

# Parallel Implementation of a Class of Adaptive Signal Processing Applications \*

Myungho Lee, Wenheng Liu<sup>†</sup> and Viktor K. Prasanna

Department of EE-Systems, EEB-200C

University of Southern California

Los Angeles, CA 90089-2562

<http://ceng.usc.edu/~prasanna>

{mlee + liu + prasanna}@halcyon.usc.edu

## Abstract

Recently, High Performance Computing (HPC) platforms have been employed to realize many computationally demanding applications in signal and image processing. These applications require real-time performance constraints to be met. These constraints include latency as well as throughput. In order to meet these performance requirements, efficient parallel algorithms are needed. These algorithms must be engineered to exploit the computational characteristics of such applications.

In this paper, we present a methodology for mapping a class of adaptive signal processing applications onto HPC platforms such that the throughput performance is optimized. We first define a new task model using the salient computational characteristics of a class of adaptive signal processing applications. Based on this task model, we propose a new execution model. In the earlier linear pipelined execution model, the task mapping choices were restricted. The new model permits flexible task mapping choices, leading to improved throughput performance compared with the previous model. Using the new model, a three step task mapping methodology is developed. It consists of 1) a data remapping step, 2) a coarse resource allocation step, and 3) a fine performance tuning step. The methodology is demonstrated by designing parallel algorithms for modern Radar and Sonar signal processing applications. These are implemented on IBM SP2 and SGI/Cray T3E, state-of-the-art HPC platforms, to show the effectiveness of our approach. Experimental results show significant performance improvement over those obtained by previous approaches. Our code is written using C and the Message Passing Interface (MPI). Thus, it is portable across various HPC platforms.

---

\*Work supported in part by the US Defense Advanced Research Projects Agency (DARPA) Embeddable Systems Program under contract no. DABT63-95-C-0092 monitored by Fort Hauchuca.

<sup>†</sup>Wenheng Liu is currently with Electrical and Computer Engineering, California State University, Los Angeles, 5151 State University Drive, Los Angeles, CA 90032, [cliu@calstatela.edu](mailto:cliu@calstatela.edu).

# 1 Introduction

Recently, High Performance Computing (HPC) platforms have been employed to realize many computationally demanding applications in signal and image processing. Many computationally intensive algorithms have been proposed for these applications. For example, in Radar signal processing, Space-Time Adaptive Processing (STAP) techniques have been developed for the next generation radar systems which will be required to provide longer range detection of increasingly smaller targets [35]. It employs adaptive array processing techniques and simultaneously combines the signals from spatial domain (collected by an array of sensor elements) and temporal domain. Such techniques have been shown to improve the accuracy and reliability of target detection in the presence of jamming sources and environmental clutter.

Such Adaptive Signal Processing (ASP) applications are typically composed of a sequence of computation stages. Each stage consists of a number of identical tasks (i.e., FFT's, QR decompositions, etc.). Each stage repeatedly receives its input from the previous stage, performs computations, and sends its output to the next stage. The first stage receives the external input data (usually a 2-D or a 3-D data from the sensors) while the last stage produces the results (ex. Doppler bins, beam-patterns, etc.) as output. The data access pattern of the stages is usually different from one another. Hard real-time performance constraints are to be met in these applications. These include latency (i.e., the time for processing one input data set) and/or throughput (i.e., the number of outputs per unit time).

Based on the computational complexity and real-time performance constraints, ASP applications demand sustained performance in the range of 1 GFlops/sec to 50 TFlops/sec. Therefore, parallel computing platforms are needed. State-of-the-art HPC platforms are integrated using commercial off-the-shelf components. Typically, these platforms consist of powerful compute nodes, memory modules, I/O devices, and high speed interconnects [5]. These systems offer programmability, system scalability, and design flexibility. Recently, many systems have been designed for ASP applications by using HPC technology. Such ASP systems process signals from an array of sensor elements on-the-fly in an embedded environment such as on an air-borne or a sea-borne vehicle. Therefore, these systems must be designed to meet physical constraints such as size and weight, and meet constraints on power.

In using HPC platforms for parallelizing ASP applications, algorithmic techniques are needed for realizing high performance. These techniques must be developed so that the available computing power of a HPC platform can be effectively utilized in performing the application. In this paper, we address the problem of optimizing the throughput performance of an ASP application on a HPC platform.

Traditional HPC applications which typically employ supercomputers for large scale simulations have not paid much attention to optimizing the throughput performance. However, in ASP applications, a sequence of input data is continuously received. The outputs must be produced to keep pace with the data input rate. Therefore, throughput performance is a key performance measure in these applications.

Previously, a number of researchers have addressed parallelizing signal and image processing applications. In [13], key techniques and issues in parallelizing applications on HPC platforms are addressed. Issues in mapping image processing and vision applications onto HPC platforms are addressed in [3, 9, 23, 24, 36]. Parallelizing irregular problems arising in vision are addressed in [7, 25]. Parallel architectures for image processing and vision are discussed in [2, 21, 37].

In [27, 28], the throughput optimization problem for sensor-based applications is addressed. The application is represented as a sequence of computation stages. They used a dynamic programming approach to assign processors to the computation stages. A linear pipelined execution model was used for the mapping. In this model, each stage is mapped onto a disjoint set of processors. The processors assigned to a stage receive their input from the processors assigned to the previous stage, perform the computations, and send their output to the processors assigned to the next stage. Clustering and replication of stages are also considered. However, the mapping choices allowed in the linear execution model are restricted. As shown in this paper, previous solutions obtained under the linear execution model can be improved. Choudhary et. al. [6] also considered the problem of optimal processor mapping for signal and image processing applications. They considered the case when a sequence of data sets is processed by a collection of stages and the inter-stage data dependence forms a series-parallel partial order. They also assume that each stage is mapped onto a disjoint set of processors. Their solution applied to the problem considered in this paper is similar to the one in [27, 28]. However, clustering and replication of stages are not considered in [6]. Also, the modeling of communication costs in [6] is different from that in [27, 28].

We approach the throughput optimization problem from a task mapping perspective. We first define a task model for ASP applications by capturing their salient computational features. The task model exploits the independent activities among the tasks in a stage. Furthermore, to efficiently exploit the state-of-the-art HPC platforms (which consist of powerful compute nodes), coarse grain (task level) parallelism is considered in our model. We then define a new execution model. In the linear execution model (that has been used in the earlier approaches), a disjoint set of processors is assigned to the tasks in each stage. The one-to-one mapping between the stages and sets of processors offers restricted

mapping choices. We propose a new execution model which can relax this restriction. It leads to higher throughput performance compared with the throughput obtained using the linear execution model. A novel *stage partitioning* technique is used to realize the new execution model by exploiting the independency of tasks in each stage.

Based on the new model, we propose a three-step task mapping methodology. Step 1 performs *Data Remapping*. This step considers the data access pattern of each stage. If adjacent stages have different data access patterns, data remapping is performed to reduce the communication cost. Step 2 performs *Coarse Resource Allocation*. This step allocates the available processors to the computation stages according to their computational complexity. A linear pipeline is formed in this step. The data remapping algorithms needed between adjacent stages (as specified in Step 1) are inserted. Inter-stage communication cost is computed based on the inserted data remapping algorithms. Step 3 performs *Fine Performance Tuning*. This step realizes the new execution model by reassigning processors, partitioning stages, and redistributing the tasks between adjacent stages. The inter-stage communication algorithms are also modified accordingly. A heuristic algorithm is used in this step to reduce the period of the pipeline in an iterative manner.

The rest of the paper is organized as follows: Section 2 introduces background information including computational characteristics of ASP applications, characteristics of the state-of-the-art HPC platforms, our task model for these applications, and the key issues in designing and engineering parallel algorithms for the ASP applications considered in this paper. Section 3 summarizes related previous approaches. A new execution model is introduced in Section 4. Using the new model, a three step task mapping methodology is presented in Section 5. In Section 6, parallel algorithms are designed for Radar and Sonar signal processing applications using our methodology. These algorithms have been implemented on IBM SP2 and SGI/Cray T3E. Our code is written using C, LAPACK (a standard linear algebra library), and the Message Passing Interface (MPI). Therefore, our code is portable across various HPC platforms. Experimental results show the effectiveness of our approach in optimizing the throughput performance compared with the previous approaches. Section 7 concludes the paper.

## 2 Background

In this section, we describe the characteristics of adaptive signal processing applications by illustrating examples from Radar and Sonar signal processing. Then we describe the computational and communication characteristics of the state-of-the-art HPC platforms. A task model is derived for adaptive signal

processing applications based on their salient computational features and by considering the characteristics of HPC platforms. Finally, we discuss the key issues in designing algorithms for optimizing the throughput performance.

## 2.1 Characteristics of a Class of Adaptive Signal Processing Applications

Adaptive Signal Processing (ASP) is useful whenever there is a requirement to process signals that result from operation in an environment of unknown statistics [10]. ASP in such a case, usually provides a significant improvement in performance compared to conventional signal processing techniques. In a typical sensor-based ASP application, the signals received from an array of sensors are linearly combined with adaptive weight matrices (known as amplitude shading), and then the components are summed to the particular signal-noise fields impinging on the sensor array. These adaptive weights are derived dynamically based on the statistics of the received signals. These weight matrices are used to optimize the array response in terms of cancelling the multiple side-lobe, maximizing signal to noise ratio, among others. In modern Radar and Sonar applications, ASP is usually used to achieve high bearing resolution and high signal-to-noise ratio. In the following, we describe Radar and Sonar signal processing as example ASP applications to illustrate their computational features.

### 1. Space-Time Adaptive Processing (STAP):

STAP techniques are being developed for the next generation radar systems which will be required to provide longer range detection of increasingly smaller targets [35]. Such techniques simultaneously combine the signals received on multiple elements of an antenna array (the spatial domain) and from multiple pulse repetition periods (the temporal domain) in a time interval called Coherent Processing Interval (CPI). STAP offers the potential to improve the radar performance in several areas such as 1) improving low-velocity target detection, 2) detecting small targets that might otherwise be obscured, 3) detection in combined clutter and jamming environments, and 4) robustness against system errors and capability to handle various types of interference.

STAP consists of three major steps. First, a set of rules called training strategy is used to estimate the interference. The second step is weight computation. Based on the training data, an adaptive weight vector is computed. Weight computation requires the solution of a linear system of equations. This step is usually computationally intensive. Finally, in weight application step, the computed weight vector is applied to obtain the test statistic. This step involves vector dot product operations. The output of the processing, the test statistic, is a scalar for each range,

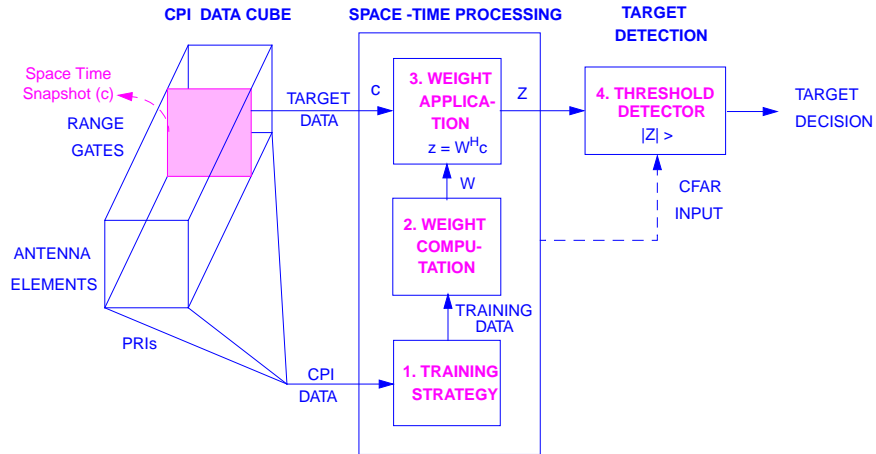


Figure 1: Overview of STAP processing

angle, and velocity. Using the output, a target presence is queried at a specified angle and Doppler. Figure 1 shows the overview of STAP processing.

A fully adaptive STAP processing computes and applies a separate adaptive weight to every element and pulse. This requires the solution of a  $MN$ -dimensional linear system of equations, where  $N$  is the number of antenna elements and  $M$  is the number of pulses transmitted in one CPI. Typically,  $N$  and  $M$  vary from ten to several hundreds.  $L$  (number of range gate) varies from 500 to 50,000. For many radar systems, the product  $MN$  ranges from several hundreds to several thousands. Many partially adaptive approaches have been developed to break down prohibitively large problems arising in fully adaptive STAP algorithms into a number of smaller, more manageable adaptive problems while achieving near-optimum performance.

## 2. Sonar Beamforming:

Beamforming is a technique which spatially filters the signals received from an array of sensors and estimates the spatial features of the sources [33]. A typical sonar beamformer has an array of sensor elements (usually a towed array or a hull-mounted array with 20 to 1000 hydrophone elements). It passively receives the acoustic propagation wave-field signals and samples the signals. In most cases, the sampling rate is less than 25 KHz. The input data (time-domain or frequency-domain) is linearly combined with a weight matrix (known as amplitude shading) to form a sonar beam for a particular direction-of-look. The requirements of high resolution and high signal-to-noise ratio in sonar systems have led to the development of adaptive sonar beamforming

techniques. Various signal models as well as noise models have been used to characterize the environmental features. Based on such models, steering vectors are derived. These vectors are used for optimal estimation of the shading for each direction-of-look. The adaptation of the entire sensor element domain is called *fully adaptive processing*. Since fully adaptive processing is computationally demanding, *partially adaptive processing* is sometimes used to reduce the dimensionality of adaptive weight matrices [33]. Numerical methods form the basis of the weight adaptation stage. Data decomposition-based least-squares algorithms are generally employed. These algorithms include QR decomposition (QRD), singular value decomposition (SVD), and Cholesky Factorization, among others. As more advanced numerical methods are used for weight adaptation and as the problem size increases, the computational complexity increases rapidly. Figure 2 shows the overview of adaptive sonar beamforming techniques. In real-time sonar beamforming, the latency constraint is in the range of few seconds while the throughput constraint is in the range of few results per second.

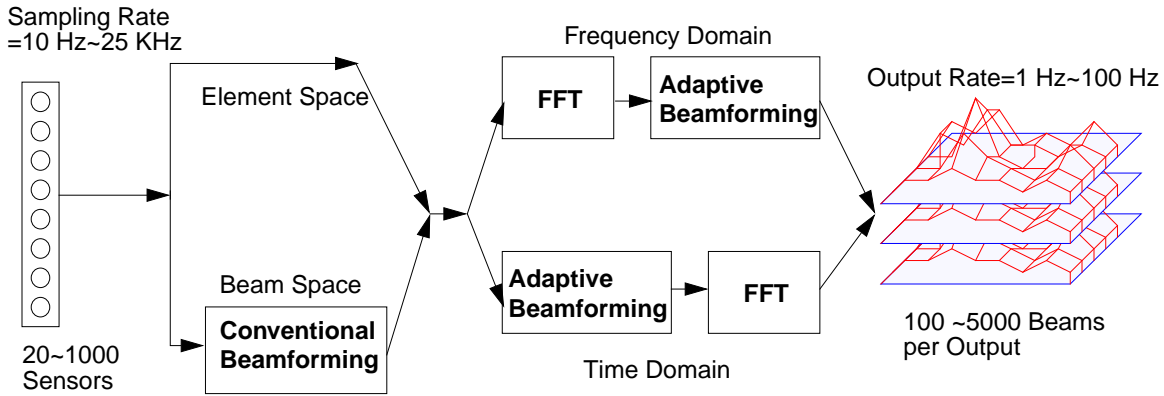


Figure 2: Adaptive sonar beamforming techniques

From the above example ASP applications, some computational characteristics (that are different from those of scientific and engineering applications) can be identified. These include:

- ASP applications are typically composed of a sequence of computation stages. Each stage repeatedly receives its input from the previous stage, performs computations, and sends its output to the next stage. A large number of identical tasks are performed in each stage. For example, each task can be a FFT or a QRD. Each task executes on its own portion of the input data. Thus, tasks in each stage are independent and can be executed in parallel. The first stage receives the

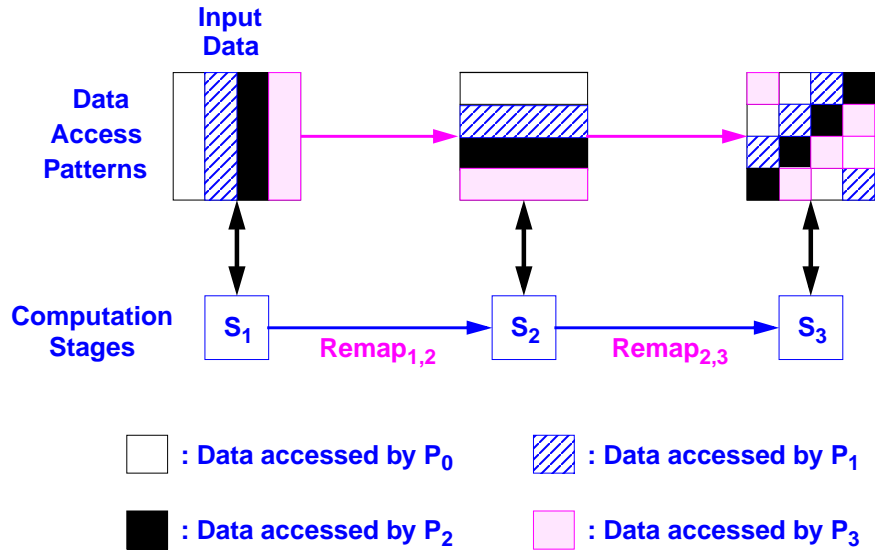


Figure 3: An illustration of data remapping

external input data (usually a 2-D or a 3-D data from the sensors) while the last stage produces the results (ex. Doppler bins, beam-patterns, etc.) as output.

- The data access pattern of each stage changes as the computation proceeds. For example, in Figure 3, processors access the two dimensional input data along columns in Stage 1, along rows in Stage 2, and along diagonals in Stage 3. Thus, initially, the matrix must be distributed such that each processor has a number of columns of the matrix. By data remapping ( $Remap_{1,2}$  and  $Remap_{2,3}$  operations), each processor can have all the necessary data to perform the tasks in each stage. Without data remapping, the data needed to perform a task is located in many processors. This incurs a lot of communication. Thus, hinders scalable performance.
- Real time performance constraints are usually imposed on ASP applications. These constraints include *latency* (i.e., response time) and/or *throughput* (i.e., number of results per unit time). The latency constraint specifies the time interval in which an input data set is processed by all the computation stages. The throughput constraint is set so that the processing can keep pace with the data input rate.

An example ASP application is illustrated in Figure 4. A Real-Time STAP ( $RT\_STAP$ ) benchmark suite has been developed by the MITRE Corporation [32]. A range of benchmarks have been defined based on the computational complexity. Figure 4 shows the most computationally demanding case. It illustrates the computation stages, the computational kernels involved in each stage, the data access

pattern of each stage, and the latency and throughput performance constraints. The computational complexity of each task and the number of parallel tasks in each stage are also shown. Different stages have different degrees of parallelism and computational complexities. The sequential execution time for each stage is measured on a Sun UltraSPARC (143 MHz, 128 Mbytes of main memory). GNU C-compiler (gcc) was used with level-2 optimization (-O2)<sup>1</sup>. The sequential execution time for each task was obtained by dividing the sequential execution time of each stage by the number of parallel tasks.

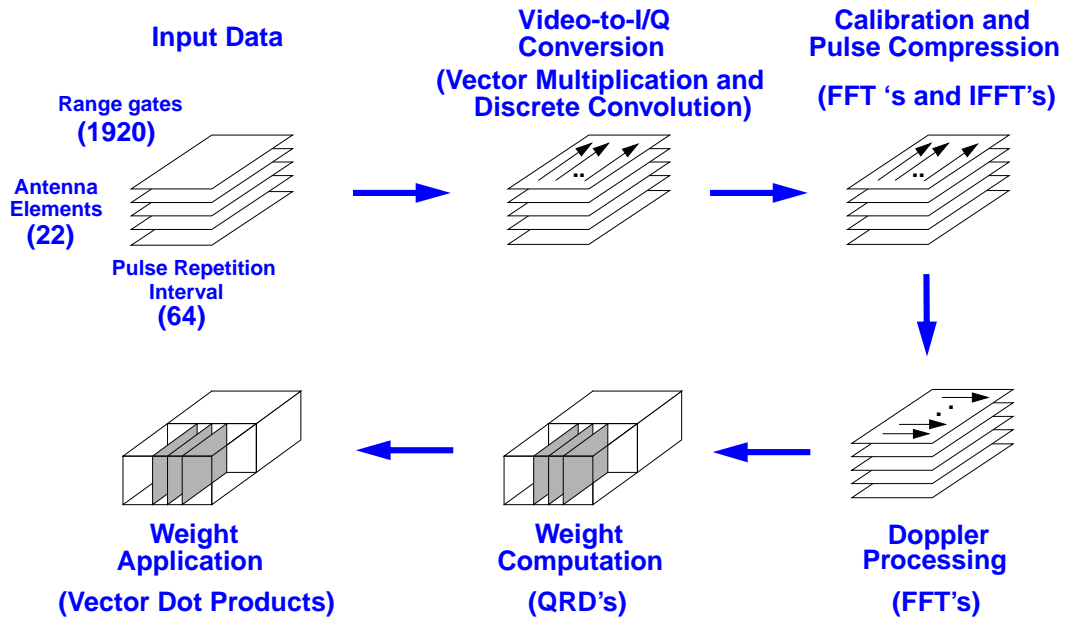
## 2.2 Characteristics of Target Architectures

The target architectures for our task mapping is HPC platforms with explicit message passing. Most of the state-of-the-art HPC platforms are built using commercial off-the-shelf (COTS) components. These platforms, typically, consist of three main components: 1) powerful compute nodes and memory modules, 2) I/O devices, and 3) a high speed interconnect [5]. Each of the compute nodes and the I/O devices are coupled to the interconnection network using a network interface. Examples of such HPC platforms include IBM SP2, SGI/Cray T3E, and Intel Paragon. The high computational power, ready availability, and cost effectiveness of these systems have led to their use in parallelizing signal and image processing applications.

In these machines, the overheads for performing communication at the user level are very high compared with the computational power of processors. The total communication time for sending a message consisting of  $m$  bytes from one processor to another can be modeled as  $T_s + m \times \tau_d$ .  $T_s$  is the *start-up time* to initiate a communication between the sender and the receiver.  $\tau_d$  is the *unit transmission time per byte of data*. In a typical HPC platform,  $T_s$  is much larger than  $\tau_d$ . Table 1 shows the peak performance of a single node, observed start-up time, and transmission time per byte for message passing on IBM SP2 and SGI/Cray T3E using the Message Passing Interface (MPI), a standard library for message passing. (Since the MPI is a standard library, a parallel code developed using MPI is portable across various HPC platforms.) For example, for IBM SP2, the values of  $T_s$  and  $\tau_d$  are 40  $\mu$ sec and 9.56 nsec per byte respectively. It is important to reduce the number of communication steps to minimize the overall communication time. Therefore, these machines are suitable for coarse grain parallel computations.

---

<sup>1</sup>The performance of the code generated using -O2 is “good”. Optimizations such as function inlining, delayed branches, instruction scheduling, among others, are performed when -O2 option is used. Further optimizations beyond -O2 (-O3 which is the highest level of optimization) results in minor performance improvement. In fact, the observed improvement in total sequential execution time was less than 1 second, which is less than 2% of the total sequential execution time (53 Seconds) (see Figure 4).



Performance Constraints:  $T_{latency} = 161.25 \text{ msec}$   
 $T_{period} = 32.25 \text{ msec}$

Computation Stages	Computational Complexity (MFlops)	Number of Parallel Tasks	Computational Complexity / Task (MFlops)	Sequential Execution Time (sec)	Sequential Execution Time / Task (msec)
(i) Video-to-I/Q Conversion	79.0	1,408	0.056	4.81	3.42
(ii) Calibration and Pulse Compression	93.6	1,408	0.067	4.77	3.39
<b>Preprocessing Total</b>	<b>172.6</b>			<b>9.58</b>	
(iii) Doppler Processing	21.8	10,560	0.002	1.67	0.16
(iv) Weight Computation	1083.2	128	8.463	40.82	318.91
(v) Weight Application	16.3	128	0.127	0.93	7.27
<b>Adaptive Processing Total</b>	<b>1121.3</b>			<b>43.42</b>	
<b>Total</b>	<b>1293.9</b>			<b>53.00</b>	

Figure 4: Computational characteristics and sequential execution time of a STAP benchmark

Machine	Peak Performance of a Single Node	Start-up Time ( $\mu$ sec)	Transmission Time per Byte (nsec/byte)
IBM SP2 (P2SC)	640 Mflops	40	9.56
Cray T3E (T3E - 1200)	1200 Mflops	16	5.45

Table 1: Computation and communication performance of some HPC platforms

As exemplified in Figure 4, the typical problem size of an individual task in ASP applications is small. For example, each QRD – the most computationally intensive task in RT-STAP – decomposes a (complex) matrix of size  $240 \times 66$ . Parallelizing such a QRD results in a large number of communication steps. Invoking a communication step takes tens of  $\mu$ seconds as shown in Table 1. However, for example, it takes 318.92 msec to perform each QRD on a single processor UltraSPARC. Thus, the communication overhead in parallelizing each QRD can be high compared with the computation time. This leads to poor utilization of processors. Therefore, exploiting parallelism finer than the level of each task is not desirable for ASP applications.

### 2.3 A Task Model for Signal Processing Applications

Based on the computational characteristics of the ASP applications and the features of HPC platforms, we define a *task model* for these applications. As described in Section 2.1, ASP applications are composed of a sequence of computation stages. Each stage consists of a number of identical tasks that can be executed independently. The data access pattern of the stages is usually different from one another. We exploit only coarse grain (task level) parallelism in each stage and consider possible data remapping between adjacent stages.

Our task model is defined as a set of pairs  $(n_i, t_i)$ , where  $1 \leq i \leq S$ .  $n_i$  is the number of tasks in *Stage*  $i$ , and  $t_i$  is the sequential execution time of each task in *Stage*  $i$ .  $n_i$  and  $t_i$  may vary from stage to stage. Figure 5 illustrates our task model representation for the RT-STAP benchmark. For example,  $n_4 = 128$ , and  $t_4 = 318.91$  msec, where  $S_4$  is the weight computation stage involving QRD's. Note that a task in Stage  $i + 1$  can begin execution only after all tasks in Stage  $i$  are completed, where  $1 \leq i \leq S - 1$ . This assumes that there is a dependency between all the tasks in Stage  $i$  and all the tasks in Stage  $i + 1$ .

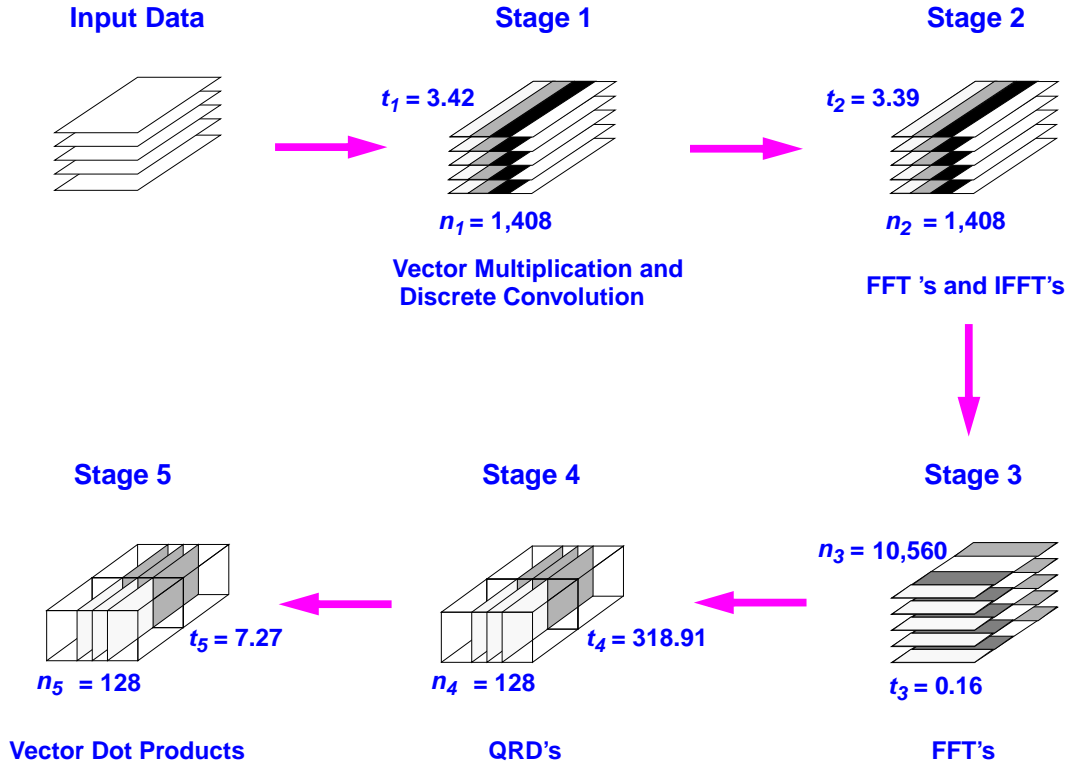


Figure 5: A task model representation of RT\_STAP benchmark. (Note:  $t_i$ 's are in msec.)

## 2.4 Key Issues in Algorithm Design and Implementation

Assume that a given ASP application is represented using our task model and a fixed number of processors ( $P$ ) are given. Our problem is to *map the ASP application onto  $P$  processors such that the throughput performance is optimized*. As described, the ASP applications considered in this paper have regular computation and communication characteristics that are known apriori. Thus, the task mapping can be performed at compile time or during parallel algorithm design phase. We perform the task mapping at the parallel algorithm design phase. Therefore, after the task mapping is performed, each processor is assigned a number of tasks (from the same stage or from different stages). Possible communication activities with other processors are also specified for each processor. Note that only coarse grain (task level) parallelism is exploited in each stage.

A trivial solution to our task mapping problem can be obtained by mapping all the stages of the application onto a single processor. Each data set from the incoming sequence of data sets is assigned to a processor in a round-robin fashion. Hence,  $P$  data sets are executed concurrently by  $P$  processors. The resulting throughput can scale linearly with the number of processors. However, this is not an

acceptable design due to the following reasons:

1. Each input data set of a ASP application is usually collected by a number of sensor elements. The collected input data must be sent to a single processor for performing all the computations. This may incur a large communication overhead, because the input data size is large and the communication steps from the sensors are serialized. Let  $T_{comm}$  denote the time for sending the input data collected by the sensor elements to a processor. Let  $T_1 = T_{comm} + T_{comp}$  be the total time for communication and computation using a single processor. Let  $T_{period}$  denote the resulting period. Then, if  $P$  processors are used,  $T_{period}$  can be computed as  $\frac{T_1}{P}$ . Note that  $T_{period} \geq T_{comm}$ . In case  $T_{comm} > \frac{T_{comp}}{P}$ , then, processors will idle, due to the late data delivery from the sensors to the processors. Therefore, large  $T_{comm}$  is undesirable. For example, consider the RT\_STAP benchmark described in Section 2.1. The data cube size is 10.8 Mbytes. Assuming an interconnection network (such as the one used in IBM SP2) in which the start-up cost is approximately 40  $\mu$ sec and the transmission rate is approximately 100 Mbytes/(sec  $\times$  processor).  $T_{comm} > 100$  msec. 100 msec is very large. Hence, the processors will be idle if we try to meet the throughput requirement. Furthermore,  $T_{comm}$  alone far exceeds the desired period (32.25 msec) specified in the benchmark suite. Therefore, this design can never satisfy the given throughput requirement.
2. An entire input data set must be stored in a single node. Besides the input data, there are some data structures generated during the execution of the application. Furthermore, memory buffer space is needed for receiving the incoming data sets. Therefore, the above solution requires large memory in each processor. A task mapping requiring a large memory space is not desirable for the class of ASP applications considered here. In addition, due to size, power, and weight constraints, if the total memory required in the application exceeds the capacity of the local main memory, then the performance will suffer due to excessive disk accesses.
3. Due to the sequential processing and due to the above overheads, the resulting latency may be too high. For example, in Figure 4, the sequential execution time for RT\_STAP benchmark running on Sun UltraSPARC is 53 seconds without considering  $T_{comm}$ . 53 seconds is too high to be acceptable for this application. The benchmark specifies a latency of 161.25 msec.

Therefore, in the remainder of this paper, the above trivial mapping is not considered.

### 3 Previous Task Mapping Approaches

Previously, several efforts have addressed the throughput optimization problem [6, 27, 28]. In this section, we describe those approaches and illustrate their limitations.

A *linear execution model* has been widely used for parallelizing ASP applications. In this model, each computation stage is mapped onto a disjoint set of processors. The output data from the processors assigned to a stage is fed forward to the processors assigned to the next stage. Thus, concurrency among the stages is exploited. Subhlok, et. al. [27, 28] considered the processor assignment problem for applications that consist of a sequence of computation stages (called data parallel tasks in their notation), using the above linear execution model. The execution time of a stage is given as a function of the number of processors assigned to that stage. Also, the communication time between two adjacent stages is modeled as a function of the number of processors assigned to these stages. A dynamic programming approach was used for optimizing the throughput (or latency) based on this model. This technique produces optimal throughput when the linear execution model is used. Techniques including *clustering* and *replication of stages* were also used. In order to reduce the high complexity of the dynamic programming algorithm, a fast greedy solution was also developed. They argue that the greedy algorithm generally produces optimal or near optimal mapping.

Choudhary et. al. [6] considered optimal processor assignment problems when a sequence of data sets is processed by a collection of computation stages (called tasks in their notation). The problem is to optimize the latency (throughput) with a given throughput (latency) requirement. They assume that the inter-stage data dependence forms a series-parallel partial order. Each stage is assumed to be mapped onto a disjoint set of processors. The execution time of a stage is given as a function of the number of processors assigned to that stage. They assumed that the communication cost can be incorporated into the computation time. They also developed a dynamic programming solution for obtaining the processor assignment. When an ASP application with a sequence of computation stages is mapped onto a given set of processors, their approach is similar to that in [27, 28]. However, they do not employ clustering or replication of stages.

The techniques in [27, 28] and [6] assume the following:

1. A disjoint set of processors is assigned to each stage.
2. All tasks in a stage can be executed on one or more processors and the execution time of each stage is a function of the number of processors assigned to the stage.

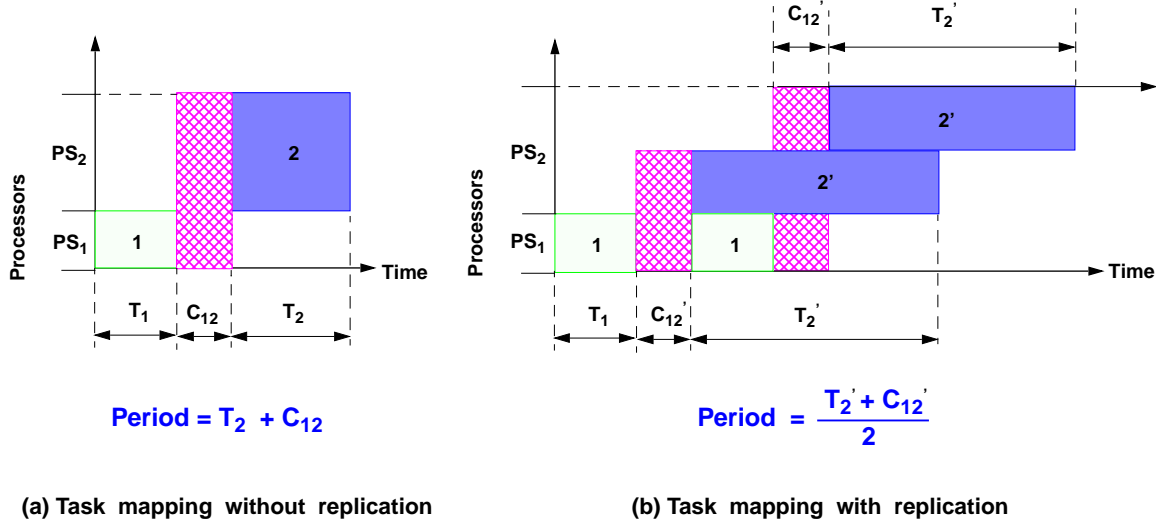


Figure 6: Hiding and amortizing communication cost using replication

Even though they were motivated by signal and image processing applications, their task model does not exploit the fact that tasks in each stage can be independent. Therefore, they do not consider scheduling of independent tasks in a stage.

Clustering and replication of stages are well known techniques for improving the throughput performance using the linear execution model. In the stage clustering technique, adjacent stages are clustered into a module. Such a module is mapped onto a set of processors. If the clustered stages have the same data layout, their data transfer between these stages is not needed. Otherwise, the processed data in a stage is fed back (remapped) to the same set of processors for executing the next stage.

Replication technique can also be used to improve the throughput performance of an application using the linear execution model. In this technique, a stage is replicated such that a sequence of incoming data sets use the replicated instances of the stage in a round robin fashion. Thus, instead of assigning a large number of processors to a stage, multiple instances of the stage are created using a small number of processors. Replication is effective in the following cases:

1. In ASP applications, the computational requirements of adjacent stages can be significantly different. Thus, the number of processors assigned to the stages may vary significantly. This may cause

a large communication overhead between these stages, because a large number of communication steps are involved. Replication can result in a smaller number of communication steps, because the number of processors assigned to those stages are more balanced. Thus, the communication cost is reduced. Furthermore, the communication cost is amortized over multiple instances of the replicated stage. Figure 6 (a) shows a task mapping without replication. Here,  $T_2 \geq T_1$ . The period for this linear pipeline is  $T_2 + C_{12}$ , where  $C_{12}$  denotes the communication cost between Stage 1 and Stage 2. Figure 6 (b) shows a task mapping with replication. The period of this pipeline is  $\frac{T_2' + C_{12}'}{2}$ , assuming  $\frac{T_2' + C_{12}'}{2} \geq T_1 + C_{12}'$ . Note that  $C_{12}'$  can be less than or equal to  $C_{12}$ . This can happen since the number of processors used for an instance of Stage 2 is only half of that in Figure 6 (a). Thus, the number of communication steps to perform the inter-stage communication is reduced. Furthermore, the communication cost ( $C_{12}'$ ) for one instance is hidden and the communication cost for the other instance is amortized over two instances. Therefore, each instance pays a communication cost of  $\frac{C_{12}'}{2}$  only.

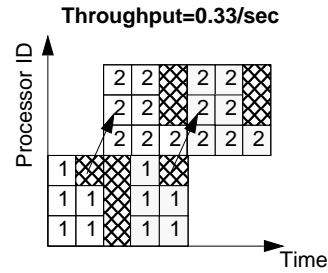
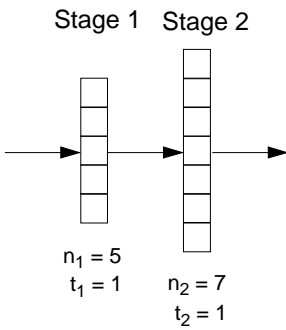
2. Some computationally demanding stages may have a low degree of parallelism, i.e., the speed-up curve is sub-linear. In such a situation, the processors assigned to that stage will not be utilized efficiently. Replication can be used such that each data set is processed by a subset of these processors. This can improve the processor utilization.

For a given number of processors, replication can increase the system throughput, however, it results in a larger latency compared with the case when replication is not used.

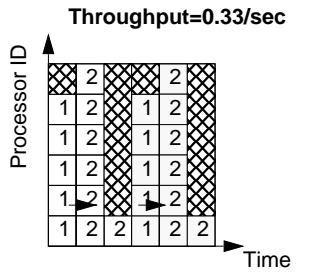
Figure 7 illustrates various task mappings using the linear execution model with and without clustering and replication of stages. In this example, a two-stage application is mapped onto 6 processors, where  $n_1 = 5$ ,  $n_2 = 7$  and  $t_1 = 1$  second,  $t_2 = 1$  second. For the sake of simplicity, we assume that the communication cost between the adjacent stages to be zero. As discussed in Section 2.4, the solution that maps all the 12 tasks in one initiation onto a single processor is not considered. Figure 7 (a) shows the mapping using the linear execution model without clustering and replication. Such a mapping is obtained by the techniques proposed in [6]. Figure 7 (b) and (c) show the mapping with clustering or replication. Both of them result in a throughput of 0.33/sec. Fig 7 (d) shows a mapping with clustering and replication. It results in a throughput of 0.43/sec. These mappings are obtained by the techniques in [27, 28]. However, the optimal throughput that can be achieved for this application is  $\frac{P}{n_1 \times t_1 + n_2 \times t_2} = \frac{6}{5+7} = 0.5/\text{sec}$ . We will show in sections 4 and 5 that we can improve their solutions and achieve the optimal throughput for this example.

## Task Mappings

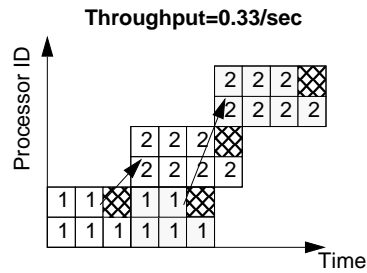
### Given Application



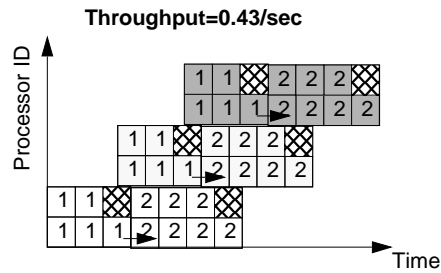
(a) Linear pipeline



(b) Linear pipeline (clustering)



(c) Linear pipeline (replication)



(d) Linear pipeline (clustering and replication)

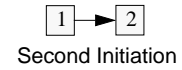
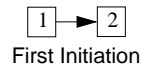


Figure 7: Various task mappings using the linear execution model with clustering and replication

## 4 An Execution Model

In this section, we propose a new execution model for mapping an ASP application onto a HPC platform. The proposed execution model is based on our task model defined in Section 2.3. The execution model is realized using a technique called *stage partitioning*. The benefits and the overheads in implementing the new execution model are also discussed. Based on this new model, an optimal task mapping algorithm using dynamic programming is derived.

### 4.1 Proposed Execution Model

As discussed in Section 3, the approach using the linear execution model may result in inefficient processor utilization due to unbalanced workload. Thus, the throughput of the resulting pipeline design may not be optimal. By clustering and replicating stages, the workload unbalance can be mitigated to some extent. However, as shown in the example in Section 3, these two techniques are not always effective in optimizing the throughput.

Our new execution model is defined to exploit the independent activities among the tasks of a computation stage. Note that, each task is mapped onto exactly one processor in our execution model. Therefore, only task level (coarse-grain) parallelism is exploited. In the linear execution model, each computation stage is mapped onto a disjoint set of processors. Therefore, there is a one-to-one mapping between the stages and the sets of processors. Our execution model, on the other hand, allows the processor sets for adjacent stages to overlap. That is, a subset of tasks of a stage (or the entire stage) can be mapped onto the processor set assigned to one of its adjacent stages. This technique relaxes the restricted choices for task mapping allowed in the linear execution model.

To realize the new execution model, we propose a technique called *stage partitioning*. Using this technique, a computation stage can be *partitioned* into disjoint subsets. Each subset consists of a number of tasks from that stage. Unlike the task mappings considered in the linear execution model, such a subset of tasks can be clustered with a subset of its adjacent stage (i.e., tasks from Stage  $i$  can be clustered with tasks in Stage  $i - 1$  and/or Stage  $i + 1$ ). The resulting clusters are mapped onto disjoint sets of processors.

We place some restrictions on partitioning stages. A stage (e.g., Stage  $i$ ) can be partitioned at most two times, thus, partitioned into at most three subsets (e.g.,  $subset_{i,1}$ ,  $subset_{i,2}$ ,  $subset_{i,3}$ ).  $subset_{i,1}$  is clustered with Stage  $i - 1$  and  $subset_{i,3}$  is clustered with Stage  $i + 1$ .  $subset_{i,2}$  is a separate cluster. In this case, Stage  $i - 1$  (Stage  $i + 1$ ) can be partitioned at most once, thus, partitioned into at most two

subsets (e.g.,  $subset_{i-1,1}$  ( $subset_{i+1,1}$ ),  $subset_{i-1,2}$  ( $subset_{i+1,2}$ )).  $subset_{i-1,2}$  ( $subset_{i+1,1}$ ) is clustered with a subset in Stage  $i$ , that is,  $subset_{i,1}$  ( $subset_{i,3}$ ).  $subset_{i-1,1}$  ( $subset_{i+1,2}$ ) can be clustered with Stage  $i - 2$  (Stage  $i + 2$ ). Therefore, two adjacent stages can be partitioned into at most five subsets and clustered into at most four consecutive clusters.

In Figure 8, the linear execution model and our new execution model are illustrated using a two-stage example. A task mapping using the linear execution model is shown in Figure 8 (a). The system throughput is determined by Stage 2, since it has the longest execution time ( $T_2$ ).  $T_2$  includes the computation time of Stage 2 and the communication time between Stage 1 and Stage 2. Using the new execution model, the throughput can be improved by stage partitioning. In this example, Stage 2 is partitioned into two subsets of tasks: Stage 2' and Stage 2''. Some tasks in Stage 2 are associated with Stage 2' and the remaining tasks are associated with Stage 2''. When these tasks are mapped onto processors, tasks in Stage 2'' are clustered with Stage 1 and are mapped onto the same processor set,  $PS_1'$  (see Figure 8 (b)). This results in a shorter period. Note that the processors can also be reassigned during stage partitioning so that  $PS_1'$  ( $PS_2'$ ) can be different from  $PS_1$  ( $PS_2$ ). The reassignment of the processors offers more flexible choices in task mapping.

Our model ensures that the precedence between successive stages is satisfied. Thus, no task in Stage  $i$  (Stage 2 in this example) can be initiated until all the tasks in Stage  $i - 1$  (Stage 1 in this example) have completed their execution. Stage 2'' is scheduled immediately following the completion of Stage 1. This task scheduling is shown in Figure 8 (b). Similarly, if Stage  $i$  is partitioned such that a subset of tasks is mapped onto the processor set assigned to Stage  $i + 1$ , this subset of tasks is scheduled immediately prior to the execution of Stage  $i + 1$ . Stage partitioning is not considered in the previous approaches [27, 6]. In fact, their task models are not amenable to stage partitioning, whereas our task model facilitates stage partitioning.

## 4.2 Benefits and Overheads of the Proposed Execution Model

To illustrate the effectiveness of the new model using the stage partitioning technique, compared with the linear model with and without clustering and replication of stages, consider the same example used in Section 3. In this example,  $n_1 = 5$ ,  $n_2 = 7$  and  $t_1 = 1$  second,  $t_2 = 1$  second, and  $P = 6$ . Based on the linear execution model using clustering and replication techniques, the “optimal” throughput achieved is 0.43/sec (as shown in Figure 7 (d) and in Figure 9 (b)). Using our new model, the throughput can be further improved. In Figure 9 (c), Stage 2 is partitioned. Five of the tasks in Stage 2 are mapped

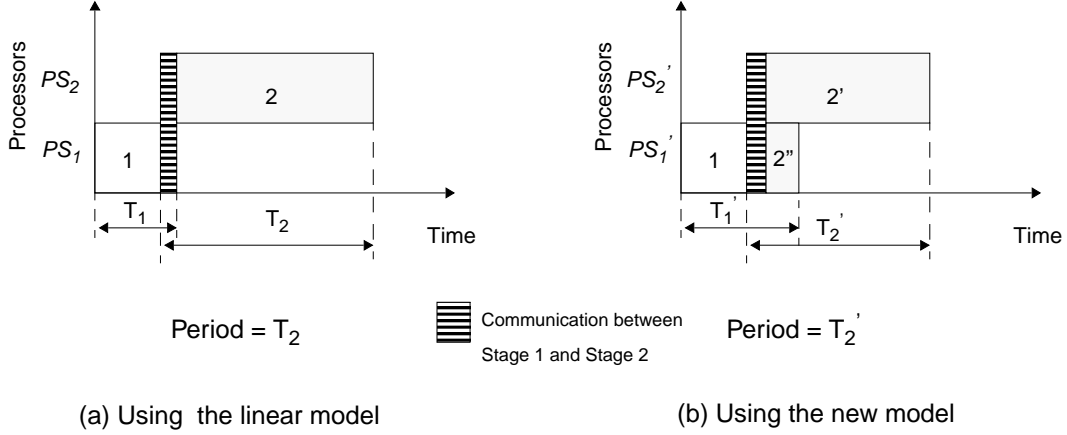
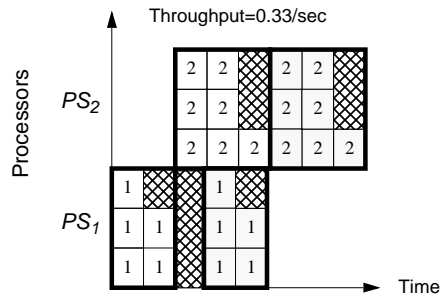
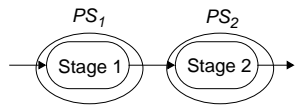


Figure 8: An illustrative example showing processor mappings using the linear execution model and the proposed model

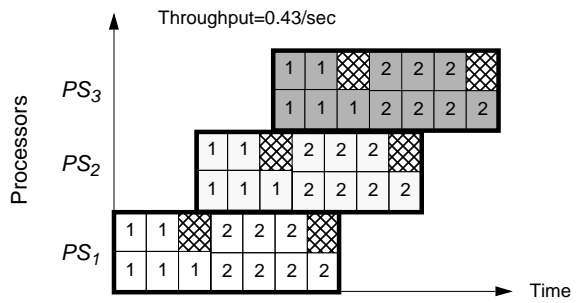
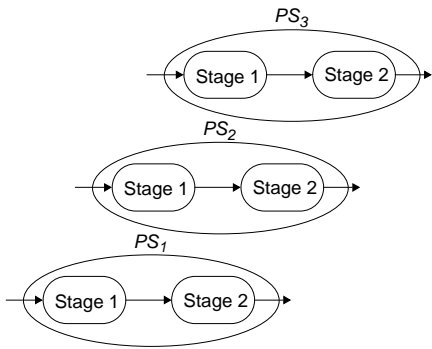
onto the 5 processors assigned to Stage 1. The remaining two tasks in Stage 2 are performed on the last processor. In this mapping, there is no unused computing power. The throughput increases to 0.5/sec. Note that, the upper bound on the throughput of this application is  $\frac{P}{n_1 \times t_1 + n_2 \times t_2} = 0.5/\text{sec}$ . Thus, for this example, our new execution model results in the optimal throughput performance.

Figure 10 shows the general scenario of pipelined execution based on our execution model. There are five clusters, denoted  $Cluster_1$  through  $Cluster_5$ . A cluster may consist of tasks from more than one stage. Each cluster is mapped onto a processor set. It can be observed that clustering (as used in earlier approaches) is implicitly considered in our approach. However, clustering in our approach is different from the one used in the linear execution model. In that model, stages are clustered and mapped onto the same set of processors. In our model, a stage can be first partitioned into subsets. Then, the subsets are clustered and are mapped onto the processors. Therefore, the clustering in our approach is more general than the clustering considered in the linear execution model. Furthermore, replication is also considered in our model. This will be addressed in our task mapping methodology in Section 5.

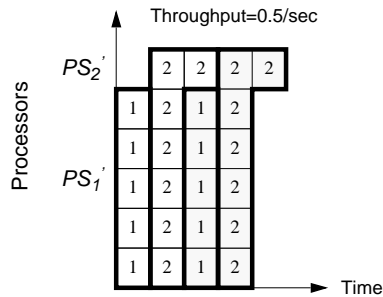
Inter-stage communication cost can be an overhead in implementing the designs based on our model. The linear execution model generates simple communication activities. A set of processors assigned to a stage communicates with the sets of processors assigned to its adjacent stages. In the proposed model, the communication activities can be more involved. Based on our model, the tasks in two consecutive stages can be partitioned into at most four consecutive clusters (for example, see Stage 1 and Stage 2 in Figure 10 (a)). Therefore, a cluster may send its results to at most three successive clusters. Figure 10 (b) illustrates these communication activities using a directed graph representation. Compared with the



(a) Linear pipeline



(b) Linear pipeline  
(clustering and replication)



(c) General pipeline  
(stage partitioning)

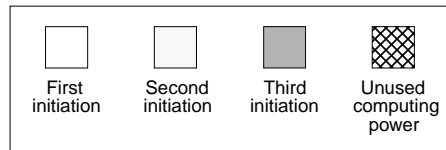


Figure 9: Illustration of task mapping using the linear execution model and the new execution model for a two-stage application

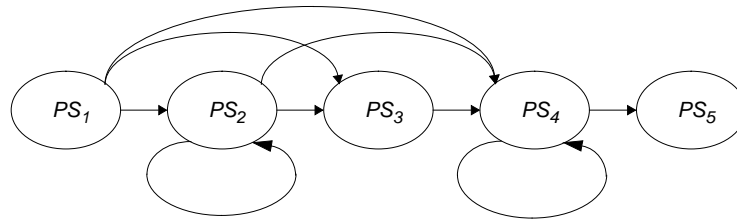
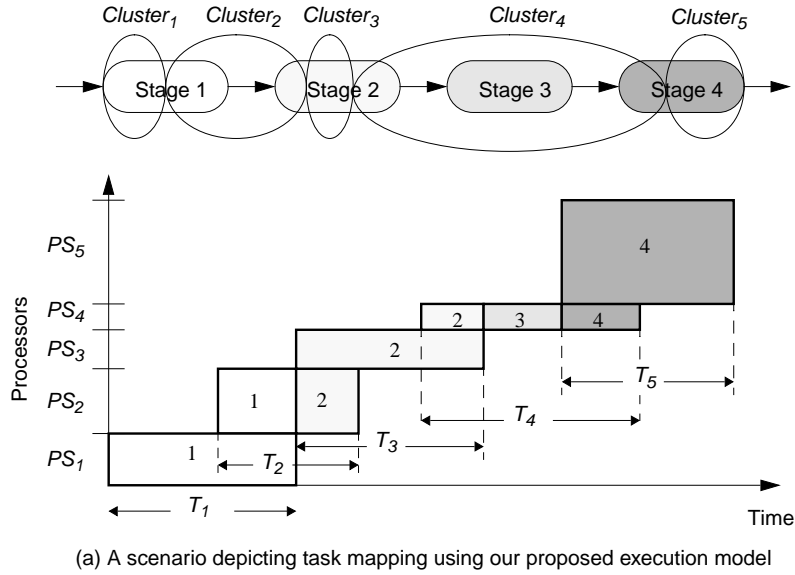


Figure 10: Task mapping and communication activities among the processor sets for a four-stage application using the proposed execution model

linear execution model, communication scheduling is needed to implement the communication activities. For instance, the processor set for  $Cluster_2$  (which consists of some tasks from Stage 1),  $PS_2$ , needs to send its results to the processor set for  $Cluster_4$  (which consists of some tasks from Stage 2),  $PS_4$ . However, the execution of  $Cluster_2$  is completed before the execution of  $Cluster_4$  starts as shown in Figure 10 (a). In order for  $PS_2$  to communicate with  $PS_4$ ,  $PS_2$  needs to synchronize with  $PS_4$ . This incurs additional idle time for  $PS_2$  and/or  $PS_4$ . In this case, asynchronous communication can be useful. By using asynchronous communication primitives,  $PS_2$ , after completing execution of the tasks from Stage 1, can send the resulting data to  $Cluster_4$ . Then, it can immediately resume the computation corresponding to the tasks from Stage 2. Hence,  $PS_2$  does not need to wait for the initiation of  $Cluster_4$  on  $PS_4$  to receive the data sent from  $PS_2$ .

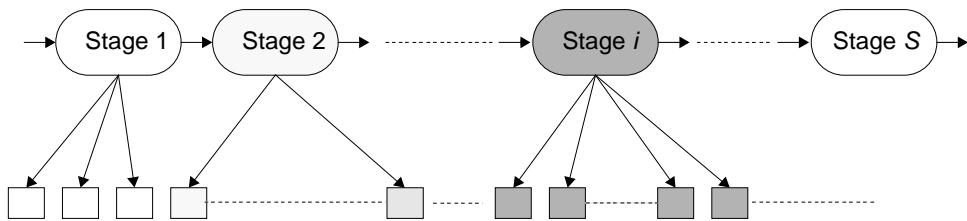
Asynchronous communication may result in a more complex coding since the communication requires a separate command to verify the reception of messages. However, by using the asynchronous communication mode, the main processor can resume its computation immediately after a message passing. This approach is especially advantageous on modern HPC platforms such as IBM SP2 and SGI/Cray T3D/E. On such platforms, message passing operations can be partially off-loaded from the main processor by using a message passing co-processor and/or Direct Memory Access (DMA) engines to copy the data from the local memory to the interface buffer or vice versa. These further reduce the time the main processor is engaged in message passing.

On the other hand, if we use synchronous communication, the sender cluster needs to be synchronized with the receiver cluster whenever a data remapping is required. For the above example based on Figure 10, PS2 has to be interrupted when PS4 is initiated with the tasks from Stage 2 to perform the data remapping between Stages 1 and 2. The interrupt itself requires additional exception handlers. Indeed, the implementation using this approach is more costly than using the asynchronous communication approach.

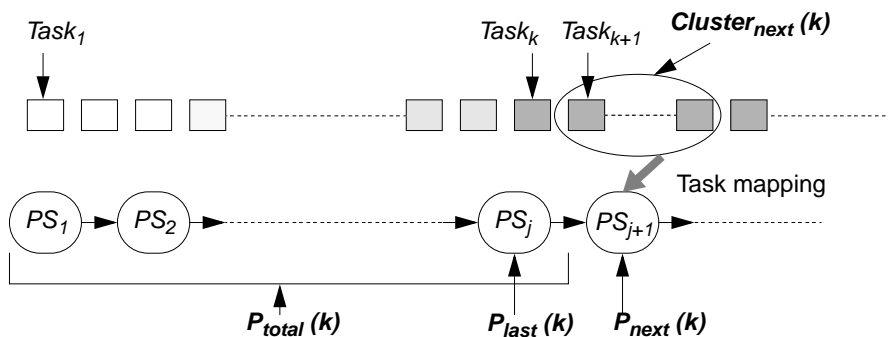
### 4.3 A Dynamic Programming Solution

In this subsection, we describe a solution based on dynamic programming for the throughput optimization problem. The inputs are an ASP application and the available number of processors. The ASP application is represented by the task model defined in Section 2.3. The output is a task mapping specified by a sequence of clusters and a number of processors assigned to each cluster. This task mapping results in optimal throughput for performing the given application. Our solution extends the one presented in [27] to suit our execution model.

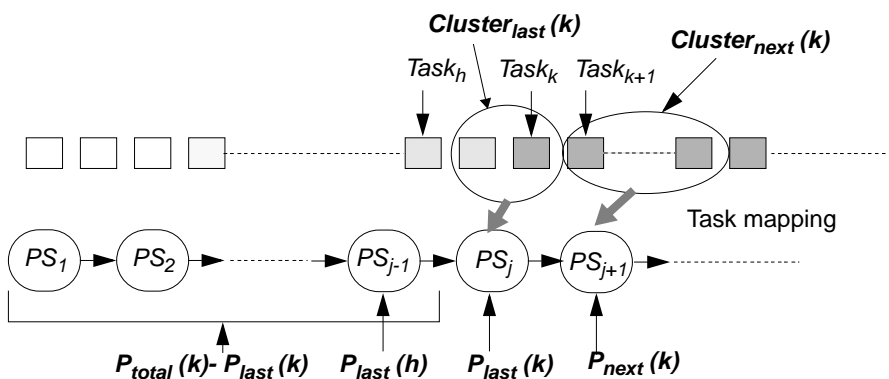
Based on our new execution model, the computation stages of the application can be partitioned into clusters by exploiting parallel activities among the tasks in a stage. Each cluster is mapped onto a disjoint set of processors. A cluster consists of a consecutive sequence of tasks. Since we use coarse-grain computation, each task in this cluster is mapped onto exactly one processor. To fully utilize the computing power, the number of tasks from each stage in a cluster must be a multiple of the number of processors assigned to that cluster. For a number of processors, the computation time to perform the tasks in a cluster is known. The communication cost between any two clusters is also known. The cost is a function of the number of tasks mapped from each stage and the number of processors assigned to each of the two clusters. Note that the communication cost between two clusters also depends on the



(a) A signal processing application expanded to form a sequence of tasks



(b) Step  $k$  of the dynamic programming formulation



(c) A subproblem considered for optimizing the solution in Step  $k$

Figure 11: Dynamic programming solution

data layout and the communication algorithms used. These issues are not considered in this approach. In the next section, the proposed heuristic algorithm considers data remapping.

As in Section 2.3, let  $S$  denote the number of computation stages. Let  $n_i$  denote the number of tasks and  $t_i$  denote the sequential execution time of each task in Stage  $i$ ,  $1 \leq i \leq S$ . Let  $P$  denote the number of available processors. For the dynamic programming formulation, the stages of the application are first expanded as shown in Figure 11 (a): all the tasks in each stage are lined up from left to right. Therefore, the application is converted into a sequence of  $K = \sum_{i=1}^S n_i$  tasks. Then, dynamic programming is used to optimize the throughput of an increasing subsequence of tasks starting from the left-most task.

To illustrate our ideas, we first assume that the communication occurs between adjacent clusters only. We also assume that the cluster that starts from  $Task_1$  (ends in  $Task_K$ ) communicates with its next (previous) cluster only. Figure 11 (b) shows the basic structure of an optimal task mapping of the tasks from  $Task_1$  to  $Task_k$ ,  $1 \leq k \leq K$ . Such a mapping can be characterized by five variables:

- $k$ : the index of  $Task_k$ .
- $Cluster_{next}(k)$ : the cluster starting from  $Task_{k+1}$  (see Figure 11.(b)<sup>2</sup>).
- $P_{next}(k)$ : the number of processors assigned to  $Cluster_{next}(k)$ .
- $P_{last}(k)$ : the number of processors assigned to the previous cluster of  $Cluster_{next}(k)$ .
- $P_{total}(k)$ : the total number of processors assigned to all the clusters upto  $Task_k$ .

Based on these parameters, the optimal throughput of the first  $k$  tasks is denoted as  $Thr_k(P_{total}(k), P_{last}(k), P_{next}(k), Cluster_{next}(k))$ . We define  $Cluster_{next}(K) = \phi$  and  $P_{next}(K) = 0$  as the boundary conditions. The optimal throughput of the application is given by the maximum of  $Thr_K(P, P_{last}(K), 0, \phi)$ ,  $1 \leq P_{last}(K) \leq P$ , assuming all the  $P$  processors are assigned. There are  $K$  steps in the dynamic programming solution. The optimal solutions of an increasing subsequence of tasks are obtained from Step 1 to Step  $K$ . In Step  $k$ , given  $P_{total}(k)$ ,  $P_{last}(k)$ ,  $P_{next}(k)$ , and  $Cluster_{next}(k)$ ,  $Thr_k(P_{total}(k), P_{last}(k), P_{next}(k), Cluster_{next}(k))$  is obtained based on the subproblems solved in the earlier steps.

---

<sup>2</sup>This parameter is required to compute the communication time between  $Cluster_{next}(k)$  and its previous cluster. This will be explained later in this section when we explain how  $Exe(Cluster_{last}(k), P_{last}(k), Cluster_{next}(k), P_{next}(k))$  is computed (last paragraph of page 26).

Figure 11 (c) shows a subproblem considered to obtain  $Thr_k(P_{total}(k), P_{last}(k), P_{next}(k), Cluster_{next}(k))$ . This subproblem is solved in Step  $h$ , where  $h < k$ . Let  $Cluster_{last}(k)$  denote the previous cluster of  $Cluster_{next}(k)$ . Let  $Exe(Cluster_{last}(k), P_{last}(k), Cluster_{next}(k), P_{next}(k))$  denote the computation and communication time for  $Cluster_{last}(k)$  when  $Cluster_{last}(k)$  is performed on  $P_{last}(k)$  processors and  $Cluster_{next}(k)$  is performed on  $P_{next}(k)$  processors. If  $h = 0$ ,  $Cluster_{last}(k)$  consists of all the tasks from  $Task_1$  to  $Task_k$ . Thus,  $Thr_k(P_{total}(k), P_{last}(k), P_{next}(k), Cluster_{next}(k))$  is given by  $\frac{1}{Exe(Cluster_{last}(k), P_{last}(k), Cluster_{next}(k), P_{next}(k))}$ . Otherwise, if  $1 \leq h < k$ , the optimal throughput is determined based on the solutions obtained in the earlier steps. As shown in Figure 11 (c), let  $P_{last}(h)$  denote the number of processors assigned to the previous cluster of  $Cluster_{last}(k)$ .  $P_{last}(h)$  can be varied from 1 to  $P_{total}(k) - P_{last}(k)$ . For a given  $h$  and  $P_{last}(h)$ , the optimal throughput of the first  $h$  tasks in Figure 11 (c) is given by  $Thr_h(P_{total}(k) - P_{last}(k), P_{last}(h), P_{last}(k), Cluster_{last}(k))$ . Given such a subproblem, the throughput of the subsequence upto  $Task_k$  is given by the minimum of  $Thr_h(P_{total}(k) - P_{last}(k), P_{last}(h), P_{last}(k), Cluster_{last}(k))$  and the throughput of  $Cluster_{last}(k)$ . The maximum throughput, over all possible  $h$  and  $P_{last}(h)$ , is  $Thr_k(P_{total}(k), P_{last}(k), P_{next}(k), Cluster_{next}(k))$ . Thus, the optimal throughput at the  $k$ -th step is given by:

- If  $h = 0$  (if  $Cluster_{last}(k)$  starts with  $Task_1$ ):

$$\frac{1}{Exe(Cluster_{last}(k), P_{last}(k), Cluster_{next}(k), P_{next}(k))}$$

- If  $h > 0$ :

$$Max_{1 \leq h < k, 1 \leq P_{last}(h) \leq P_{total}(k) - P_{last}(k)} ( Min(Thr_h(P_{total}(k) - P_{last}(k), P_{last}(h), P_{last}(k), Cluster_{last}(k)), \frac{1}{Exe(Cluster_{last}(k), P_{last}(k), Cluster_{next}(k), P_{next}(k))}) ) )$$

In this formulation,  $Exe(Cluster_{last}(k), P_{last}(k), Cluster_{next}(k), P_{next}(k))$  must be computed at the  $k$ -th step. It is the sum of the computation time of  $Cluster_{last}(k)$  and the communication time between  $Cluster_{last}(k)$  and its adjacent clusters. Since  $Cluster_{last}(k)$  is performed using  $P_{last}(k)$  processors, the computation time of  $Cluster_{last}(k)$  is known. Besides, the communication time between  $Cluster_{last}(k)$  and  $Cluster_{next}(k)$  is also known, since  $Cluster_{last}(k)$ ,  $P_{last}(k)$ ,  $Cluster_{next}(k)$ , and  $P_{next}(k)$  are all known. Similarly, the communication between  $Cluster_{last}(k)$  and its previous cluster is known in Step  $h$ . Thus,  $Exe(Cluster_{last}(k), P_{last}(k), Cluster_{next}(k), P_{next}(k))$  can be computed. Note that the cluster that starts from  $Task_1$  (ends in  $Task_K$ ) does not have a previous (next) cluster. Thus, the corresponding communication time is zero.

In Step  $k$ ,  $1 \leq k \leq K$ , solutions to  $Thr_k(P_{total}(k), P_{last}(k), P_{next}(k), Cluster_{next}(k))$  are obtained. The dynamic programming algorithm is described in the following:

For  $k = 1$  to  $K$

For  $P_{total} = 1$  to  $P$ ,  $P_{last} = 1$  to  $P_{total}$ , and  $P_{next} = 1$  to  $P - P_{total}$  and every possible  $Cluster_{next}(k)$   
 Compute  $Thr_k(P_{total}(k), P_{last}(k), P_{next}(k), Cluster_{next}(k))$ .

Now, we estimate the complexity of the dynamic programming formulation. First, we estimate the number of solutions obtained in Step  $k$ . Each of  $P_{total}(k)$ ,  $P_{last}(k)$ , and  $P_{next}(k)$  can be assigned  $O(P)$  distinct values. Besides, when  $P_{next}(k)$  processors are assigned to  $Cluster_{next}(k)$ , the number of tasks to be clustered is a multiple of  $P_{next}(k)$ . Therefore, the number of solutions obtained in step  $k$  is  $O(P^2 \times \sum_{i=1}^P \frac{K}{i}) = O(P^2 K \ln P)$ . Then, we estimate the computational complexity to obtain each  $Thr_k(P_{total}(k), P_{last}(k), P_{next}(k), Cluster_{next}(k))$ . The variable,  $P_{last}(h)$  can be varied from 1 to  $P_{total}(k) - P_{last}(k)$ , while the variable  $Cluster_{last}(k)$  can be assigned  $O(\frac{K}{P_{last}(k)}) = O(K)$  distinct values (since the number of tasks in this cluster is a multiple of  $P_{last}(k)$ ). With given  $Cluster_{last}(k)$  and  $P_{last}(h)$ , the resulting throughput is computed in constant time. Thus, the time complexity to obtain an optimal solution is  $O(PK)$ . In summary, there are  $K$  steps in the dynamic programming formulation. Each step obtains  $O(P^2 K \ln P)$  optimal solutions. The time to compute each solution is  $O(PK)$ . Thus, the total computational complexity is  $O(P^3 K^3 \ln P)$ . Note that this is an off-line procedure.

When the communication between non-adjacent clusters is considered, the computational complexity further increases. As shown in Figure 10 (b), in the extreme case, a processor set can communicate with three successive processor sets. Thus, the task mapping for four consecutive clusters should be considered in the formulation. Four additional parameters need to be introduced into  $Thr_k()$  to estimate the communication cost. They are  $P_{next+1}(k)$ ,  $Cluster_{next+1}(k)$ ,  $P_{next+2}(k)$ , and  $Cluster_{next+2}(k)$ . As we assumed in the beginning of this section, the number of tasks from each stage in a cluster must be a multiple of the number of processors assigned to that cluster to fully utilize the computing power. Thus, when  $P_{next+1}(k)$  processors are assigned to  $Cluster_{next+1}(k)$  and  $P_{next+2}(k)$  processors are assigned to  $Cluster_{next+2}(k)$ , the number of tasks to be clustered is a multiple of  $P_{next+1}(k)$  and  $P_{next+2}(k)$ , respectively. This increases the number of optimal solutions to be considered by a multiplicative factor of  $O((\sum_{i=1}^P \frac{K}{i})^2) = O(K^2 (\ln P)^2)$ . However, the time complexity to obtain each such solution remains the same. Therefore, the total complexity becomes  $O((P^3 K^3 \ln P) \times (K^2 (\ln P)^2)) = O(P^3 K^5 (\ln P)^3)$ . Even though the task mapping is performed off-line, the high complexity makes this solution unattractive. A

simple approach for reducing the complexity is to initially group a number of tasks (say  $N$  tasks) into one super-task. Thus,  $K$  can be replaced by  $K' = \frac{K}{N}$ . In the next section, we propose a simple heuristic algorithm which drastically reduces the complexity of the task mapping.

## 5 A Heuristic Task Mapping Methodology

Based on our task model and our execution model, in this section, we describe a three-step task mapping methodology. This methodology is used to maximize the throughput of an ASP application on a given number of processors of a HPC platform. The output of this methodology is a task mapping specified by a sequence of clusters and the number of processors assigned to each cluster, and the data remapping to be performed between clusters. Communication costs between clusters are considered in the throughput optimization. Techniques including data remapping, replication of stages, stage partitioning, and clustering are used in our design methodology. Figure 12 gives an overview of the methodology. Details of the methodology are given in Section 5.1. An illustrative example using our methodology is shown in Section 5.2.

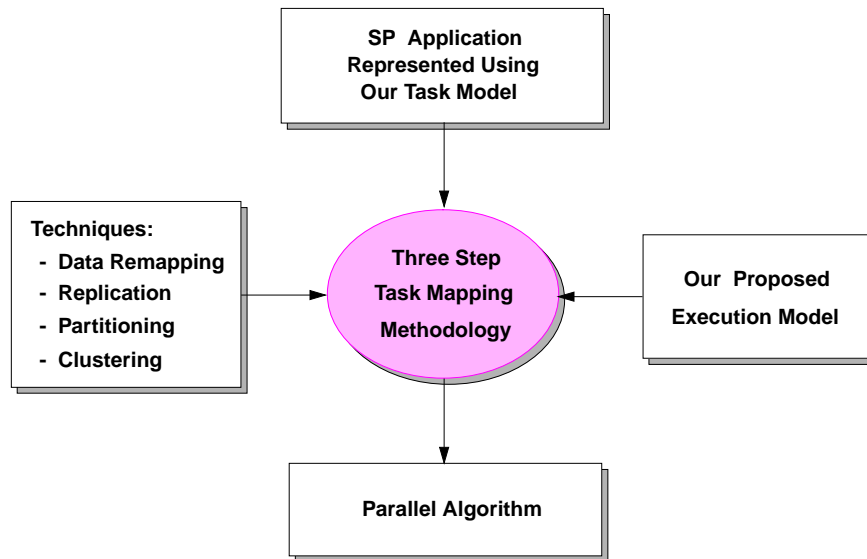


Figure 12: Overview of the task mapping methodology

### 5.1 A Three-Step Task Mapping Methodology

As described in Section 2.3, let  $S$  denote the number of computation stages. Let  $n_i$  and  $t_i$  denote the number of tasks and the sequential execution time of each task in Stage  $i$  respectively. Let  $P$  denote

the number of available processors. Based on our execution model, these stages can be partitioned into clusters. Each cluster is mapped onto a disjoint set of processors. Coarse-grain parallelism is exploited for task mapping. Thus, each task in a cluster is mapped onto exactly one processor.

The computation time to perform the tasks in a cluster using a number of processors is given. The communication time between any two processor sets to perform a given data remapping is also given. This cost depends on the number of processors assigned to the clusters, the number of tasks in the clusters, the data layouts, the data size, and the communication algorithm used to perform the data remapping. As in Section 4.2, a processor set may communicate with several other processor sets to perform data transfer among the clusters mapped onto these processor sets.

Our methodology consists of three steps. In Step 1, data remapping between stages is considered. Coarse resource allocation is performed in Step 2 based on the amount of computation to be performed in each stage. Possible replication of stages are also considered in this step. In Step 3, a fine performance tuning is performed using stage partitioning. A fast heuristic algorithm is developed for this step. The details of these steps are described in the following:

### **Step 1: Data Remapping**

In this step, the data layout of each computation stage is chosen based on its data access pattern. Data remapping is used to modify the data layout of successive stages so that each processor contains all the data needed to execute a task.

Data remapping is a crucial step in the overall mapping process. Efficient data remapping algorithms are required to reduce the communication cost by decreasing the number of start-ups and by scheduling the messages to be delivered in a conflict-free manner. A straightforward approach for data remapping is to use a *direct schedule*: data blocks are sent directly from the source nodes to the destination nodes. This approach incurs minimum data transmission cost but communication start-up cost can be high. To minimize the start-up cost, *indirect schedules* [17, 18] can be used. In this approach, data blocks are sent to their destination through intermediate “relay” nodes. At these intermediate nodes, messages destined to the same node are combined. This approach reduces the start-up cost by combining the data to be sent to the same destination.

Note that the previous approaches [6, 27] do not consider data layout optimization using data remapping in their task mapping methodologies. Therefore, the performance of the resulting implementations may be severely degraded due to remote memory accesses during the execution.

## Step 2: Coarse Resource Allocation

This step performs an initial task mapping onto the given number of processors. A coarse resource allocation is performed and a linear pipeline is generated. Thus, the computation stages are mapped onto disjoint sets of processors. To balance the workload, the number of processors assigned to Stage  $i$ ,  $1 \leq i \leq S$ , is determined in proportion to the computational complexity of that stage (i.e.,  $n_i \times t_i$ ). Let  $P_i$  denote the number of processors assigned to Stage  $i$ ,  $1 \leq i \leq S$ . We set  $P_i = \lfloor \frac{n_i \times t_i \times P}{\sum_{i=1}^S n_i \times t_i} \rfloor$ . We also define  $r_i$  to be the number of unused processors in Stage  $i$ . If  $P_i \leq n_i$ , all of the  $P_i$  processors are used and  $r_i = 0$ . The computation time of Stage  $i$  is  $\lceil \frac{n_i}{P_i} \rceil \times t_i$ . On the other hand, if  $P_i > n_i$ , Stage  $i$  is replicated  $\lfloor \frac{P_i}{n_i} \rfloor$  times. Each replicated instance of the stage is mapped onto exactly  $n_i$  processors. This results in  $r_i = P_i - \lfloor \frac{P_i}{n_i} \rfloor \times n_i$  unused processors in Stage  $i$ . If replication is used, the average computation time of Stage  $i$  is  $\frac{t_i}{\lfloor \frac{P_i}{n_i} \rfloor}$ .

Let  $P_F$  denote the number of *free* processors. After the above allocation,  $P_F = P - \sum_{i=1}^S (P_i - r_i)$ . We adopt a greedy approach to assign these free processors to the computation stages. The greedy approach is summarized as follows:

1. Compute the period of each stage.
2. Let  $SS_{max\_period}$  denote the set of bottleneck stages (stages with the largest period).
3. Find the minimum number of processors required to reduce the period of each of the stages in  $SS_{max\_period}$ . Let  $P_m$  denote the total number of processors required for these stages.
4. If  $P_F - P_m < 0$ , then Stop.
5. Assign the  $P_m$  processors to the stages in  $SS_{max\_period}$  so that the throughput of the pipeline is improved.
6.  $P_F = P_F - P_m$ . Go to Step 1.

If  $P_F > 0$  after the above assignment, the remaining free processors are assigned to an arbitrary stage with the largest period.

## Step 3: Fine Performance Tuning

In Step 3, stage partitioning is used to further improve the throughput. A bottleneck stage is defined to be the stage with the largest period. The period of a stage includes its computation time and the communication time with its previous stage and its successor stage. The communication cost between

adjacent stages is known based on the data remapping algorithm chosen in Step 1. A heuristic algorithm is used to improve the period of the pipeline in an iterative manner. In each iteration, stage partitioning is performed to balance the workload among the processors. Each iteration is summarized as follows:

- 3.1 Let  $\Delta (> 0)$  be a bound on the performance improvement per iteration.
- 3.2 Compute the set of bottleneck stage(s),  $SS_{max\_period}$ . Let  $max_{period}$  be the period of the bottleneck stage(s).
- 3.3 For each Stage  $i$ , Stage  $i \in SS_{max\_period}$ , minimize the period of the pipeline by a local optimization as follows:
  - (a) Reassign the processors assigned to Stage  $i$  and Stage  $i + 1$  and redistribute the tasks in those stages including the associated data remapping.
  - (b) Reassign the processors assigned to Stage  $i$  and Stage  $i - 1$  and redistribute the tasks in those stages including the associated data remapping.
- 3.4 Compute the set of bottleneck stage(s),  $SS'_{max\_period}$ . Let  $max'_{period}$  be the period of the bottleneck stage(s) after the above performance tuning.
- 3.5 If  $max_{period} - max'_{period} < \Delta$ , then Stop.  
Else, set  $max_{period} = max'_{period}$  and go to step 3.3.

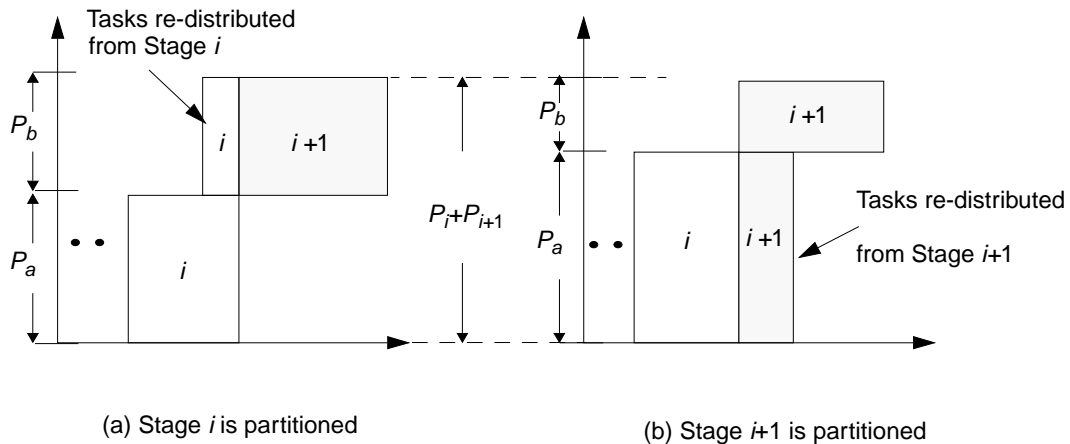


Figure 13: Examples of stage partitioning

Figure 13 shows a stage partitioning using an example. Stage partitioning between Stage  $i$  and Stage  $i + 1$  is shown, where Stage  $i$  is the bottleneck stage. Let  $P_i$  and  $P_{i+1}$  denote the number of processors

assigned to Stage  $i$  and Stage  $i + 1$ , respectively in Step 2. These processors are reassigned. After processor reassignment, Stage  $i$  or Stage  $i + 1$  is partitioned to improve the period of the pipeline design. Figure 13 shows two cases, where Stage  $i$  and Stage  $i + 1$  are partitioned. In this figure,  $P_a$  processors are assigned to Stage  $i$  and  $P_b$  processors are assigned to Stage  $i + 1$  after processor reassignment and task redistribution. Note that  $P_a + P_b = P_i + P_{i+1}$ . The same procedure is applied between Stage  $i$  and Stage  $i - 1$ . The time to perform one iteration of Step 3.3 in the above algorithm is  $O(P)$ .

The period of the bottleneck stage(s) generated by the above heuristic is a monotonically decreasing sequence. Thus, the above heuristic terminates after a finite number of iterations. Also note that, if a stage is replicated in Step 2, only one instance of the replicated stage needs to be considered in stage partitioning. The same stage partitioning optimization is applied to the rest of the replicated instances.

In the following, the steps of our task mapping methodology are illustrated using an example. Consider a two stage application in Figure 14 (a). Stage 1 consists of  $n_1$  tasks, each having an execution time of  $t_1$ . Each task uses a one-dimensional array of  $n_2$  elements. Stage 2 consists of  $n_2$  tasks with an execution time of  $t_2$ . Each task uses a one-dimensional array of  $n_1$  elements. The first step of our methodology determines whether the data access patterns in these stages are different (See Figure 14 (b)). If so, a data remapping is performed. A data remapping, corner turn, is inserted between the stages (See Figure 14 (c)).

In the next step, a coarse resource allocation is performed. Based on the estimated execution time of each stage, processors are assigned. Stage 1 and Stage 2 are assigned  $P_1 = \lfloor \frac{n_1 \times t_1 \times P}{n_1 \times t_1 + n_2 \times t_2} \rfloor$ ,  $P_2 = \lfloor \frac{n_2 \times t_2 \times P}{n_1 \times t_1 + n_2 \times t_2} \rfloor$  processors, respectively. In this example, we have assumed that there are no free processors. With the assigned number of processors, the execution time for each stage is computed. Based on the data remapping pattern (corner turn) and the number of processors assigned to the stages ( $P_1$  and  $P_2$ , respectively), the time for data remapping ( $T_{CT}$ ) is computed. In the illustration,  $T_2 > T_1$ . Thus, the period of this pipeline is  $T_{LP} = T_2 + T_{CT}$ . (See Figure 14 (d)).

In the final step, we perform stage partitioning to further improve the throughput. Stage 2 is partitioned and a subset of tasks is mapped onto the same processor set as Stage 1 (See Figure 14 (e)). Processor reassignment is not performed. Note that the resulting design eliminates idling of processors shown in Figure 14 (d). The new execution time for Stage 2,  $T_2'$ , is smaller than  $T_2$ . Note also that the new data remapping time,  $T_{CT}'$ , is likely to be smaller than  $T_{CT}$ , because the amount of data received by each processor is reduced compared with the linear pipeline design [30]. The resulting period for this design,  $T_{SP}$ , is  $T_2' + T_{CT}'$ , assuming  $T_2' > T_1 + T_2''$ .

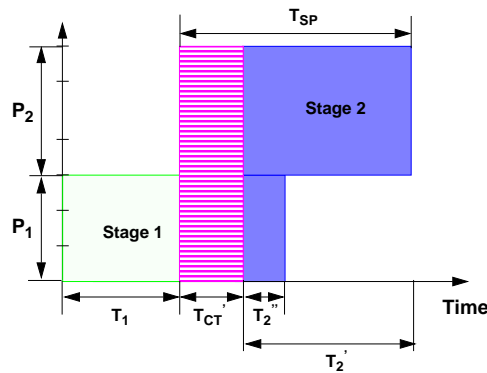
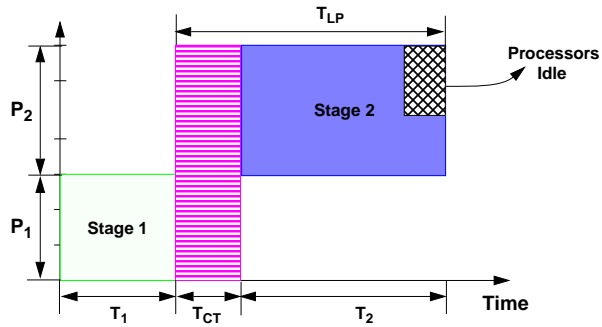
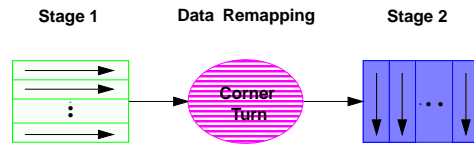
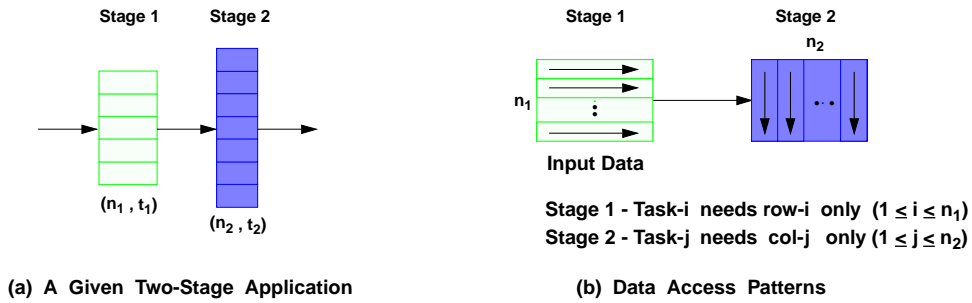


Figure 14: An illustrative example showing our methodology

## 5.2 An Illustrative Example

In the following, the effectiveness of the above mapping methodology is illustrated by using the example discussed in Section 3 and in Section 4.2. Our methodology is used to map the two-stage application onto 6 processors. Let  $n_1 = 5, n_2 = 7$ . Let  $t_1 = 1$  second and  $t_2 = 1$  second. Step 1 determines the data layout of each of the stages and the data remapping to be performed between adjacent stages. As discussed in Section 2.3, the solution that maps all the 12 tasks in one initiation onto a single processor is not considered. Based on the linear execution model with clustering or replication, as proposed in [27], the “optimal” throughput achieved is 0.43/sec (as shown in Figure 7 (d)). Using our mapping methodology, the throughput can be improved. The coarse resource allocation in Step 2 results in an initial mapping such that 2 ( $= \lfloor \frac{5 \times 6}{12} \rfloor$ ) processors are assigned to Stage 1 and 3 ( $= \lfloor \frac{7 \times 6}{12} \rfloor$ ) processors are assigned to Stage 2. Therefore, 1 ( $= 6 - (2 + 3)$ ) processor is designated as free processor. After coarse resource allocation,  $T_1 = \lceil \frac{5}{2} \rceil \times 1 = 3$  seconds and  $T_2 = \lceil \frac{7}{3} \rceil \times 1 = 3$  seconds. Assigning the free processor to either Stage 1 or Stage 2 can reduce the period of Stage 1 or Stage 2 to 2 seconds. The free processor is assigned to Stage 1, in this example. Then, fine performance tuning is performed. Figure 15 shows possible processor reassignments and task redistributions using stage partitioning. The optimal throughput is obtained when  $P_a=5$  and  $P_b=1$  as shown in Figure 15. In this case, Stage 2 is partitioned. Five of the tasks in Stage 2 are mapped onto the 5 processors assigned to Stage 1. The remaining two tasks in Stage 2 are performed on the last processor. In this mapping, there is no unused computing power. The throughput increases to 0.5/sec. This is the optimal throughput for this problem.

## 6 Experimental Results

We have performed various implementations for Radar benchmarks and Sonar benchmarks on IBM SP2 and SGI/Cray T3E to show the effectiveness of our mapping methodology. The experimental results are presented in this section. We first describe the details of our benchmark implementations. We then present results for Radar benchmarks, followed by results for Sonar benchmarks.

### 6.1 Implementation Details

We have implemented two Radar benchmarks. These benchmarks are taken from the RT\_STAP benchmark suite developed by the MITRE Corporation [32]. A range of benchmarks based on the overall computational complexity is described in [32]. We have considered the medium complexity and the hard

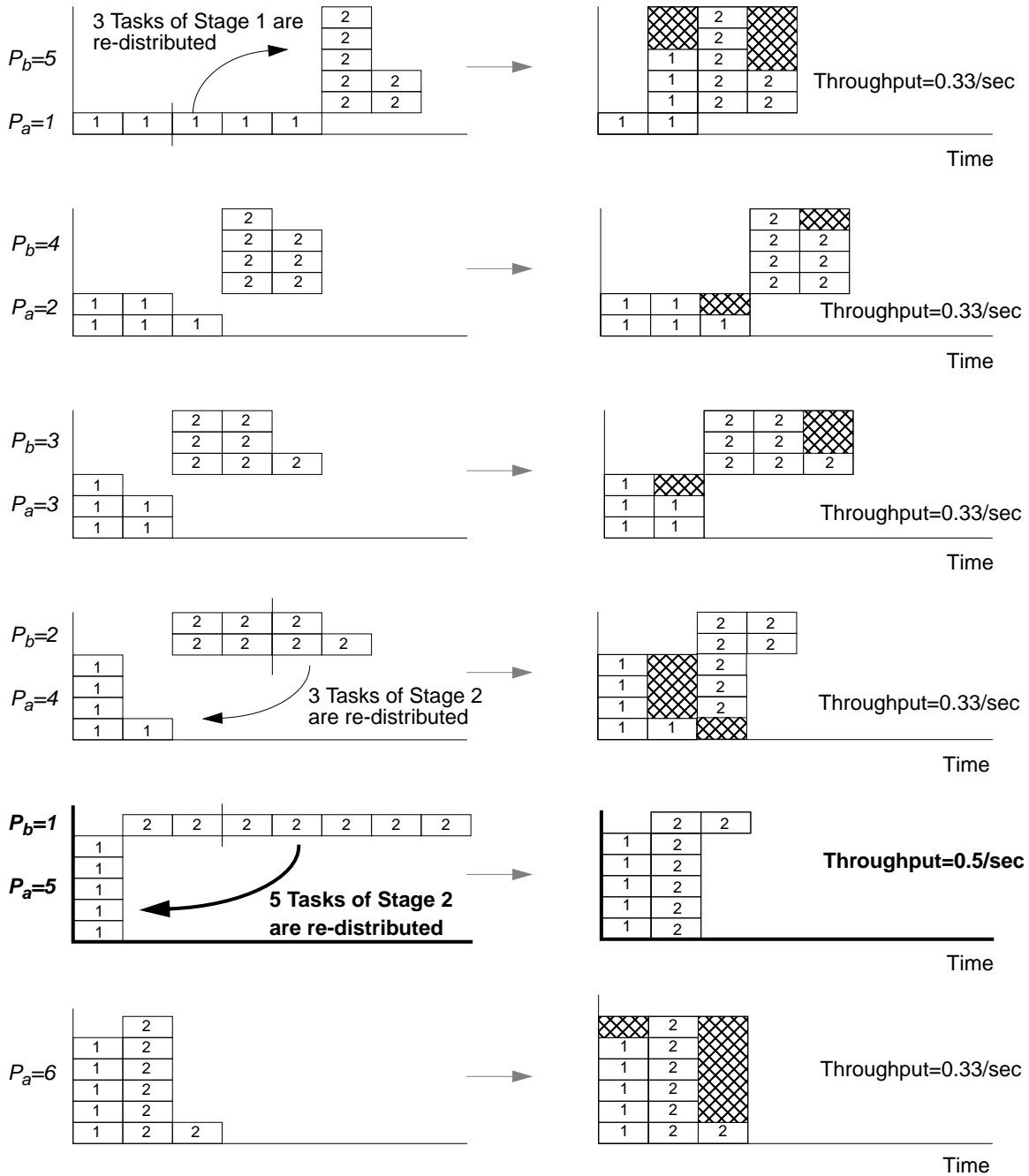


Figure 15: Illustration of stage partitioning for a two-stage application

Machine	Peak Performance of a Single Node	Start-up Time ( $\mu$ sec)	Transmission Time per Byte (nsec/byte)
IBM SP2	266 Mflops	40	28.6
Cray T3E	600 Mflops	28	5.45

Table 2: Computation and communication characteristics of IBM SP2 and SGI/Cray T3E used for our benchmark implementation

complexity benchmarks for our implementations. Both benchmarks consist of two main components: preprocessing step and adaptive array processing step. In our implementation, we focus only on the adaptive array processing step, because the most important functions (interference and clutter suppression) are performed in this step. Also, the preprocessing is not computationally demanding. Therefore, the considered benchmarks are First-Order Doppler-Factored STAP and Third-Order Doppler-Factored STAP. We also implemented an MVDR adaptive Sonar beamforming benchmark. The idea of MVDR is to maintain unity gain in the direction-of-look and to minimize the contributions to the output from interference (not the signal of interest) by adapting a weight matrix to minimize the output variance. The Naval Undersea Warfare Center (NUWC) proposed and implemented an MVDR adaptive beamformer [15]. Two experiments are performed in our implementations. Additional details of the Radar and Sonar benchmarks are described in Section 6.2 and Section 6.3.

Our implementations were performed on the IBM SP2 at Maui High Performance Computing Center (MHPCC) and on the SGI/Cray T3E at San Diego Supercomputer Center (SDSC). IBM SP-2 at the MHPCC is configured with up to 256 nodes. Each node is based on the POWER2 processor with multiple functional units. The nodes of SP2 are interconnected by a bidirectional Multistage Interconnection Network (MIN). SGI/Cray T3E at the SDSC is configured with upto 272 nodes. Each node is based on the DEC Alpha 21164 microprocessors. The interconnection network of T3E is a 3-D torus. Table 2 shows the computation and communication characteristics of the machines used for our implementation.

In our implementations, to compare our mapping methodology which is denoted as  $A3$ , we also considered two previous approaches (denoted as  $A1$  and  $A2$ ). We developed algorithm designs based on  $A1$ ,  $A2$ , and  $A3$ . These are denoted as  $D1$ ,  $D2$ , and  $D3$ . Designs  $D1$  and  $D2$  examine all possible task mappings under the linear execution model. However, no clustering or replication is used in  $D1$ .  $D2$  also uses the linear execution model but allows clustering and replication of stages. Note that  $D1$  and

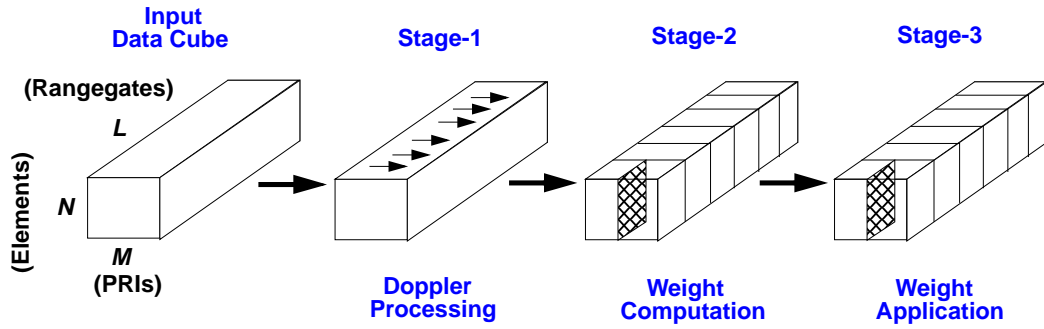
$D2$  are the best performance that can be achieved using the techniques in [6] and [27] respectively.

As described in Section 5.1,  $D3$  exploits data remapping. Data remapping can significantly improve the performance. However, it is not stated in [27] whether data remapping is explicitly performed or not. Thus,  $D2$  (as well as  $D1$ ) may use a fixed data layout throughout the computation stages. If data remapping is not used in  $D1$  and  $D2$ , the performance improvement of  $D3$  over  $D1$  and  $D2$  can be significantly higher. To expose the performance improvement using our task model and stage partitioning, we also perform data remapping between adjacent stages (Step 1 in our approach) to implement both  $D1$  and  $D2$ . Therefore, in the following sections, the observed performance improvement can be attributed solely to various task mapping techniques.

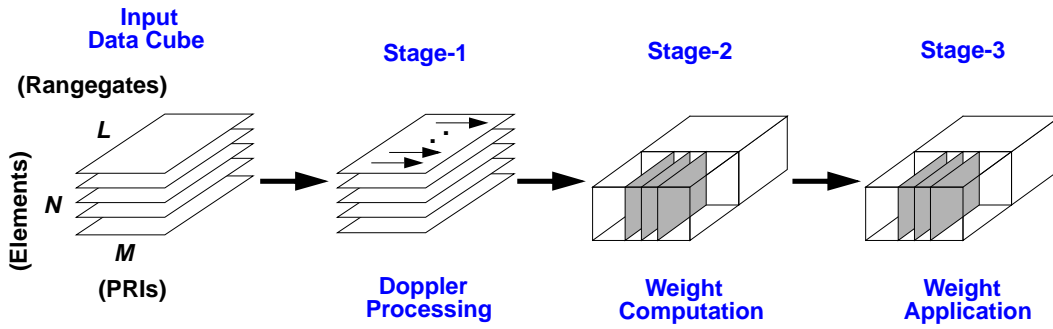
Our code is written using C, LAPACK (a standard linear algebra library), and the Message Passing Interface (MPI). LAPACK was used for implementing the covariance matrix factorization using QRD or SVD. MPI was used to implement data remapping between stages. By using these standards, our code is portable across various HPC platforms.

## 6.2 Results for Radar Benchmarks

We implemented a First-Order Doppler-Factored STAP and a Third-Order Doppler-Factored STAP as described earlier. In figure 16, these benchmarks are illustrated. Both benchmark consists of three computation stages: 1) Stage 1: Doppler processing; 2) Stage 2: Weight computation by covariance matrix factorization; 3) Stage 3: Weight application. Given an input data cube of size  $N$  (number of channels)  $\times M$  (PRI's)  $\times L$  (range gates), Doppler processing is implemented by applying a discrete Fourier Transform of length  $K$  across  $M$  pulses of the data for a given range gate and channel, where  $K$  represent the number of Doppler cells to be processed. Discrete Fourier transform is implemented using FFT. A set of adaptive weights is computed by factorizing a matrix using QR decomposition (QRD). For the Third-Order Doppler-Factored STAP, the covariance matrix is formed by using the data from adjacent Doppler bins centered around each bin. In the First-order Doppler-factored STAP, the covariance matrix is formed by using a single Doppler bin. Therefore, the amount of computation for a QRD is significantly reduced compared to the Third-Order Doppler-Factored STAP. In the following subsections, implementation results for both benchmarks are presented.



(a) First-Order Doppler-Factored STAP



(a) Third-Order Doppler-Factored STAP

Figure 16: Illustration of First-Order Doppler-Factored STAP and Third-Order Doppler-Factored STAP

Processing Stages	Computations Involved	Sequential Execution Time on IBM SP-2	Sequential Execution Time on SGI/Cray T3E	Number of tasks
Stage 1	Doppler Processing	28 $\mu$ sec	110 $\mu$ sec	7680
Stage 2	Weight Computation	9,800 $\mu$ sec	4,300 $\mu$ sec	384
Stage 3	Weight Application	100 $\mu$ sec	56 $\mu$ sec	384

Table 3: First-Order Doppler-Factored STAP on a single node of IBM SP2 and on a single node of SGI/Cray T3E

### 6.2.1 Results for First-Order Doppler-Factored STAP

A data cube of size 16 (number of channels)  $\times$  64 (PRI's)  $\times$  480 (range gates) was used as input for First-Order Doppler-Factored STAP. The sequential execution time of each task in each stage running on a single node of IBM SP2 and a single node of SGI/Cray T3E, and the number of tasks in each stage are shown in Table 3. Library routines (IBM essl library and LAPACK) were used for implementing the computational kernels (FFT, QRD, etc.) involved in each stage. MPI was used to implement data remapping between stages. Note that, in general, SGI/Cray T3E shows higher computation performance than IBM SP-2, except in Stage 1, where Doppler processing involving FFT is performed. The FFT routine in IBM essl library, `scft()`, offers impressive performance.

Figure 17 shows the throughput performance of First-Order Doppler-Factored STAP using  $D1$  through  $D3$  on IBM SP2 and SGI/Cray T3E. The number of processors was varied from 70 to 120 (in increments of 10 processors). Latency is not a main concern in this paper. However, those designs which generate latency greater than 0.3 seconds are not practical, and, thus, were not considered. The results show that the design based on our approach ( $D3$ ) consistently results in higher throughput performance than those designs using the previous approaches ( $D1$ ,  $D2$ ). Note that  $D1$  and  $D2$  correspond to the best performance possible using the approaches in [6] and [27] respectively, if data remapping is employed.

The results show superior performance of our design over  $D1$  and some improvement over  $D2$  on both SP2 and T3E. For instance, our approach increases the throughput by approximately 85.5% compared with  $D1$  and by 4.4% over  $D2$ , when 100 processors of IBM SP2 are used. In this benchmark, the workload of Stage 2 dominates that of Stage 1 and Stage 3. Furthermore, the sequential execution time of each task in Stage 2 is much larger than that in Stage 1 and Stage 3. Previous approaches based on the linear execution model result in inefficient task mapping for Stage 2. This stage becomes the bottleneck stage with the largest execution time ( $max_{period}$  defined in Step 3 of our mapping methodology described in Section 5.1) among all the stages. The tasks cannot be evenly mapped to the processors, unless the number of processors assigned to Stage 2 is an integer multiple of the number of tasks in this stage. Design  $D1$ , thus, results in low throughput performance.  $D2$  is also based on the linear execution model. However,  $D2$  achieves high throughput performance by using clustering and replication of stages. Our task mapping methodology based on the new execution model,  $A3$ , incorporates stage partitioning technique so that some of the tasks from Stage 1 (and/or Stage 3) can be clustered with those from Stage 2. Using this flexible task mapping technique, the execution time variance among the task clusters can be further smoothed out. Thus, a higher throughput can be achieved.

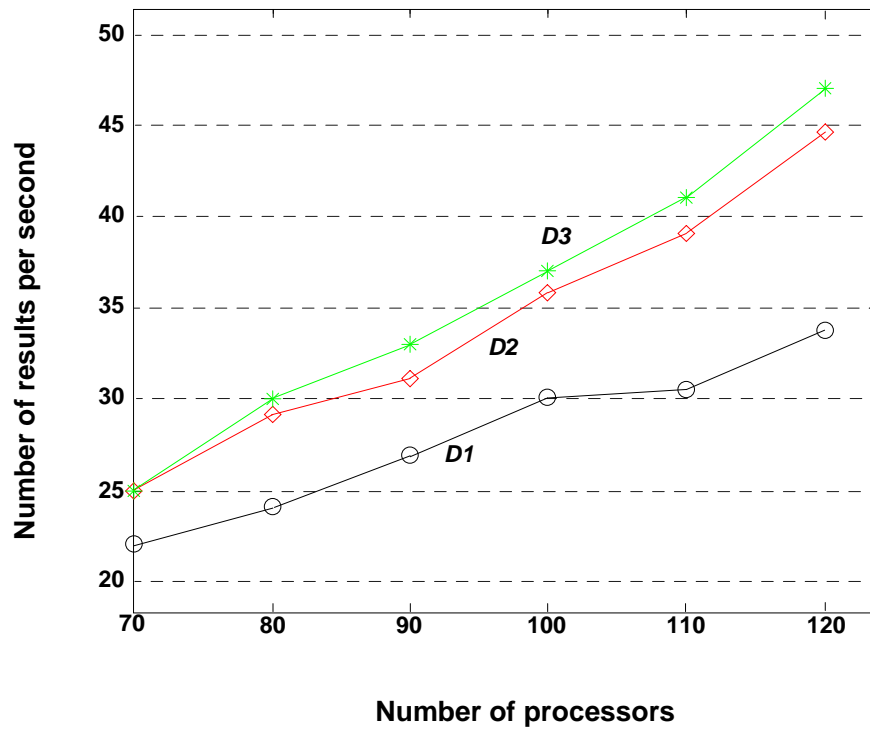
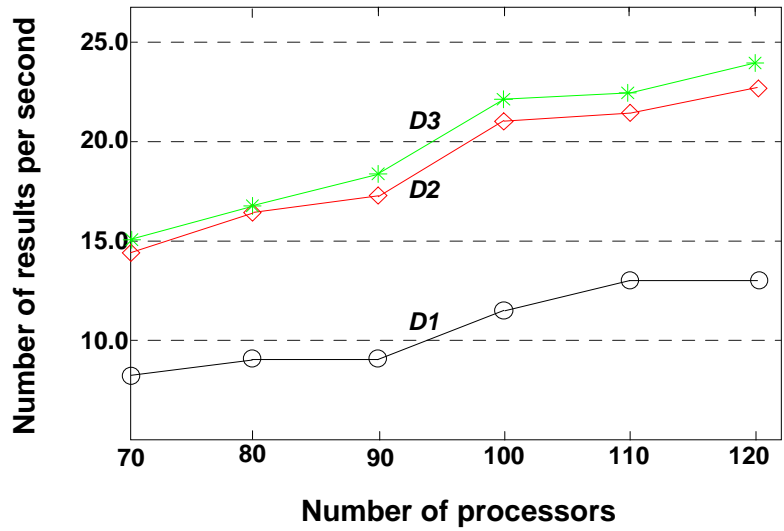


Figure 17: Performance Comparison of  $D1$  through  $D3$  for First-Order Doppler-Factored STAP on IBM SP2 (above) and SGI/Cray T3E (below)

Processing Stages	Computations Involved	Sequential Execution Time on IBM SP-2	Sequential Execution Time on SGI/Cray T3E	Number of tasks
Stage 1	Doppler Processing	28 $\mu$ sec	110 $\mu$ sec	10560
Stage 2	Weight Computation	499,000 $\mu$ sec	221,000 $\mu$ sec	128
Stage 3	Weight Application	1,270 $\mu$ sec	700 $\mu$ sec	128

Table 4: Third-Order Doppler-Factored STAP on a single node of IBM SP2 and on a single node of SGI/Cray T3E

However, in this benchmark, the performance improvement of design  $D3$  (using  $A3$ ) over  $D2$  is limited. Note that, in our implementation using  $D3$ , the performance improvement is achieved mostly from the improvement in the computation time. Improvement in the communication time is insignificant, because  $D1$ ,  $D2$ , as well as  $D3$  use efficient data remapping. In this benchmark, the computation time is small compared with the communication time. Therefore, the performance improvement of our technique is not high.

### 6.2.2 Results for Third-Order Doppler-Factored STAP

A data cube of size 22 (number of channels)  $\times$  64 (PRI's)  $\times$  480 (range gates) was used as input for the Third-Order Doppler-Factored STAP. The sequential execution time of each task in each stage running on a single node of IBM SP2 and a single node of SGI/Cray T3E, and the number of tasks in each stage are shown in Table 4. Library routines (IBM essl library, LAPACK) were also used for implementing the computational kernels (FFT, QRD, etc.) involved in each stage. MPI was used to implement data remapping between stages. Again note the higher computation performance of SGI/Cray T3E compared with IBM SP-2, except in Stage 1, where Doppler processing involving FFT is performed. Using the FFT routine in IBM essl library, `scft()`, the sequential execution time on IBM SP-2 is impressive.

Figure 18 shows the throughput performance of Third-Order Doppler-Factored STAP using  $D1$  through  $D3$  on IBM SP2 and SGI/Cray T3E. The number of processors was varied from 70 to 120 (in increments of 10 processors). As in the First-Order Doppler-Factored STAP, latency is not a main concern here. However, those designs which generate latency greater than 3 seconds are not practical, and, thus, were not considered. As in the First-Order Doppler-Factored STAP, the design based on our approach consistently results in higher throughput performance than those designs using the previous

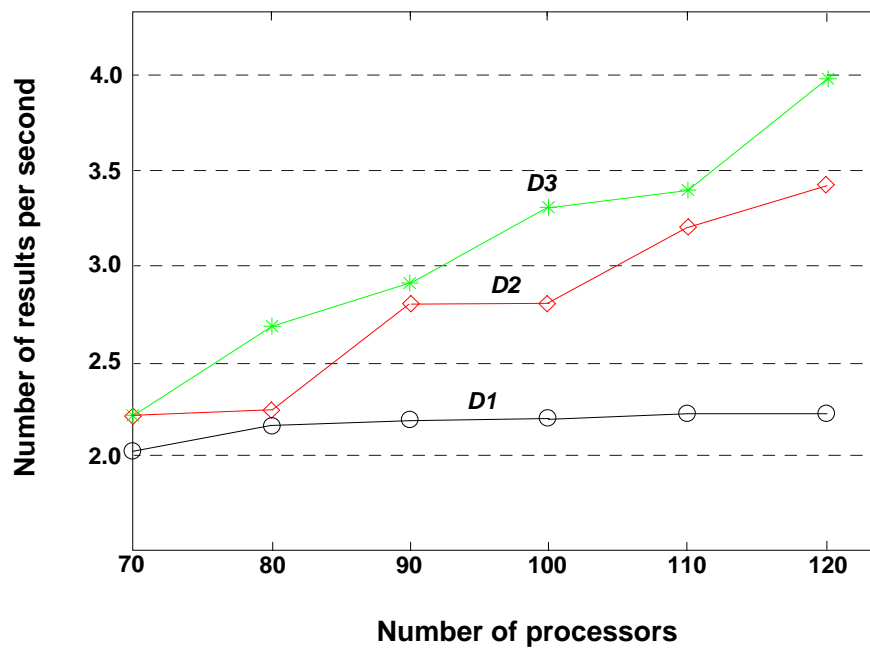
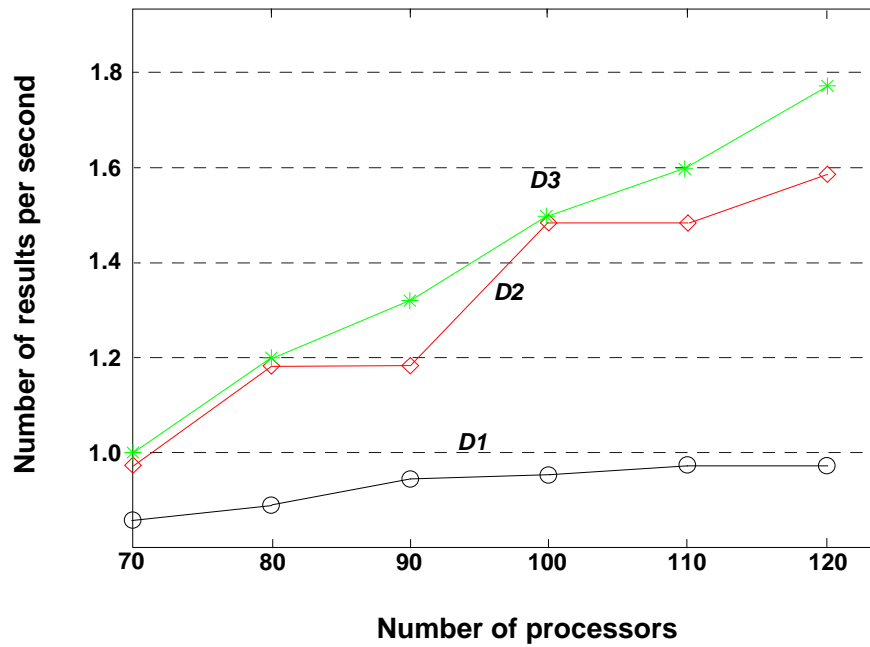


Figure 18: Performance Comparison of  $D1$  through  $D3$  for Third-Order Doppler-Factored STAP on IBM SP2 (above) and SGI/Cray T3E (below)

approaches.

As in the First-Order Doppler-Factored STAP, the design based on our approach ( $D3$ ) results in superior performance over the design  $D1$ . The performance improvement of the design  $D3$  over  $D2$  is larger than that in the case of First-Order Doppler-Factored STAP. For instance, our approach increases the throughput by approximately 78.9% over  $D1$  and by 15.7% over  $D2$  when 120 processors of SGI/Cray T3E are used. In this benchmark, the workload of Stage 2 dominates that of Stage 1 and Stage 3 as in the First-Order Doppler-Factored STAP. Also, the sequential execution time of each task in Stage 2 is much larger than that in Stage 1 and Stage 3. Thus, design  $D1$  results in inefficient utilization of the processors and results in low throughput performance. By clustering and replication of stages, design  $D2$  improves the performance over  $D1$ .  $D3$  further improves the performance over  $D2$ . The performance improvement of  $D3$  over  $D2$  is greater than that in the First-Order Doppler-Factored STAP. This is because, in this benchmark, computation dominates communication. Thus, the benefit of our flexible task mapping choices based on the new execution model becomes more noticeable. In fact, our approach results in smooth performance improvement and, thus, is scalable, whereas the previous approaches result in “step-shaped” performance curves as the number of processors increases. Note, again, that the designs shown here based on  $D1$  and  $D2$  correspond to the best possible results using the approaches in [6] and [27] respectively, if data remapping is employed.

### 6.3 Results for Sonar Benchmarks

A parallel algorithm was designed for an adaptive sonar beamforming benchmark using our methodology. The benchmark performs adaptive beamforming in frequency domain. The incoming time-domain signals are first converted to frequency domain using FFT's. Then, the resulting frequency-domain signals are linearly combined with fully adaptive weight matrices (See Figure 19). The details of the beamformer can be found in [15]. Two experiments were performed on IBM SP2. In Experiment 1, the beamformer uses 64 sensor elements ( $N$ ) to sample the signals. 64 frequency bins ( $F$ ) were chosen with 128 beams formed in each frequency bin ( $B$ ). In Experiment 2, the beamformer uses 128 sensor elements ( $N$ ) to sample the acoustic signals. 128 frequency bins ( $F$ ) were chosen with 256 beams formed in each frequency bin ( $B$ ) (See Table 5 for details). Library routines (IBM eszl library and LAPACK) were also used for implementing the computational kernels (FFT, SVD, etc.) involved in each stage. MPI was used to implement data remapping between stages.

Figure 20 shows the throughput performance of the implementations using  $D1$  through  $D3$  on IBM

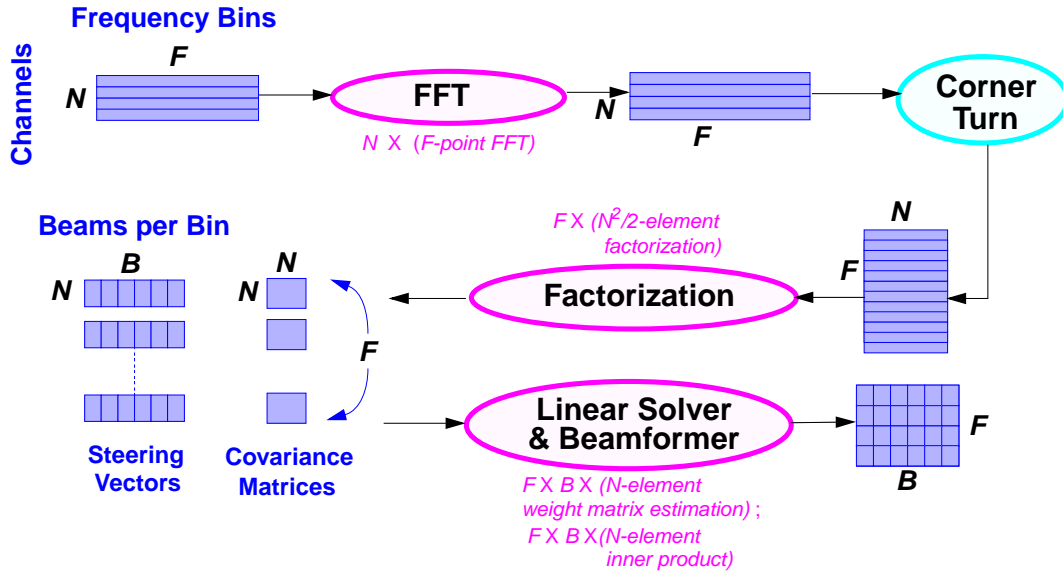


Figure 19: Illustration of MVDR Sonar beamformer

Processing stage	Functionality	Experiment 1		Experiment 2	
		T (μsec)	Ni	T (μsec)	Ni
Stage 1	FFT	88	64	260	128
Stage 2	Covariance matrix factorization	68,500	64	807,718	128
Stage 3	Weight adaptation and beamforming	1,900	8192	7,575	32,768

- The LAPACK subroutine, **cgsvd**, is used to perform single-precision complex-number singular value decomposition (SVD) on SP-2.
- T denotes the sequential execution time of each task in a stage on IBM SP-2.
- Ni denotes the number of tasks in a stage.

Table 5: Characteristics of the two MVDR Sonar beamformers

SP2. In Experiment 1 (2), the number of processors was varied from 70 to 125 (100 to 220) in increments of 5 processors. As in the Radar benchmark case, those designs which generate latency greater than 1 (12) second(s) are not practical and, thus, were not considered. In Experiment 1, when 75 or 100 processors are used, *D3*, as well as *D1* and *D2*, effectively utilize the computing power based on their corresponding execution models. The resulting throughputs, in these cases, are about the same. However, in most of the other cases, the resulting implementations using *D1* and *D2* cannot effectively utilize the computing power. In these cases, our design methodology results in superior performance over *D1* and *D2*. For instance, in Experiment 1, our approach increases the throughput by approximately 24.3% (21.4%) compared with *D1* (*D2*) when 125 processors are used. In Experiment 2, our approach increases the throughput by approximately 43.6% (13.5%) compared with *D1* (*D2*) when 210 (190) processors are used. The flexibility of our new execution model in mapping tasks results in higher throughput performance in Sonar benchmarks. Note, also, that in our implementations of *D1* and *D2*, data remapping was performed. Otherwise the performance of these algorithms is significantly worse due to excessive communication overheads.

In Sonar benchmarks, Stage 2 also has the largest sequential execution time per task. However, unlike Radar benchmarks, the workload of Stage 3 dominates that of Stage 1 and Stage 2 (See Table 5). The sequential execution time of each task in Stage 3 is small, thus, the tasks in Stage 3 can be efficiently mapped onto available processors. Thus, obtaining high performance depends on how efficiently the tasks of Stage 2 are mapped onto the processors. Computation dominates communication in Sonar benchmarks. Therefore, the performance results of the Sonar benchmarks are similar to those of the Third-Order Doppler-Factored STAP. The observed performance improvements of *D3* over *D2* are generally higher than those in the case of the Radar benchmarks.

## 7 Conclusion

In this paper, we developed a methodology to design parallel algorithms for optimizing the throughput performance for a class of ASP applications. Throughput is a key performance measure in this class of applications. However, this performance measure has not been paid much attention in traditional HPC applications.

A task model was first derived for ASP applications by exploiting their salient computational features. Coarse grain parallelism was considered in our task model to efficiently exploit the state-of-the-art HPC platforms. Based on the task model, we defined a new execution model. The new model exploits the

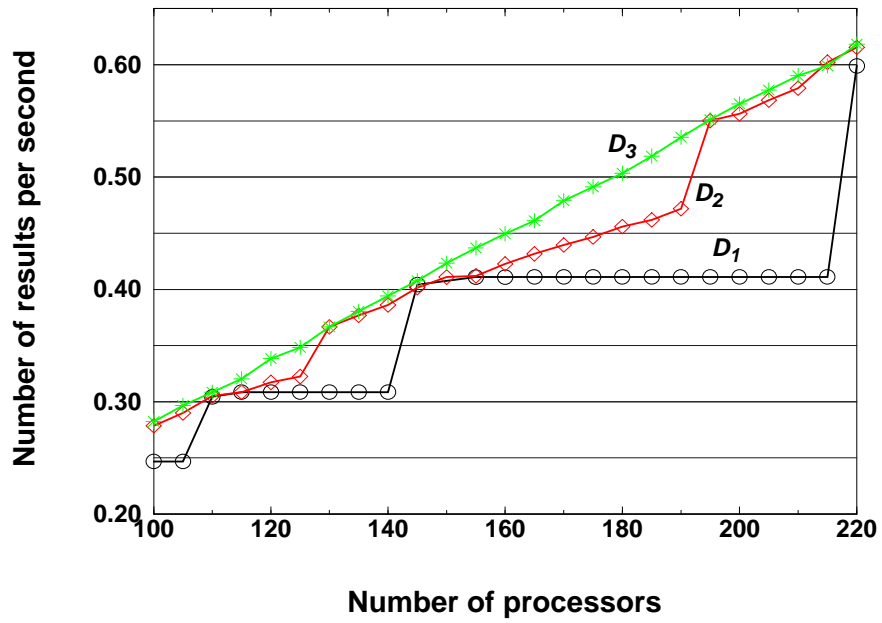
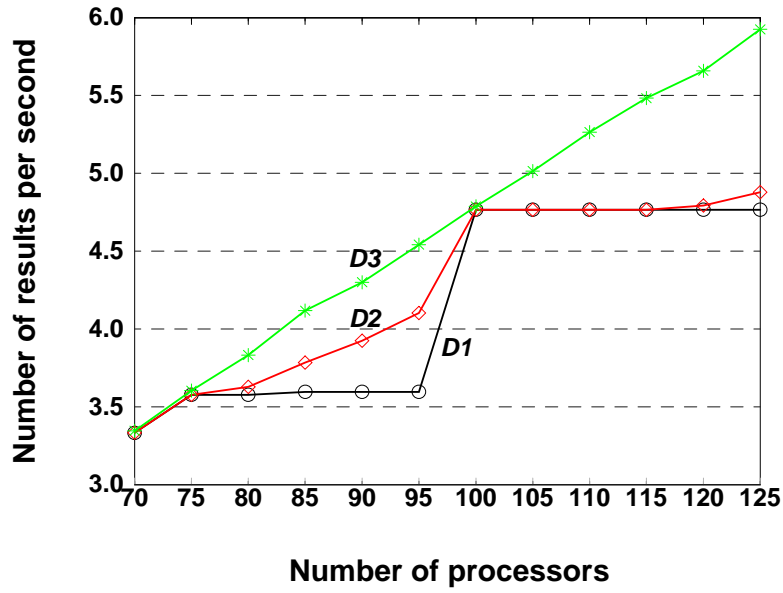


Figure 20: Performance Comparison of  $D1$  through  $D3$  in Experiment 1 (above) and Experiment 2 (below)

independency of tasks in each computation stage of the ASP application. It also relaxes the one-to-one mapping between the sets of processors and the computation stages assumed in the earlier linear execution model. Therefore, the new model allows more flexible task mapping choices, hence, leads to better throughput performance than the linear execution model. A novel stage partitioning technique was used to realize the new model by partitioning a stage into several subsets and mapping those subsets onto processors.

A heuristic task mapping methodology was developed based on the new execution model. It consists of three steps: data remapping; coarse resource allocation; and fine performance tuning. Using this mapping methodology, parallel algorithms were designed for modern Radar and Sonar signal processing applications. These algorithms were implemented on IBM SP2 and SGI/Cray T3E, state-of-the-art HPC platforms. The performance results show that our approach results in significant performance improvement over previous approaches.

Besides the task mapping problem addressed in this paper, there are a number of important future research areas for ASP applications that are worthy of exploration:

1. Many of the computing platforms for ASP applications integrate heterogeneous components. For example, some nodes may be specialized to do matrix factorization, while others may be designed for FFT operations. In this scenario, task mapping to meet a given performance requirement and synthesizing a system to minimize the number of processors are interesting problems that require further attention.
2. The computing platform for ASP applications also have physical constraints such as size, weight, and power. Algorithmic issues related to these constraints need to be studied to develop a comprehensive framework to aid system designers in developing ASP applications.
3. During the past few years, several new architectures have been proposed and implemented for signal and image processing and/or commercial applications. These include Digital Signal Processor (DSP) clusters, Symmetric Multi Processors (SMPs) and Distributed Shared Memory (DSM) machines. In these systems, to obtain high efficiency, the memory hierarchy must be carefully managed.

We have used radar and sonar applications to illustrate our ideas. Our related results in using HPC for ASP applications can be found in [29, 30, 19].

## 8 Acknowledgment

The implementations reported in this paper were performed on the IBM SP2 at the Maui High Performance Computing Center (MHPCC). Implementations on SGI/Cray T3E were performed at the San Diego Supercomputer Center (SDSC).

## References

- [1] E. Anderson, et. al., "LAPACK Users' Guide - Release 2.0,"  
URL: [http://www.netlib.org/lapack/lug/lapack\\_lug.html](http://www.netlib.org/lapack/lug/lapack_lug.html).
- [2] P. Athanas and A. Abbott, "Addressing the Computational Requirements of Image Processing with a Custom Computing Machine: An Overview", in Proc. Workshop on Reconfigurable Architectures, at IPPS'95, Santa Barbara, CA, 1995.
- [3] I. Banicescu and S. Hummel, "Balancing Processor Loads and Exploiting Data Locality in Irregular Computations," *IBM Research Report*, February 1995.
- [4] R. Bernecky, "Sonar Beamforming Challenge Problems," presented at the DARPA/ITO Embeddable Systems PI Meeting, San Diego, June 1996.
- [5] P. Bhat, Y. W. Lim, and V. Prasanna, "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.
- [6] A. Choudhary, B. Narahari, D. Nicol, and R. Simha, "Optimal Processor Assignment for a Class of Pipelined Computations," *IEEE Trans. Parallel and Distributed Systems*, Vol. 5, No. 4, April 1994, pp. 439-445.
- [7] A. Ferreira and J. Rolim (Eds.), "Parallel Algorithms for Irregularly Structured Problems," Proceedings of Second International Workshop (IRREGULAR '95), Lyon, France, September, 1995.
- [8] R. A. Games, "Benchmarking Methodology for Real-Time Embedded Scalable High Performance Computing," MITRE Technical Report MTR 96B0000010, March 1996.
- [9] D. Gerogiannis and S. Orphanoudakis, "Load Balancing Requirements in Parallel Implementations of Image Feature Extraction Tasks," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 4, No. 9, pp. 994-1013, 1993.

- [10] S. Haykin, "Adaptive Filter Theory", 3rd Edition, Prentice Hall, 1996.
- [11] "High Performance Fortran Forum,"  
URL: <http://www.crpc.rice.edu/HPFF/home.html>.
- [12] S. D. Kaushik, C.-H. Huang, J. Ramanujam, and P. Sadayappan, "Multiphase Array Redistribution: Modeling and Evaluation," in proceedings of the 9th International Parallel Processing Symposium (IPPS '95), Apr. 1995.
- [13] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Parallel Algorithms*, Benjamin/Cummings, 1994.
- [14] M. Lee, W. Liu, and V. K. Prasanna, "A Mapping Methodology for Designing Software Task Pipelines for Embedded Signal Processing," in proceedings of the 3rd International Workshop on Embedded HPC Systems and Applications (EHPC '98) at the 12th International Parallel Processing Symposium (IPPS '98), and the 9th Symposium on Parallel and Distributed Processing (SPDP '98), Orlando, April 1998.
- [15] M. Leonhardt, "Implementation of Minimum Variance Distortionless Response (MVDR) Adaptive Beamforming Algorithm," NUSC Technical Document 8453, July 1989.
- [16] Y. W. Lim and V. K. Prasanna, "Efficient Algorithms for General Block-Cyclic Redistribution," Technical Report, Department of EE-Systems, USC, Aug. 1996.
- [17] Y. W. Lim and V. K. Prasanna, "Scalable Portable Implementations of Space-Time Adaptive Processing," in proceedings of the 10th International Conference on High Performance Computers, June 7, 1996.
- [18] Y. W. Lim, P. B. Bhat, and V. K. Prasanna, "Efficient Algorithms for Block-Cyclic Redistribution of Arrays," IEEE Symposium on Parallel and Distributed Processing, Oct. 1996.
- [19] Y. W. Lim, P. B. Bhat, and V. K. Prasanna, "Efficient Algorithms for Block-Cyclic Redistribution of Arrays," *Algorithmica*, to appear.
- [20] W. Liu, C. Wang, and V. K. Prasanna, "Portable and Scalable Algorithms for Irregular All-to-all Communication," in proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96), May 1996.

- [21] M. Maresca, "Polymorphic Processor Arrays", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 5, pp. 490-506, May 1993.
- [22] "Message Passing Interface Standard,"  
URL: <http://www.mcs.anl.gov/mpi/index.html>.
- [23] P. Narayanan, L. Chen, and L. Davis, "Effective Use of SIMD Parallelism in Low- and Intermediate-Level Vision," *IEEE Computer*, Vol. 25, No. 2, pp. 68-73, 1992.
- [24] C. Ou and S. Ranka, "Parallel Remapping Algorithms for Adaptive Problems," *Proc. of Symposium on the Frontiers of Massively Parallel Computation*, pp. 367-374, 1995.
- [25] V.K. Prasanna and C.-L. Wang, "Parallelizing Vision Computations on CM-5: Algorithms and Experiences," Proceedings of IRREGULAR '95, September, 1995.
- [26] A. Skjellum, "Embedded, Real-time MPI and MsgWay,"  
URL: <http://www.ito.arpa.mil/ResearchAreas/Embeddable.html>.
- [27] J. Subhlok, and G. Vondran, "Optimal Mapping of Sequences of Data Parallel tasks," in proceedings of the Fifth ACM SIGPLAN Symposium of Principles and Practice of Parallel Programming, July 1995.
- [28] J. Subhlok and G. Vondran, "Optimal Latency-Throughput Tradeoffs for Data Parallel Pipelines," *Proc. Eighth Annual ACM Symposium on Parallel Algorithms and Architecture (SPAA)*, June 1996.
- [29] J. Suh and V. K. Prasanna, "Portable Communication Algorithms for Implementing SAR," First Annual High-Performance Embedded Computing Workshop, Lexington, MA, September 1997.
- [30] J. Suh and V. K. Prasanna, "Parallel Implementations of Synthetic Aperture Radar on High Performance Computing Platforms," in proceedings of the IEEE International Conference on Algorithms And Architectures for Parallel Processing, Melbourne, Australia, December 1997.
- [31] W.F. Tichy, "Should computer scientists experiment more?," *IEEE Computer*, 31:32-40, 1998.
- [32] J.A. Torres, and R.T. Williams, "RT-STAP: Real-Time Space-Time Adaptive Processing Benchmark", MITRE Corporation, Feb. 1997.
- [33] B. D. Van Veen and K. M. Buckley, "Beamforming: A Versatile Approach to Spatial Filtering," *IEEE ASSP Magazine*, Apr. 1988.

- [34] C. L. Wang, P. B. Bhat, and V. K. Prasanna, “High-Performance Computing for Vision,” in proceedings of the IEEE, vol. 84, No. 7, July 1996.
- [35] J. Ward, “Space-Time Adaptive Processing for Airborne Radar,” Technical Report 1015, Massachusetts Institute of Technology, Lincoln Laboratory, Dec. 1994.
- [36] J. Webb, “High Performance Computing in Image Processing and Computer Vision,” *International Conference on Pattern Recognition*, pp. 218-222, September 1994.
- [37] C. C. Weems, S. P. Levitan, A. R. Hanson, and E. M. Riseman, “The Image Understanding Architecture,” *International Journal of Computer Vision*, 2, 251-282(1989).
- [38] M.V. Zelkowitz and D. Wallace, “Experimental models for validating technology”, *IEEE Computer*, 31:23–31, 1998.