method, given by Brennan et al. [7], and also discussed by Ward and Steinhardt [9], is referred to as *PRI-staggered post-Doppler STAP*. This method performs FFTs on multiple subsets of the data to obtain the matrices for weight calculation.

Other methods, referred to as *beamspace algorithms*, combine the data from several sensors prior to weight vector calculation. As described by Ward [6], the primary difference between beamspace and element-space methods is the multiplication of the data matrix by a carefully selected beamforming matrix. The design of the beamforming matrix is more of a problem of signal processing than of parallel computing. Therefore, element-space pre-Doppler and post-Doppler methods were used as model implementations for the library. For reasons of space, we concentrate on the post-Doppler algorithms.

Three observations may be made about partially adaptive STAP methods. First, the operations involved in each method are similar, but may occur in a different order or involve different subsets of the data. Second, individual STAP algorithms may require performing the same operation several times on different data. Finally, the weight computation and filtering steps in STAP algorithms require data from orthogonal dimensions of the data cube.

These observations suggest details of the parallel implementation. The first two observations imply that a small number of modules, flexible with regard to how the input data subcubes are selected, will allow different algorithms to be implemented with a minimum amount of additional programming. The third observation is very important, as it implies that interprocessor communication will be required if the overall computation is to be performed on multiple processors.

Figures given in Ward [6] imply a sustained computational power on the order of 10^{11} to 10^{13} operations per second would be required to implement the joint-domain optimum algorithm in real time. For a higher-order post-Doppler method, the computational requirement is on the order of 10^{9} to 10^{11} operations per second. These figures are in the upper range of the current sustained performance of large parallel supercomputers. Therefore, it is important to provide flexibility in the development of parallel STAP

methods, so that programs for new methods may easily be created, experimented with, and transported to future parallel systems.

3 Design of Parallel Software for Partially Adaptive STAP Methods

To characterize parallel approaches to reduced-dimension STAP methods, consider data partitioning of an element-space post-Doppler algorithm. Three alternatives may be identified, based on the decision to distribute particular operations. Fig. 1 shows the data necessary for FFT operations and for the weight calculation. One alternative is to distribute the data so that each FFT operation may be performed on a single processor but the processors must work together to calculate the weights. Similarly, the data may be distributed so that the processors must cooperate on the FFT step, but can work independently on weight calculations. A final alternative is to initially distribute the data so that the FFT can be performed on a single processor and change the distribution during the algorithm so that each weight calculation is performed on a single processor. We refer to these different methods as the *parallel QR*, *parallel FFT*, and *transpose* methods, respectively.

These methods can be implemented with only a small number of parallel modules: single-processor and multiple-processor versions of QR factorization, triangular solve, and FFT. The versions of these algorithms written for this project combine messages for multiple problems and do not make use of ScaLAPACK [10], [11]. The QR factorization and triangular system solution methods implemented for this project use row distribution of matrices. They are based on work by O'Leary and Whitman [12] and Li and Coleman [13], respectively. The parallel FFT method used may be found in Van Loan [14].

In the transpose method, all-to-all communication is required to change the distribution of the matrix between the FFT and weight calculation steps. Two main classes of all-to-all communication algorithms exist, known as *direct send* and *store-and-forward* algorithms [15], [16]. Both the ScaLAPACK matrix transpose algorithm and our algorithm are based on direct send [17]. Initial implementations showed that the direct send method outperformed the store-and-forward method [18].

Three target machines were available for this study: an IBM SP2 at the Cornell Theory Center, an Intel Paragon at Rome Laboratory, and another Intel Paragon at Sandia National Laboratory. These machines each use a different operating system; the SP2 uses AIX 4.1, the Rome Paragon uses Intel's OSF/1 Unix implementation version 1.4, and the Sandia Paragon runs the Sandia/University of New Mexico operating system (SUNMOS [19]). Each includes a version of the standard message-passing interface library, MPI [20], and a machine-specific communication library (MPL [21] on the SP2, NX [22], [23] on the Paragon). The Intercom library [24] is used to provide collective operations usable on subsets of processors when NX is used on the Paragon. LAPACK [25] and optimized versions of the BLAS [26], [27], [28] (ESSL [29] on the SP2, the Kuck and Associates math library on the Paragon [30]) are available on each machine. Each library also includes a single-processor FFT subroutine.

In implementing even a limited set of modules on these different systems, several complications arise related to differences in communication and math libraries on the different machines. To overcome these difficulties and provide the flexibility desired, the modules were implemented using a layered design. Communication and math layers allowed the modules to run independent of the hardware, communication, and math libraries used. These communication and math layers are described in more detail in the following sections.

3.1 Communication Layer

The communication interface layer is modeled on MPI, but contains only a subset of the full MPI functionality, since several MPI features (for example, derived datatypes) are not found in NX, Intercom, or MPL, and were not needed in this application. It abstracts the varying implementations of collective communication operations in the three communication libraries, imposing a consistent view of groups and contexts across the libraries. It supports blocking and nonblocking sends and receives. In addition, the following group functions are supported: all-to-all communication, scatter, broadcast, and reduction. Variants of all of these operations are found in each of the chosen communication libraries, with the exception of all-to-all communication, which is not found in the Intercom library. A simple direct-send algorithm was implemented for all-to-all communication when using the Intercom library. For p processors, this algorithm sends to a maximum of c other processors in each of p/c stages for some small value of c, compromising between contention and synchronization.

3.2 Math Layer and Matrix Structure

The ESSL library on the SP2 assumes that data is stored in column-major order, while the Kuck and Associates library on the Paragon assumes data is stored in row-major order. The use of different storage orders on different machines decreases the portability of code and is also directly related to performance in many cases. The math layer is designed to allow the programmer who is aware of these effects to store a matrix in an advantageous order, while minimizing the effects on portability.

A matrix interface provides support for allocating and referring to elements of matrices, including matrices stored on a single processor and parallel matrices distributed according to a *block-scattered* distribution [10]. Macros and function interfaces are provided to allow the necessary BLAS, LAPACK, and other functions (QR/LQ, triangular system solution, matrix-vector multiplication, matrix-matrix multiplication, FFT) to be called for arbitrary submatrices of larger matrices regardless of the underlying storage order. A "multiple problem descriptor" structure was created which allows the user to specify a regular one-dimensional or two-dimensional pattern of storage for submatrices within a larger matrix. These descriptors are used within functions to determine indices and control loop iterations so functions may be easily reused in different situations without being rewritten.

3.3 Library Configuration

A major goal of this project is that the parallel STAP programs should be easily portable and reconfigurable. The program `imake` was employed to achieve this goal. The `imake` program creates makefiles using a high-level description of the target and a database of files that specify the compilers, compiler flags, libraries, utilities, and procedures to use to build target programs [31]. The utility allows program dependency descriptions to become transportable with the code. The names of the communication and BLAS libraries, the library path, and other variables can be stored in a single configuration file. To move to a new platform, library, or operating system, only the configuration file needs to be edited.

4 Models

As described in a previous report [18], we have developed execution-time models of each parallel module. These models are designed to give the programmer an idea of the performance which may be expected from each component for a particular size of problem. Such models allow appropriate mapping of the parallel problem without requiring multiple implementations to be tested.

We describe the behavior of each module in terms of computation and communication parameters. These parameters are found by performing a least-squares fit to execution time data. Models of individual modules are combined to produce models of the overall execution time. Brehm et al. [32] showed that a similar approach produced good models of a message-passing parallel program used to solve the nonlinear shallow water equations.

The communication time model used here is the very simple linear model. The time to send a message of n bytes between two processors is assumed to be

$$T_{comm} = \alpha + n\beta, \quad (1)$$

where α is the message startup time and β is the time required to send one byte.

The computation time model is an adaptation of the widely-used first-order model of computation given by Hockney [33]. Since higher-level modules involving multiple vector operations are being modeled, the coefficient of overhead in each module is assumed to be proportional to the number of "steps" s performed,

$$T_{comp} = \gamma F + \delta s, \quad (2)$$

where F is the number of floating-point operations to be performed, γ summarizes the computation performance of the machine, and δ represents "startup time" for a particular vector operation such as an inner or outer product.

The model of each module is an algebraic expression involving the parameters α, β, γ, and δ. The coefficients of each parameter are expressions corresponding to the problem and machine size. The expressions for each module are given in Table 1.[1] For the parallel matrix transpose, terms involving γ and δ are included even though no floating-point computation is performed. These terms reflect the cost of data movement on the local processor.

To obtain the values of the parameters, each module was executed multiple times for many different coefficient values, corresponding to different problem sizes. When recording execution times, two initialization runs were performed, and then the execution time was recorded for five subsequent runs using the same problem and machine size. The execution time for that machine and problem size was considered to be the minimum execution time over the five recorded runs.[2] A least-squares problem involving a matrix of computed coefficient values and the resulting execution times was solved to obtain the values of the model parameters. These parameters are given in Table 2. Estimated execution times from these models are given in the next section.

5 Parallel Algorithm Demonstration

The goal of our research was to provide a set of modules to allow multiple parallel versions of STAP methods to be built and to demonstrate that the performance of these versions was predictable. In this section, we show predicted and actual execution times for multiple parallel STAP

1. The parallel triangular solve model is somewhat complicated, and is not given here; see Lebak [5] or Li and Coleman [13].

2. On the IBM SP2 and the Intel Paragon, executing the initialization runs greatly improves the obtained execution time. It is assumed that a real radar processing program would be executing a continuous loop and processing data as it arrives; the initialization runs simulate the performance that would be achieved by such a program. The same approach is used in other studies, for example, Xu and Hwang [2].

TABLE 1
Model Expressions and Test Data Sized Used to Obtain Parameters for Each Model

Operation	Model Equation	Test Data Sizes
Send/receive n bytes	$\alpha + n/\beta$	$n \in \{8, 16, \ldots, 72000\}, p = 2$
Broadcast/reduction, n bytes	$(\alpha + n/\beta)\log_2(p)$	$n \in \{8, 16, \ldots, 72000\}$, $p \in \{2, 4, 8, 16\}$
QR, $\kappa n \times n$ matrix	$\gamma 8n^3(\kappa - \frac{1}{3}) + \delta n$	$n \in \{10, 20, \ldots, 100\}, \kappa \in \{3, 4, 5\}$
k Row-wrap QR, $\kappa n \times n$ matrix	$\frac{8kn^3}{p}(\kappa - \frac{1}{3})\gamma + \delta kn +$ $2(n\alpha + 4kn^2\beta)\log_2 p$	$n \in \{10, 20, \ldots, 100\}, \kappa \in \{3, 4, 5\}$, $p \in \{2, 4, 8, 16\}, k \in \{2, 4, 8, 16\}$
Triangular solve, $n \times n$ matrix	$4n^2\gamma + n\delta$	$n \in \{10, 20, \ldots, 100\}$
κn FFTs of length k	$5\gamma\kappa nk\log_2 k + \kappa n\delta\log_2 k$	$k \in \{8, 16, 32, 64\}$, $n \in \{10, 20, \ldots, 60\}, \kappa = 3$
κn parallel FFTs of length k	$\frac{8\gamma\kappa nk\log_2 k}{p} + 6\alpha\log_2 p +$ $32\frac{\beta\kappa nk\log_2 p}{p} + \delta\kappa n\log_2 k$	$k \in \{16, 32, 64, 128\}, \kappa = 3$, $n \in \{10, 20, \ldots, 60\}, p \in \{2, 4, 8, 16\}$
Transpose, $\kappa n \times nk$ matrix	$16\frac{\kappa kn^2(p-1)}{p^2}\beta +$ $16\frac{\kappa kn^2}{p}\gamma + kn\delta$	$n \in \{10, 20, \ldots, 100\}, \kappa = 3$, $p \in \{2, 4, 8, 16\}, k \in \{16, 32, 64\}$

In all cases, p is the number of processors.

programs. The programs were tested on all three parallel machines available to us and under each communication library. This paper only reports results for the SP2 using MPI and the Sandia Paragon under SUNMOS using NX/Intercom. These combinations of operating system and library produced the best performance on each machine [5].

5.1 Parallel Programs

The parallel modules were used to implement an element-space pre-Doppler method, four parallel variations of higher-order post-Doppler processing, and two parallel variations of PRI-staggered post-Doppler processing. We concentrate on the post-Doppler algorithms in this section to emphasize the parallel variations available with the modules: these variations could be applied to other methods, as well.

The higher-order post-Doppler processing (HOPD [8]) and PRI-staggered post-Doppler processing (PSPD [9]) methods can be implemented in three basic parallel variations. These are the parallel QR, parallel FFT, and transpose methods referred to in Section 2. Results for earlier versions of the parallel QR and parallel FFT methods were reported elsewhere [18], [34]: new versions of each method were implemented for this work using the parallel modules.

5.1.1 Data Sets

Sixteen different values of the parameters L (number of range samples), M (number of pulses), and N (number of channels) and the order d were used to evaluate the parallel programs. Parameter values were chosen based on examples given by Ward [6], Brennan et al. [7], and Di Pietro [8]. Names of and values in each parameter set are shown in Table 3.

Data sets a and c use small values of M and N, and data sets b and d use larger values of M and N. Within data sets a and c, the values of M, N, and L are varied so as to hold the memory use constant. In data sets b and d, M and L are held constant, and N varies from 16 to 64. Due to space limitations, only results for a selected subset of these data sets are given here. More complete results can be found in Lebak [5].

5.1.2 Hybrid Method

As the transpose method offers no option for reducing execution time when the real-time deadline is not met using the maximum number of processors (that is, $p = M$), we

TABLE 2
Model Parameters for Each Module

	SP2 Parameters				Paragon Parameters			
Operation	α μs	β ns/byte	γ ns/flop	δ μs	α μs	β ns/byte	γ ns/flop	δ μs
Send/receive	106.4	38.5			33.8	6.24		
Broadcast/reduction	154	58.5			135	58.6		
QR			5.73	37.3			11.37	20.8
Row-wrap QR	56.1	114.2	9.05	71.5	50.7	87.4	11.11	246.6
Triangular solve			8.98	0.85			2.65	12.9
FFT			5.3	3.37			16.8	3.53
Parallel FFT	74.3	49.9	42.3	50.6	90.8	42.3	110.1	2.25
Transpose		18.6	4.84	18.8		7.86	18.6	9.17

implemented a *hybrid* method that combines the purely coarse-grain parallelism of the transpose method with the fine-grain parallelism of the parallel QR method. As in the transpose method, the hybrid method begins with the data distributed so that the FFTs may be performed on a single processor. During redistribution, the data must be arranged so that small groups of processors may work together on the weight calculation. Assume the machine size is $p = p_w p_g$, where p_w is the number of processors that work together on each weight calculation and p_g is the number of processor groups. The number of groups is constrained to be $p_g \leq M$, where M is the number of problems, but p_w may be as large as is practical for a given problem. The transpose method is the case when $p_w = 1$: in this sense, the hybrid method is an extension to the transpose method to allow more processors to be used. This extension is easily implemented with the parallel modules.

TABLE 3
Data Cube Sizes Used for Evaluation Purposes

Set Name	L	M	N	d	Set Name	L	M	N	d
a1	250	32	32	1	b1	500	64	16	1
a2	500	32	16	1	b2	500	64	32	1
a3	500	16	32	1	b3	500	64	48	1
a4	1000	16	16	1	b4	500	64	64	1
c1	250	32	32	2	d1	500	64	16	2
c2	500	32	16	2	d2	500	64	32	2
c3	500	16	32	2	d3	500	64	48	2
c4	1000	16	16	2	d4	500	64	64	2

5.2 Test Results

For our purposes, the ability of the models to accurately predict execution time is of as much interest as the performance of the programs. The predicted and actual execution times are shown on the same graph. The quality of the fit from a model for a particular data set and series of machine sizes will be determined by computing the relative error of the fit at each point and averaging over all points tested. This statistic will be referred to as the mean relative error for a particular method, data set, and machine.

5.2.1 Higher-Order Post-Doppler Basic Implementations

Figs. 2 and 3 show the actual and predicted execution times for data set a1. The transpose method performs better than either of the other two methods. On each machine, the parallel QR method starts out faster than the parallel FFT method, but as the number of processors increases the parallel FFT method becomes faster. For the parallel QR method, the actual and predicted execution times increase for 32 or more processors.

As L increases, the models predict that the parallel QR method should exhibit better performance than the parallel

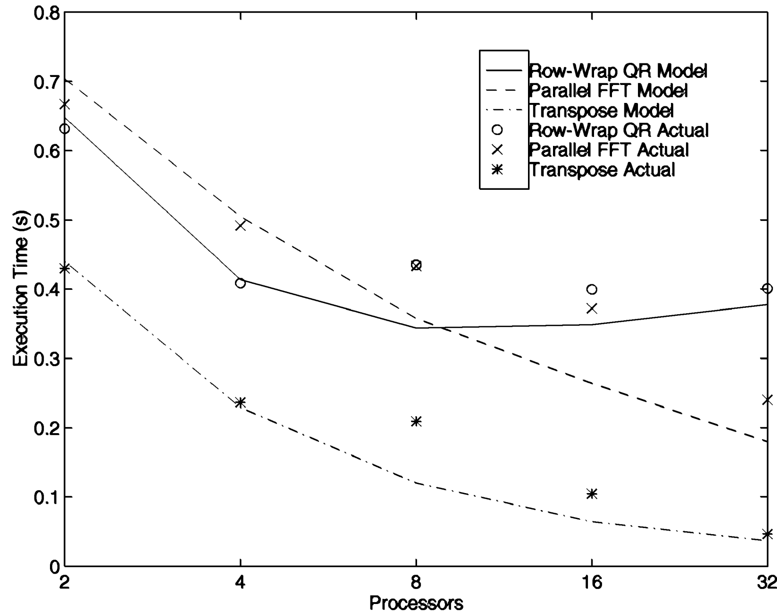

Fig. 2. Comparison of actual and predicted execution time of element-space post-Doppler STAP on the SP2 for three parallel methods for data set a1.

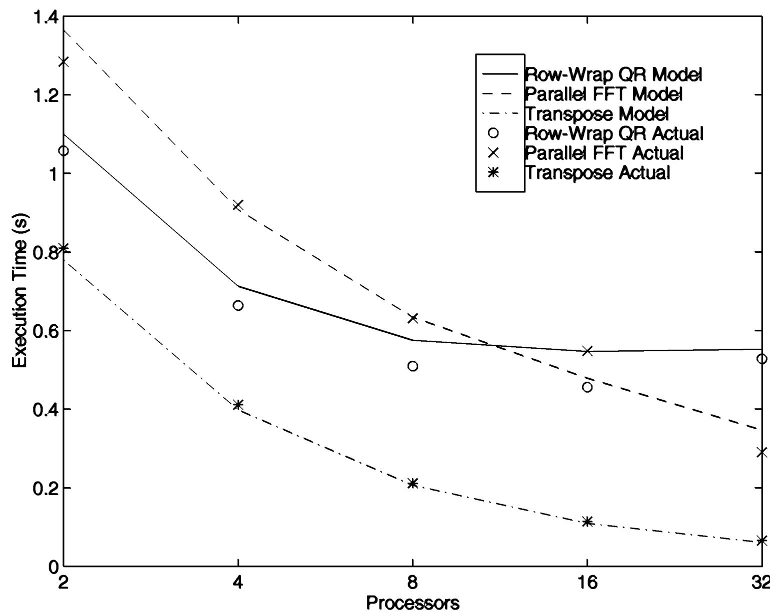

Fig. 3. Comparison of actual and predicted execution time of element-space post-Doppler STAP on the Paragon for three parallel methods for data set a1.

FFT method. That this is indeed the case is illustrated in Fig. 4, which shows the performance on data set a4 on the Paragon. In this data set, L is four times as large as in data set a1. For a small number of processors, the performance of the parallel QR method is nearly equal to the performance of the transpose method, but the transpose method scales better. The parallel FFT method is clearly inferior to either for this particular data set.

As the number of columns in the matrix increases, the parallel QR method does not perform as well as the others. This method also does not scale as well as the others. The parallel QR method can show decent performance for a small number of columns and a large matrix aspect ratio. The parallel FFT method is usually a better choice than the parallel QR method for an order d greater than one or for a large number of processors. However, in all cases, the transpose method demonstrates better performance than the other two, sometimes by quite a large margin.

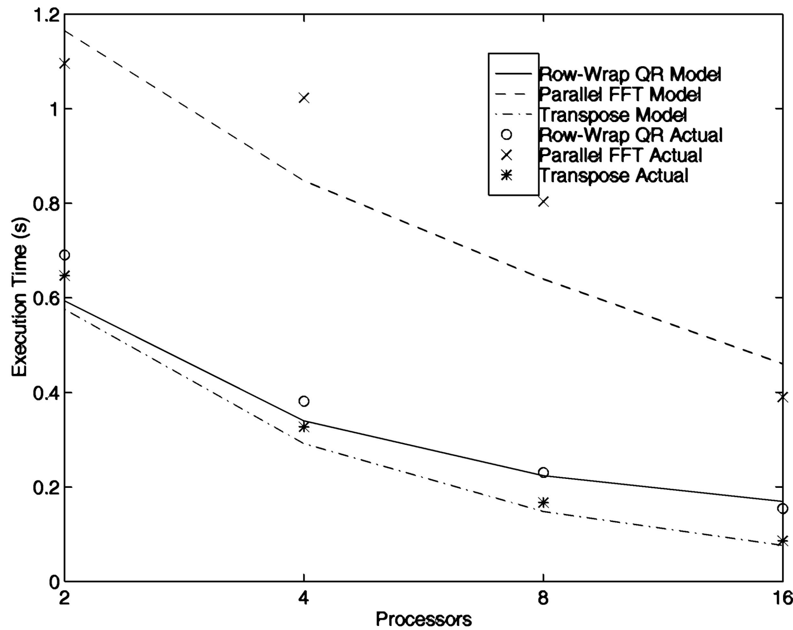

Fig. 4. Comparison of actual and predicted execution time of element-space post-Doppler STAP on the Paragon for three parallel methods for data set a4.

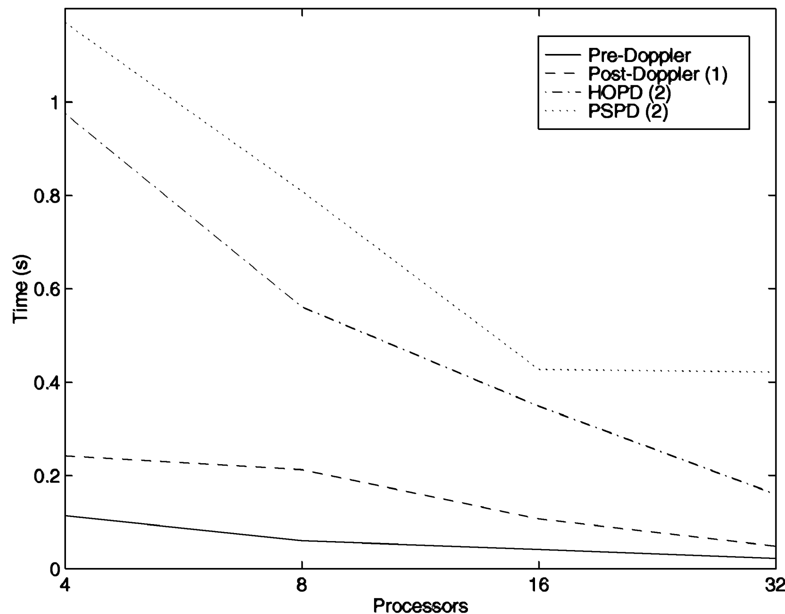

Fig. 5. Comparison of execution time of the pre-Doppler, order-1 post-Doppler, order-2 PRI-staggered post-Doppler, and order-2 higher-order post-Doppler methods on the SP2.

5.2.2 PRI-staggered Post-Doppler Implementations

As an exercise in the use of the library in fast prototyping, parallel QR and transpose implementations of the PRI-staggered post-Doppler method were created from the library routines. When the order d is equal to one, the PSPD and HOPD algorithms are the same, allowing easy verification. Additional software development time for the PSPD methods was very small. The matrix and problem descriptor structures discussed in Section 2 allowed reuse of the high level code for the higher-order post-Doppler method without requiring rewriting of the low-level code. Implementation of the two PRI-staggered methods was completed in less than 10 hours of work performed in a single 24-hour period. The methods were fully portable between the SP2 and the Rome Paragon once implemented.

In terms of the relative performance of the parallel QR and transpose implementations, the results for the PSPD method are much the same as for the HOPD method and so are not shown here. The models predict this behavior, showing that they are applicable to different algorithms. Fig. 5 compares the execution time of the pre-Doppler

TABLE 4
Execution Time of the Transpose and Hybrid Methods for Higher-Order Post-Doppler Processing on the SP2 and the Paragon

		SP2			Paragon		
			Hybrid			Hybrid	
Set	M	Transpose	$p_w = 2$	$p_w = 4$	Transpose	$p_w = 2$	$p_w = 4$
a1	32	**48.2**	54.9	72.3	71.3	65.1	**61.5**
a2	32	37.5	60.9	**34.6**	49.4	41.8	**40.3**
a3	16	121.6	**72.8**	80.7	119.8	82.3	**60.5**
a4	16	75.1	50.5	**38.9**	88.4	58.3	**38.8**
b1	64	73.96	98.22	**73.69**	63.7	63.2	**49.6**
b2	64	144.70	**126.19**	206.49	132.3	108.5	**78.9**
b3	64	**158.17**	177.79	213.41	225.2	171.6	**121.0**
b4	64	317.93	**254.56**	342.59	338.5	253.0	**168.8**
c1	32	**159.4**	264.9	307.6	281.3	280.1	**209.1**
c2	32	**126.9**	160.3	189.0	178.0	153.3	**115.8**
c3	16	**328.1**	377.7	346.0	511.3	376.1	**255.1**
c4	16	232.0	375.9	**184.3**	323.0	217.6	**140.9**
d1	64	**190.16**	268.28	367.10	215.4	176.1	**129.9**
d2	64	**365.91**	534.48	529.56	596.2	450.2	**303.3**
d3	64	**673.78**	919.08	930.47	1154.8	846.0	**532.8**
d4	64	**1064.87**	1350.42	1164.35	1862.1	1327.6	**819.8**

Time is given in ms. The fastest execution time for each data set on each machine is indicated in **boldface**.

method, the order-1 post-Doppler transpose method, and the PSPD and HOPD transpose methods with order 2. The data sets correspond either to set a1 (for the order-1 methods) or c1 (for the order-2 methods). The execution times for the methods are in direct proportion to their accuracy. The PSPD method, which took the longest time to execute, is generally considered to be more accurate than the HOPD method, and either of them are more accurate than the order-1 post-Doppler method, which is in turn more accurate than the pre-Doppler method [6].

5.2.3 HOPD Hybrid Implementation

Table 4 shows the time achieved by the transpose method for each data set for the two machines and compares with the times achieved by the use of two or four processors per problem using the hybrid method. The fastest execution time for each particular data set is shown in boldface. The results are very encouraging. Depending on the data set, the use of two or four processors per problem can reduce the execution time, sometimes by as much as a factor of 2. The hybrid method appears especially attractive as the size of the problem increases.

Table 4 shows that the Paragon is much faster than the SP2 in many cases, particularly for data set d. This result cannot be used to make general statements about the performance of the two machines, since the code here was constructed with portability as a priority over performance. Nonetheless, it is an interesting and unexpected result.

The conclusion, then, is that for the Paragon, the hybrid method is often a good way of achieving higher performance if additional processors are available. On the SP2, the transpose method should be used instead of the hybrid method in many cases, although for small data sets the hybrid method is worth considering.

The code for the hybrid implementation is very similar in form to the other implementations, thanks to the parallel modules described in Section 2. No additional work was required to transport the program between the two machines, which is a clear benefit of the portable nature of the modules.

5.2.4 Model Accuracy

For the parallel QR, parallel FFT, and transpose methods, the models predict the best method and the general tendencies of each method quite well, as was seen in the previous sections. The actual results are not always close, though the trend is toward increased accuracy as the amount of computation increases. This trend can be seen in Table 5, where the mean relative error for each method, data set, and machine is given for data sets b1-b4. The workload increases in these sets from set b1 to set b4 (see Table 3).

The models for the hybrid method on the Paragon give relative errors that are very similar in magnitude to the relative error shown for the other methods, indicating that the models can reliably be used to predict when the hybrid method will be of use on the Paragon. Unfortunately, the models were not a very reliable indicator of the performance of the hybrid method for the SP2. This most likely was caused by a heavy multiuser workload on the SP2 when the timing experiments were conducted.

6 CONCLUSIONS

Space-time adaptive radar algorithms come in many variations and allow multiple parallel implementations. The fundamental characteristics of the parallel STAP methods are:

1. The operations involved in each method are similar, but may occur in a different order, or involve

TABLE 5
Mean Error in the Predicted Execution Times Relative to the Actual Execution Times for the Higher-Order Post-Doppler Method for Data Sets b1-b4 on the SP2 and the Paragon

	SP2					Paragon				
				Hybrid					Hybrid	
Set	QR	FFT	T.	$p_w = 2$	$p_w = 4$	QR	FFT	T.	$p_w = 2$	$p_w = 4$
b1	16.40	22.86	32.48	75.3	69.1	22.7	14.4	16.5	44.6	36.7
b2	9.44	25.39	35.61	54.7	75.8	14.9	17.4	9.02	23.9	14.5
b3	6.59	22.87	25.42	43.5	61.3	16.6	13.6	6.8	16.0	9.7
b4	5.46	21.83	24.84	39.5	64.9	15.23	13.15	5.38	13.3	5.8

different subsets of the data, for each particular method.

2. In any particular method, the same operations may be performed multiple times with different data.
3. Data from orthogonal dimensions of the data cube is required in consecutive operations.

The goal of this research was to create subroutines that allow a radar system designer to easily parallel STAP algorithms.

We identified and demonstrated a set of portable, flexible modules for implementing parallel STAP algorithms. The modules are built using a portable communication layer and an object-based math layer. The communication layer can be transported to three different communication libraries on two different parallel machines under three different operating systems. This layer can be redefined to allow transportation to other machines and operating systems. The math layer allows allocation of matrices in an efficient storage order and easy description of multiple problems contained in a single matrix. Together, these layers give the designer control over the assignment of subproblems to groups of processors.

The modules were used to implement and demonstrate eight different parallel STAP methods. All methods implemented using the modules are easily portable and can run using different numbers of processors.

Of the three basic parallel higher-order post-Doppler methods, the transpose method has the best performance. However, the number of processors that may be used to improve the execution time of this method is limited to the number of problems being solved. The parallel QR is most useful when the matrices involved in a particular problem have a small number of columns and a large number of rows. The hybrid method overcomes the limitation of the transpose method through a combination of coarse-grain and fine-grain parallelism. The modules allow the user to easily change both the number of processors used and the degree to which coarse-grain and fine-grain parallelism is employed in the calculation.

Use of the hybrid method can reduce execution time by a factor of 2 or more. This demonstrates that while coarse-grain parallelism should be used as much as possible, fine-grain parallelism can contribute substantially to the performance of parallel programs and should not be ignored by parallel program developers.

Overall, the parallel modules presented here make the use of either fine-grain or coarse-grain parallelism in STAP methods easier for the programmer. Experimentation with different algorithms, architectures, and software is possible by reconfiguring the modules. The flexibility of the modules makes them a valuable tool for the radar system designer.

ACKNOWLEDGMENTS

This effort was sponsored in part by the Advanced Research Projects Agency and the Air Force Research Laboratory/IFTC, under grant number F30602-95-1-0016. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Advanced Research Projects Agency, the Air Force Research Laboratory, or the U.S. government.

The authors acknowledge the use of three parallel computers. The first, the 512-node IBM PowerParallel SP2, is located at the Cornell Theory Center, which receives major funding from the National Science Foundation (NSF)

and New York State. Additional funding comes from the Advanced Research Projects Agency (ARPA), the National Institutes of Health (NIH), IBM Corporation, and other members of the Center's Corporate Research Institute. The primary Paragon experiments were performed on the 64-node and 1840-node Intel Paragon supercomputers located at the Massively Parallel Computing Research Laboratory (MPCRL) at Sandia National Laboratories. The authors are also indebted to Dr. M. Linderman and the system staff at the Air Force Research Laboratory High Performance Computing Facility, who provided access to a 321-node Intel Paragon supercomputer.

This work was performed while Dr. J.M. Lebak was at the Cornell University School of Electrical Engineering.

REFERENCES

[1] K. Hwang, Z. Xu, and M. Arakawa, "Benchmark Evaluation of the IBM SP2 for Parallel Signal Processing," *IEEE Trans. Parallel and Distributed Systems,* vol. 7, no. 5, pp. 522–535, May 1996.

[2] Z. Xu and K. Hwang, "Modeling Communication Overhead: MPI and MPL Performance on the IBM SP2 Multicomputer," *IEEE Parallel and Distributed Technology,* vol. 7, pp. 9–23, Mar. 1996.

[3] J.A. Torres and R.T. Williams, "RT-STAP: Real-Time Space-Time Adaptive Processing Benchmark," Technical Report MTR 96B0000021, MITRE Corporation, 1996.

[4] C.M. DeLuca, C.W. Heisey, R.A. Bond, and J.M. Daly, "A Portable Object-Based Parallel Library and Layered Framework for Real-Time Radar Signal Processing," *ISCOPE '97: First Conference on Int'al Scientific Computing in Object-Oriented Parallel Environments,* Dec. 1997.

[5] J.M. Lebak, *Portable Parallel Subroutines for Space-Time Adaptive Processing,* PhD thesis, Cornell Univ., 1997.

[6] J. Ward, "Space-Time Adaptive Processing for Airborne Radar Data Systems," Technical Report 1015, Massachusetts Inst. of Technology Lincoln Laboratory, Lexington, Mass., Sept. 1994.

[7] L.E. Brennan, D.J. Piwinski, and F.M. Staudaher, "Comparison of Space-Time Adaptive Processing Approaches Using Experimental Airborne Radar Data," *Proc. 1993 National Radar Conf.,* pp. 176–181, Mar. 1993.

[8] R.C. Di Pietro, "Extended Factored Space-Time Processing for Airborne Radar Systems," *26th Ann. Asilomar Conf. Signals, Systems, and Computing,* pp. 425–430, 1992.

[9] J. Ward and A.O. Steinhardt, "Multiwindow Post-Doppler Space-Time Adaptive Processing," *Proc. IEEE Seventh SP Workshop Statistical Signal and Array Processing,* pp. 461–464, June 1994.

[10] J. Choi, J.J. Dongarra, L.S. Ostrouchov, A.P. Petitet, D.W. Walker, and R.C. Whaley, "The Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines," Technical Report ORNL/TM-12470, Oak Ridge National Laboratory, Oak Ridge, Tenn., Sept. 1994.

[11] J. Choi, J.J. Dongarra, and D.W. Walker, "The Design of a Parallel Dense Linear Algebra Software Library: Reduction to Hessenberg, Tridiagonal, and Bidiagonal Form," Technical Report ORNL/TM-12472, Oak Ridge Nat'l Laboratory, Oak Ridge, Tenn., Jan. 1995.

[12] D.P. O'Leary and P. Whitman, "Parallel QR Factorization by Householder and Modified Gram-Schmidt Algorithms," *Parallel Computing,* vol. 16, pp. 99–112, 1990.

[13] G. Li and T.F. Coleman, "A Parallel Triangular Solver for a Distributed-Memory Multiprocessor," *SIAM J. Scientific and Statistical Computing,* vol. 9, pp. 485–502, May 1988.

[14] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform.* Soc. for Industrial and Applied Mathematics, 1992.

[15] S.L. Johnsson and C.-T. Ho, "Algorithms for Matrix Transposition on Boolean n-cube Configured Ensemble Architectures," *SIAM J. Matrix Analysis and Applications,* vol. 9, pp. 419–454, July 1988.

[16] S.H. Bokhari, "Multiphase Complete Exchange: A Theoretical Analysis," Technical Report 93-64, Inst. for Computing Applications in Science and Eng., Hampton, Va., Aug. 1993.

[17] J. Choi, J.J. Dongarra, and D.W. Walker, "Parallel Matrix Transpose Algorithms on Distributed Memory Concurrent Computers," Technical Report ORNL/TM-12309, Oak Ridge National Laboratory, Oak Ridge, Tenn., Oct. 1993.

[18] J.M. Lebak, R.C. Durie, and A.W. Bojanczyk, "Toward a Portable Parallel Library for Space-Time Adaptive Methods," Technical Report CTC96TR242, Cornell Theory Center, June 1996.

[19] A.B. Maccabe, K.S. McCurley, R. Riesen, and S.R. Wheat, "SUNMOS for the Intel Paragon: A Brief User's Guide," *Proc. Intel Supercomputer Users' Group, 1994 Annual North America Users' Conf.,* pp. 245–251, June 1994.

[20] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," Technical report, Univ. of Tennessee, Knoxville, Apr. 1994.

[21] Cornell Theory Center, "Introduction to MPL: Parallel Programming on the IBM SP1 Workshop," Apr. 1994

[22] Intel Supercomputer Systems Division, *Paragon User's Guide,* 1.2 beta ed., Beaverton, Ore., 1994.

[23] K.S. McCurley, "Intel NX Compatibility under SUNMOS," Technical Report 93-2618, Sandia Nat'l Laboratories, Albuquerque, N.M., Nov. 1993.

[24] P. Mitra, D.G. Payne, L. Shuler, R. van de Geijn, and J. Watts, "Fast Collective Communication Libraries, Please," Technical Report TR-95-22, Dept. of Computer Sciences, Univ. of Texas, June 1995.

[25] E. Anderson, Z. Bai, C. Bishof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK: Users' Guide.* SIAM Press, 1992.

[26] C.L. Lawson, R.J. Hanson, D.R. Kincaid, and F.T. Krogh, "Basic Linear Algebra Subprograms for Fortran usage," *ACM Trans. Math. Software* vol. 5, pp. 308–323 Sept. 1979

[27] J.J. Dongarra, J. Du Croz, S. Hammarling, and R.J. Hanson, "An Extended Set of FORTRAN Basic Linear Algebra Subprograms," *ACM Trans. Math. Software,* vol. 14, pp. 1–17, Mar. 1988.

[28] J.J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff, "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. Mathematical Software,* vol. 16, pp. 1–17, Mar. 1990.

[29] IBM Corporation, *Engineering and Scientific Subroutine Library.* Release 4, Kingston, N.Y., Mar. 1990.

[30] Kuck and Assoc., *CLASSPACK Basic Math Library User's Guide.* Release 1.1, Champaign, Ill., Mar. 1991.

[31] P. DuBois, *Software Portability with imake.* O'Reilly and Associates, 1993.

[32] J. Brehm, P.H. Worley, and M. Madhukar, "Performance Modeling for SPMD Message-Passing Programs," Technical Report ORNL/TM-13254, Oak Ridge Nat'l Laboratory, Oak Ridge, Tenn., June 1996.

[33] R.W. Hockney, *The Science of Computer Benchmarking.* SIAM, 1996.

[34] S.J. Olszanskyj, J.M. Lebak, and A.W. Bojanczyk, "Parallel Algorithms for Space-Time Adaptive Processing," *Proc. Ninth Int'l Parallel Processing Symp.* pp. 77–81, Apr. 1995.

James M. Lebak received BS degrees in mathematics and electrical engineering in 1989, and the MS degree in electrical engineering in 1991 from Kansas State University, Manhattan, Kansas. He received the PhD degree in electrical engineering from Cornell University, Ithaca, New York, in 1997. Since October of 1996, he has been employed by the MIT Lincoln Laboratory as a technical staff member in the Embedded Digital Systems group. Dr. Lebak is a member of the Society for Industrial and Applied Mathematics, and serves as cochair of the vector, signal, and image processing library (VSIPL) forum, a DARPA-initiated effort to standardize mathematical library interfaces on embedded systems. He is interested in parallel and numerical algorithms for digital signal processing.

Adam W. Bojanczyk graduated from the University of Warsaw, Warsaw, Poland. From 1983 to 1985, he was a postdoctoral fellow in the Center for Mathematical Analysis at the Australian National University, Canberra, Australia. In 1986, he joined the computer science faculty in Washington University, St. Louis, Missouri. In 1987, he joined Cornell University, where he is a faculty in the School of Electrical Engineering. He is also associated with the Center of Applied Mathematics and Theory Center at Cornell. His research interests are parallel algorithms and architectures for signal processing.