Design of Application Software for Embedded Signal Processing

Wenheng Liu and Viktor K. Prasanna

CENG 98-05

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213) 740-4483

February 1998

Design of Application Software for Embedded Signal Processing [*]

Wenheng Liu and Viktor K. Prasanna

Department of EE-Systems, EEB-200C
University of Southern California
Los Angeles, CA 90089-2562
http://ceng.usc.edu/~prasanna
{liu + prasanna}@halcyon.usc.edu

Abstract

High Performance Computing (HPC) technology is being used to provide scalable and cost-effective solutions to many Embedded Signal Processing (ESP) applications. Such applications are computationally intensive and require real-time performance. To meet such requirements, algorithmic techniques are needed to effectively utilize available HPC platforms.

In this paper, we address the design issues in performing embedded signal processing applications on HPC platforms. The computational characteristics of ESP applications are first identified. Then, the state-of-the-art in HPC technology is briefly reviewed. Based on the features of the ESP applications and those of the HPC platforms, we identify the key issues in designing ESP application software. We define a task model to capture the features of ESP applications. Various design problems are defined using this model. Then, we address the issues in developing scalable and portable algorithms for these applications. The algorithmic issues are illustrated using our task mapping methodology. This methodology is developed based on our execution model using a novel technique called *stage partitioning*. There are three steps in the task mapping methodology: data remapping, coarse resource allocation, and fine performance tuning. Using this methodology, a case study is shown in parallelizing an adaptive sonar beamformer on an IBM SP-2.

[*]Work supported in part by DARPA Embeddable Systems Program under contract no. DABT63-95-C-0092 monitored by Fort Hauchuca.

1 Introduction

Recently, High Performance Computing (HPC) technology has been employed to realize Embedded Signal Processing (ESP) applications such as radar and sonar systems. Such systems repeatedly sample radar or acoustic signals by an array of sensor elements. The sampled data is processed on-the-fly by a computing system embedded on an air-borne or a sea-borne vehicle. Thus, real-time constraints are to be met. Besides, when such applications are implemented, system compactness becomes an important issue since these radar and sonar systems are restricted in size, weight, and power [6]. Computationally demanding algorithms have been proposed for such applications. These algorithms are typically composed of a linear chain of processing stages. Each stage consists of a large number of identical tasks (ex., FFTs or QR-decomposition algorithms). Based on the computational characteristics and real-time requirements, these applications demand sustained performance in the range of 1 GFLOPs/s to 50 TFLOPs/s.

State-of-the-art HPC platforms are integrated using commercial off-the-shelf (COTS) components. From a signal processing users' perspective, such a technology can reduce the acquisition time and cost, and can offer system scalability and design flexibility. These advantages make such platforms to be attractive for ESP applications [9]. Algorithmic techniques are needed for realizing high performance. These techniques must be developed so that 1) a suitable platform can be configured based on the computational features of the application to be developed, and 2) the available computing power of a given platform can be effectively utilized in performing the application.

The main focus of this paper is the design of ESP application software. We identify the characteristics of such applications in terms of their computational requirements, data layouts, and latency and throughput constraints. An ESP application, *adaptive sonar beamformer*, is described to illustrate the typical values of the data sizes, throughput rates, and computational complexities. Then, we briefly survey the state-of-the-art in HPC technology. We address the advantages and challenges in using HPC technology for implementing ESP applications. To describe the software design issues in this context, we define a task model to capture the features of ESP applications. This model specifies the independent activities in each processing stage. We also identify various optimization problems in parallelizing ESP applications.

Following the above, we address the key issues in developing scalable and portable algorithms for ESP applications. An approach for developing parallel signal processing algorithms is to exploit fine-grain parallelism (i.e. each task such as an FFT or a QRD is mapped onto several processors). However, these algorithms, when tailored to suit ESP applications, usually result in large communication overhead. Thus, scalability and efficiency suffer. In this paper, we focus on the algorithmic issues in exploiting coarse-grain parallelism. These issues include design of data layouts and task mapping.

We show a task mapping methodology for application software development. The task mapping methodology is developed based on our execution model [12]. This model uses a novel *stage partitioning* technique to exploit the independent activities in a processing stage. We use our methodology to maximize the throughput of an ESP application for a given platform size. There are three steps in this methodology. Step 1 performs *Data Remapping*. This step remaps data between adjacent stages by determining the data layout of each stage. Data remapping techniques are used to change the data layout between successive stages. Step 2 performs *Coarse Resource Allocation*. This step allocates the available processors to the processing stages to form a linear software pipeline. Step 3 performs *Fine performance Tuning*. This step realizes stage partitioning using a heuristic. The heuristic algorithm reduces the period of the pipeline in an iterative manner. The resulting application software using this methodology is called a "software task pipeline". An adaptive sonar beamformer has been implemented using this design methodology. Experimental results are summarized to illustrate the effectiveness of our design methodology.

The rest of the paper is organized as follows: Section 2 summarizes the features of ESP applications and briefly describes the state-of-the-art HPC technology. Section 3 defines a task model and identifies various design problems that arise in using this technology for ESP applications. Section 4 addresses the issues in the design of scalable ESP application software. Section 5 describes a task mapping methodology for parallelizing such applications. Section 6 describes a case study in developing a sonar application on a HPC platform. Section 7 concludes the paper.

2 Embedded Signal Processing Applications and HPC Technology

This section summarizes the characteristics of typical ESP applications. The computational requirements and real time constraints of an illustrative ESP application are also described. State-of-the-art HPC technology to perform such ESP applications is, then, surveyed.

2.1 An Example ESP Application: Adaptive Sonar Beamforming

An ESP application is typically composed of a sequence of processing stages. Each stage performs a large number of identical tasks. For example, each task can be a FFT or a QR-decomposition (QRD).

Each stage repeatedly receives its input from the previous stage, performs the computations, and sends its output to the next stage. The first stage receives the external 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.

ESP applications have real time constraints assigned to them. These constraints usually include *latency* (i.e., response time), and *throughput* (i.e., number of results produced per unit time). The latency constraint is set so that the system variables (ex. beam-patterns and/or adaptive weights in adaptive beamforming applications) can be updated within a given time to adapt the system to the environment. The throughput constraint is set so that the processing can keep pace with the input data rate.

An *adaptive sonar beamforming* application is described in the following to illustrate the characteristics of a typical ESP application. Throughout the paper, we use *the number of floating point operations* (FLOPs) of the application to be the metric denoting the total computational complexity of the tasks in an application. In contrast, we define *the number of floating point operations per second* (FLOPs/s) to be the metric denoting the computing power required for performing the application in real-time (based on a given real-time constraint).

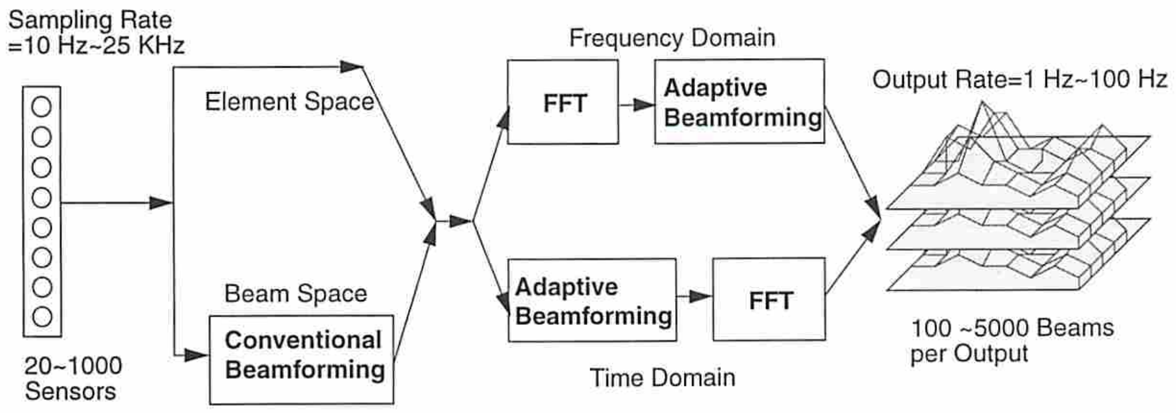


Figure 1: Some adaptive sonar beamforming techniques and their performance requirements

Beamforming is the technique which spatially filters the signals received from an array of sensors and estimates the spatial features of the sources [27]. 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

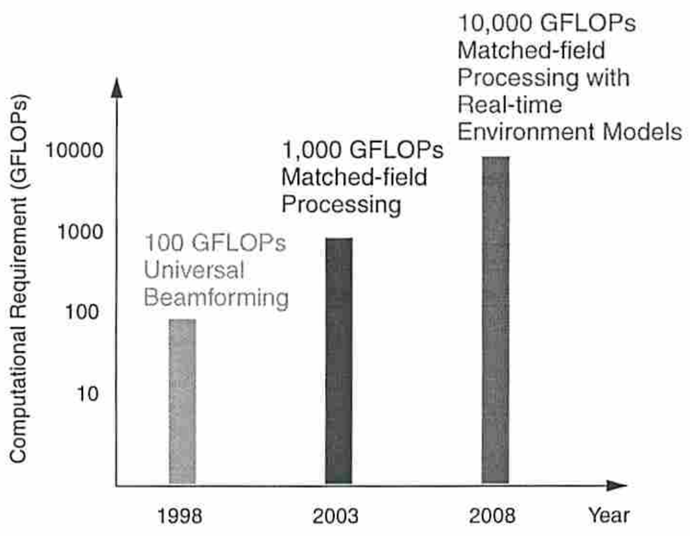


Figure 2: Projected increase in the computational requirements for sonar beamforming

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 used, sometimes, to reduce the dimensionality of adaptive weight matrices [27]. 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. Figure 1 shows the various techniques for adaptive sonar beamforming. Adaptive sonar beamforming is a computationally intensive ESP application. Figure 2 shows the projected increase in the computational complexity of present and next generation sonar beamformers [2]. The increase in the complexity is mainly due to the use of more advanced numerical methods in weight adaptation and due to the increase in the problem size. 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.

To further illustrate the real-time computational requirement, we consider a sonar benchmark application. This application employs a Minimum Variance Distortionless Response (MVDR) algorithm to

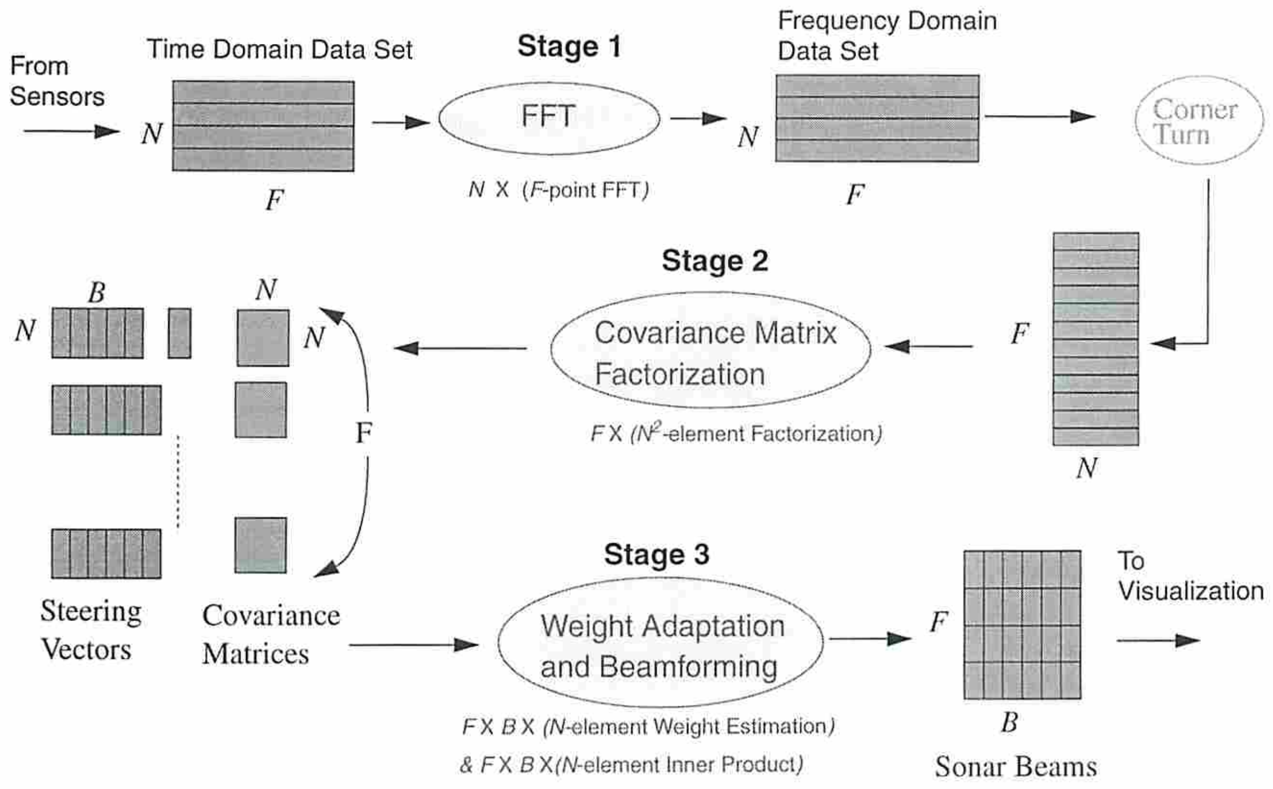

From Sensors
Time Domain Data Set
N
F
Stage 1
FFT
N X (F-point FFT)
Frequency Domain Data Set
N
F
Corner Turn
Stage 2
Covariance Matrix Factorization
F X (N²-element Factorization)
F
N
B
N
N
N
F
Steering Vectors
Covariance Matrices
Stage 3
Weight Adaptation and Beamforming
F X B X (N-element Weight Estimation)
& F X B X(N-element Inner Product)
F
B
Sonar Beams
To Visualization

Figure 3: A MVDR adaptive sonar beamformer

perform adaptive sonar beamforming. Figure 3 shows the processing stages of this benchmark. Table 1 shows the computational complexity of the various stages. In Table 1, N denotes the number of sensors, F denotes the number of frequency bins, and B denotes the number of beams per bin. These parameters can be in the following range: N=64 to 1024, F=64 to 1024, and B=128 to 1024. The value of these parameters can vary significantly depending on the design of the system and its intended purpose. Assuming the throughput of this benchmark to be in the range of 1 to 5 results per second, to realize this benchmark, the needed computing power is in the range of 290 MFLOPs/s (10^6 FLOPs/s) to 44 TFLOPs/s (10^{12} FLOPs/s). Note that this is the needed sustained performance. The computational requirements of ESP applications are generally beyond the computational capacity of a single processor. In the next subsection, we briefly describe the state-of-the-art HPC technology for parallelizing ESP applications.

Table 1: Characteristics of a MVDR sonar benchmark

Processing Stage	Functionality	Complexity of each Task	Number of Tasks
Stage 1	**FFT**	$8F\log_2 F$	N
Stage 2	**Covariance Matrix Factorization**	$33N^2$	F
Stage 3	**Weight Adaptation and Beamforming**	$8(N^2+N)B$	FB

2.2 HPC Technology for Embedded Signal Processing

Most of the state-of-the-art HPC platforms are built using commercial off-the-shelf (COTS) components. With recent advances in semiconductor technology, the COTS components have steadily improved in computational performance, cost, and/or capacity. Using these components, HPC platforms are being built to achieve computing power in the range of GFLOPs/s (10^9 FLOPs/s) and ultimately TFLOPs/s (10^{12} FLOPs/s) for various scientific and engineering applications. These platforms, typically, consist of three main components: 1) compute nodes (and memory modules situated physically close to each node), 2) I/O devices, and 3) high speed interconnects [4]. Each of the compute nodes and the I/O devices are coupled to the network using a network interface. The architecture of these platforms is flexible; they can be easily expanded by including new components. Examples of such HPC platforms include IBM SP-2, Cray T3D/T3E, Intel Paragon, the Embedded High Performance Scalable Computing System(EHPSCS) platform at Lockheed Martin Labs, the Myricom 2-level multicomputer, the RACE architecture from Mercury, and the Embedded Touchstone HPC from Honeywell. The high computational capacity, ready availability and cost effectiveness of these systems have led to their use in parallelizing embedded signal processing applications [28].

Several parallel programming paradigms have been used to realize applications on HPC platforms. These paradigms provide user-level functionality for utilizing and exploiting the architectural features of the platforms. Various parallel programming paradigms including shared variable, data parallel, and message passing have been defined and compared in [9].

The Use of HPC technology for embedded signal processing provides several advantages over the earlier solutions based on custom VLSI [4]. The VLSI based approach uses dedicated VLSI chips (ex. systolic networks) to perform the computations. Since the algorithms are built into the hardware, the implementations need substantial hardware as well as software redesign when new signal processing algorithms are developed.

Although the use of HPC for ESP applications is advantageous with respect to system scalability, programmability, and flexibility, the COTS components are not customized to achieve superior performance for a particular application. Thus, the hardware integration of the COTS components alone may not be sufficient to meet performance requirements of these applications. Besides, when the tasks in an ESP application are mapped onto a HPC platform, communication is needed among the tasks. Communication latency is the major overhead in HPC platforms and is significantly higher than systolic type architectures. The communication step in a systolic architecture essentially consists of a processor reading from a register of an adjacent processor in one clock cycle, whereas in a typical HPC platform, the start-up time to initiate a communication activity can be in the order of tens of microseconds. Table 2 shows the observed start-up time for message passing on various HPC platforms. Message passing has been widely used to perform communication in parallelizing ESP applications. Issues in using message passing paradigm are addressed in Section 4.2. To exploit the HPC technology for ESP applications, algorithmic techniques are required for mapping and scheduling the efficient computation and communication. These techniques can result in the effective utilization of the available computational power provided by the HPC platforms.

Table 2: Communication features of some HPC platforms

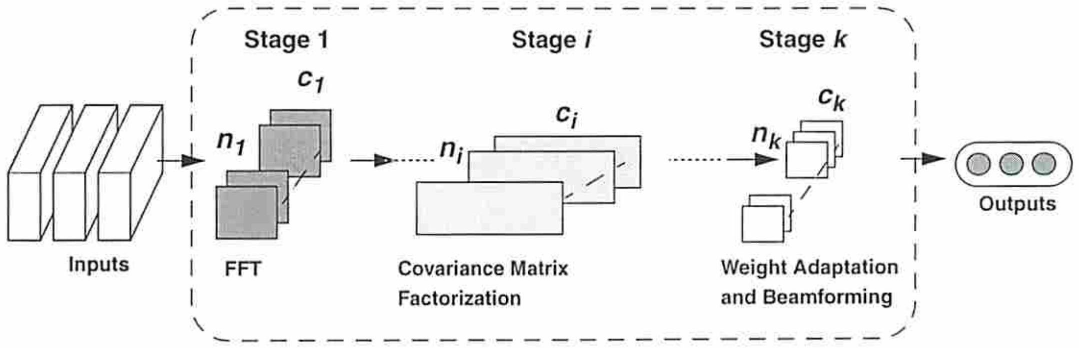
Machine	Start-up Time(μsec)
SP-2	39 ~ 46
T3E	28
Paragon	37.7

3 Design Problems

In this section, we identify the optimization problems that arise in implementing ESP applications on HPC platforms. We first propose a task model of ESP applications. This model captures the salient computational features of ESP applications. Based on this task model, we define various task mapping problems that arise in peforming ESP applications on HPC platforms.

3.1 A Task Model of Embedded Signal Processing

An ESP application consists of a sequence of processing stages. In each stage, a number of tasks are performed. Let n_i denote the number of tasks in *Stage* i and c_i denote the time complexity of each task in *Stage* i, where, $1 \leq i \leq k$. The tasks in each stage execute on disjoint sets of input data. Each stage repeatedly receives its input data from the previous stage, performs computations, and sends its output data to the next stage. Figure 4 illustrates this task model. Such data processing is typical of radar and sonar systems. Note that the data access pattern of each stage can be different. Thus, data remapping may have to be performed to reduce the communication time within a stage [15].



Stage 1
Stage i
Stage k
c1
ci
ck
n1
ni
nk
Inputs
FFT
Covariance Matrix Factorization
Weight Adaptation and Beamforming
Outputs

Figure 4: A task model representation of an embedded signal processing application

3.2 Optimization Problems

We identify several design problems that arise in performing ESP applicatons on HPC platforms. These include the optimization of the real-time performance and the system synthesis problems. The key parameters to be optimized in such systems are latency, throughput and the number of processors employed. In addition, these systems are constrained by size, weight, and power. These issues, however, are not considered in this work. Figure 5 illustrates the system latency and throughput in performing an ESP application. In this figure, this application is mapped onto a HPC platform with P (homogeneous) processors. The signal processing application is initiated repeatedly to process the incoming data samples. Based on the three parameters to be optimized, a sample of the design problems are:

- For a given number of nodes,
  - Minimize latency (no throughput constraint).
  - Minimize latency while satisfying a given throughput constraint.

  - Maximize throughput (no latency constraint).
  - Maximize throughput while satisfying a given latency constraint.

- Find a mapping of the tasks onto the processors such that the number of processors used is minimized while satisfying a given throughput and/or latency constraint(s).

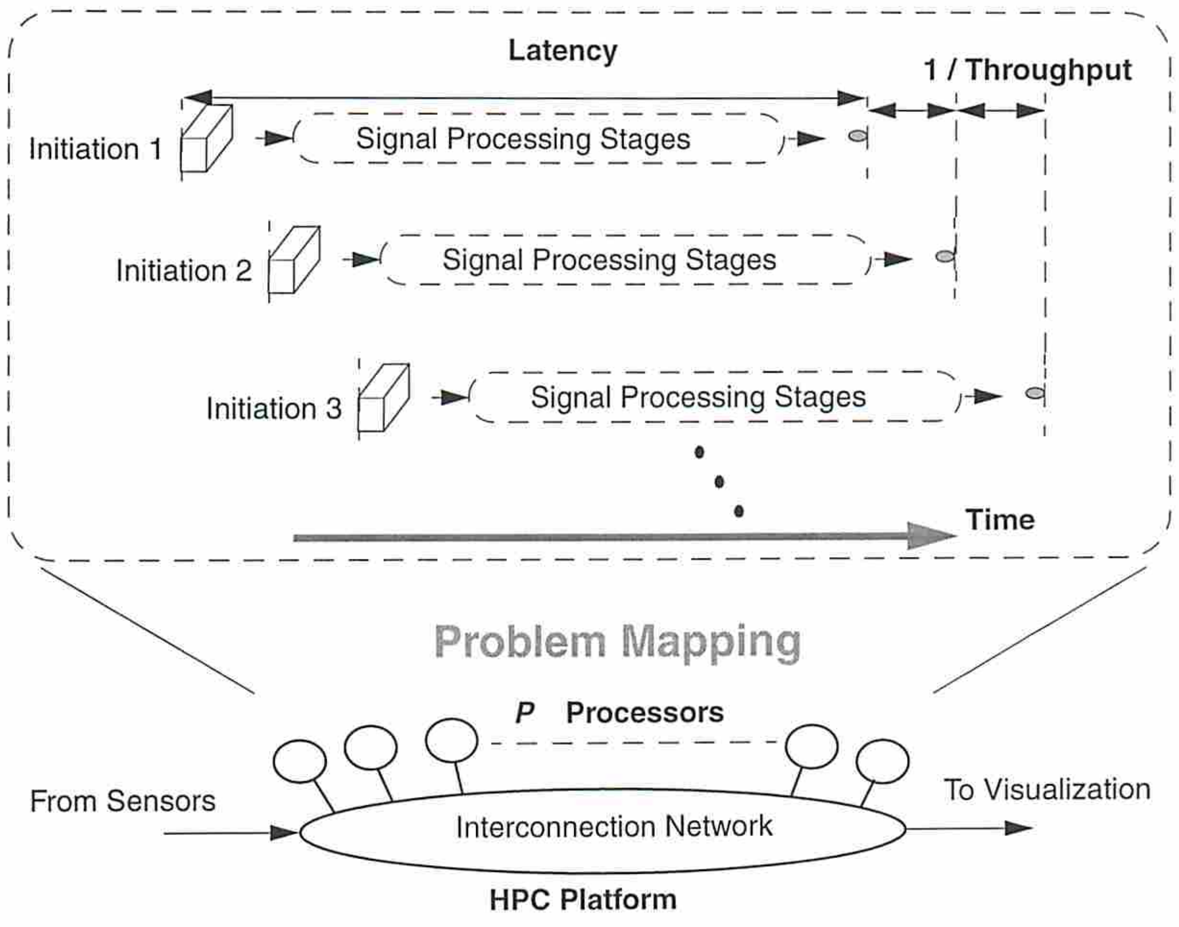


Figure 5: Illustration of latency and throughput when an ESP applicaton is mapped onto HPC platforms

4 Scalable and Portable Algorithm Design

In this section, we address the issues in designing scalable and portable algorithms using coarse-grain computing. We define a computational model to represent HPC platforms. This model is suitable for developing algorithms for ESP applications. Then, we discuss the key algorithmic issues in achieving scalable performance.

4.1 Scalability

A parallel algorithm is considered scalable if the execution time of the algorithm on a machine with P processors is proportional to $1/P$. As described in Section 2.2, when an ESP application is mapped onto a HPC platform, communication time is the major overhead which can adversely impact the scalability of the solution.

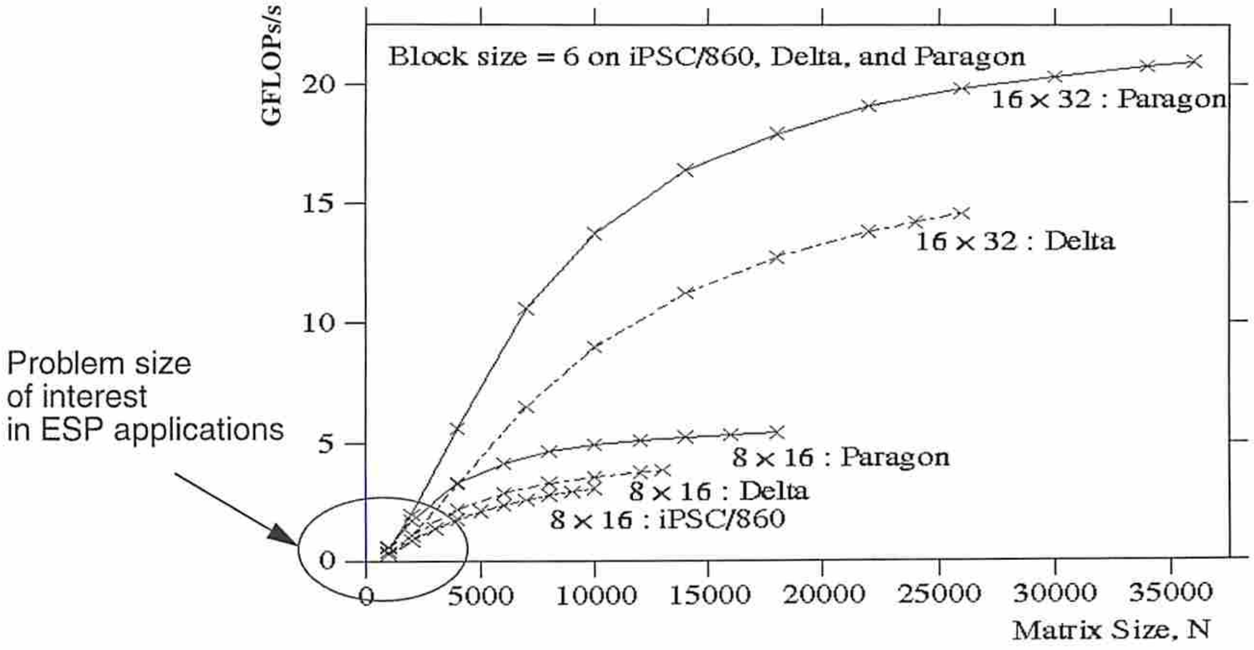


Figure 6: QR factorization on the Intel iPSC/860, Delta, and Paragon [5]

Previously, parallel linear algebra libraries such as ScaLAPACK have been used to develop signal processing applications on HPC platforms. In such a parallel library based approach, each task (ex. a QR decomposition) is partitioned and performed on several processors. However, in ESP applications, the matrices to be decomposed are relatively small. A typical task in such applications requires only few KFLOPs/s to few MFLOPs/s computational power. Practically, each processor on a HPC platform can perform several such tasks. Thus, the above library based approach is not suitable for parallelizing embedded applications. In [5], the ScaLAPACK library has been implemented to perform linear algebra operations such as LU or QR decomposition and Cholesky Factorization. The performance of the implementations of QRD on various HPC platforms is shown in Figure 6 [5]. The experimental results show that the efficiency of ScaLAPACK drops significantly when the problem size is small. For instance, ScaLAPACK can achieve about 21GFLOPs/s on 16×32 node Paragon for a matrix of size $35,000 \times$

35,000. However, if the matrix size is less than $1,000 \times 1,000$ (which is of interest in ESP applications), only 1 GFLOPs/s is sustained using ScaLAPACK. This is due to large communication overhead.

The key algorithmic advance needed to achieve scalable performance and high efficiency for ESP applications is to use *coarse-grain parallelism* (to exploit parallelism at task or job level) to reduce communication overhead. This suits the computational features of ESP applications which consist of a large number of relatively small tasks. When coarse-grain parallelism is exploited, the efficiency of the resulting application software depends mainly on the data layout of the stages and task mapping. These issues are addressed in Section 4.4 and Section 4.5 respectively.

4.2 Portable Algorithms Using Message Passing

State-of-the-art HPC platforms offer environment that includes portable high-level programming languages (such as C, C++, High Performance Fortran (HPF [8]), among others) and software standard libraries (such as Message Passing Interface (MPI/MPI-2 [19])) for implementing parallel algorithms. These platforms also provide portable serial libraries for efficient computation. For instance, Linear Algebra PACKage (LAPACK) [11] provides library subroutines to perform matrix operations such as LU and QR decomposition on uniprocessors. These languages and libraries provide an abstraction for representing concurrent activities as well as for interaction among these activities. The use of these portable languages and standard libraries enables software portability and legacy software re-use. This can, therefore, reduce the time to design and implement application software on HPC platforms.

When ESP applications are parallelized, the variation in the time performance is a critical issue since such applications are performed on-the-fly. The message passing programming paradigm has been widely used for developing such parallel algorithms. In this paradigm, explicit interaction among the concurrent activities is performed using, for example, "send" and "receive" commands. This paradigm, although considered to be tedious and can be error-prone, allows the programmer explicit control over the communication activities. Thus, the variation in the time performance of the resulting ESP application software can be made manageable and predictable.

Figure 7 illustrates portable algorithm design using a serial language (such as C) compiler and MPI. Algorithms specified in such a notation can be executed on various HPC platforms such as SP-2, T3E, and Paragon, among others. A parallel code for this example is shown in Figure 8. This code is compiled and dispatched so that each processor can execute it with a private copy of the code using its input data. Portable library subroutines for sequential operations (such as FFT and QRD) can be used

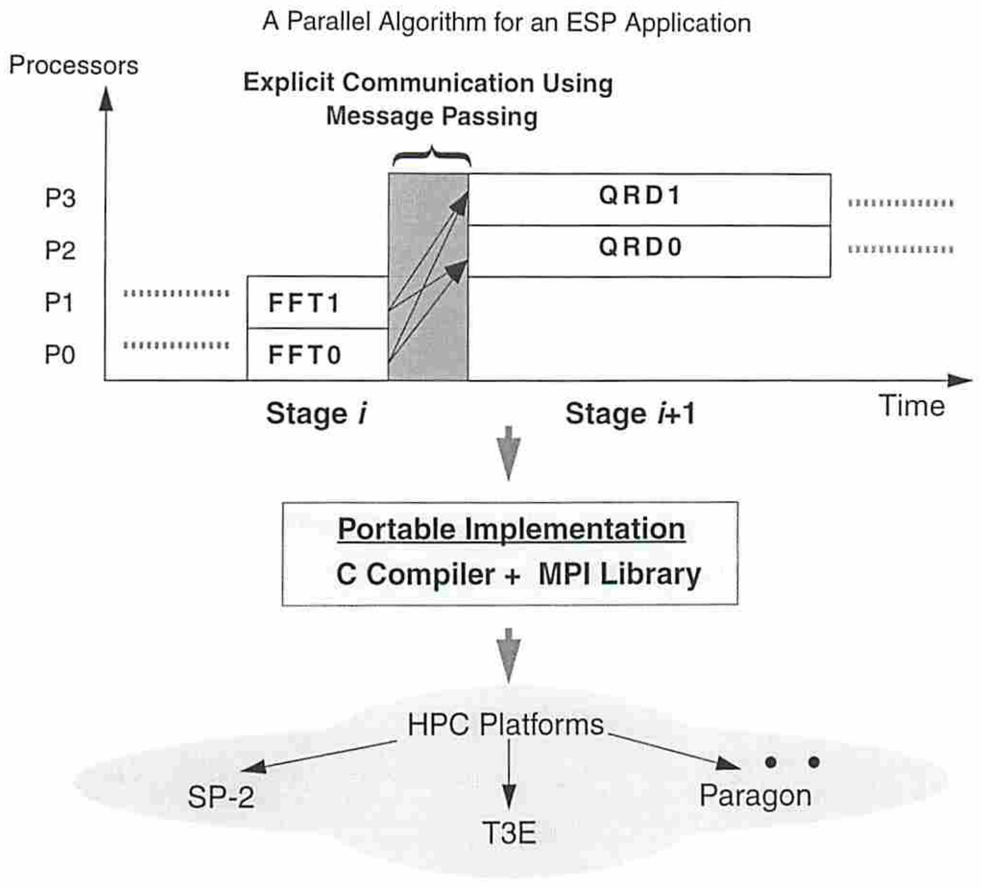


Figure 7: Portable algorithm design using C and MPI

since each such operation is performed in a single processor. Concurrent activities and communication among them can be implemented using MPI. The **MPI_Comm_rank** command specifies an identifier number, my_id, for each processor (P0 through P3 in this example). Each processor executes the tasks mapped onto it. These processors operate independently until they encounter communication or barrier synchronization. In Figure 8, processor P0 (P1) performs FFT0 (FFT1). Then P0 (P1) sends the results of FFT0 (FFT1) to P2 and P3 respectively. Processor P2 (P3) receives these results as input for the execution of QRD0 (QRD1). Note that, in our task model, the tasks in a stage (ex. FFT0 and FFT1) access disjoint subsets of the input data set during their execution and hence, they are data independent. The MPI point-to-point communication commands, **MPI_Send** and **MPI_Recv**, are used to realize communication between the sets of processors allocated to Stage i and Stage $i + 1$.

```
ESP_Example()
{

MPI_Comm_rank(MPI_COMM_WORLD, &my_id );

if(my_id==P0) { /* Execute tasks mapped onto P0 */
 .........
 /* FFT0 */
 FFT(...);
 MPI_Send(...,P2, ...);
 MPI_Send(...,P3, ...);
 .........
 }

else if(my_id==P1) { /* Execute tasks mapped onto P1 */
 .........
 /* FFT1 */
 FFT(...);
 MPI_Send(...,P2, ...);
 MPI_Send(...,P3, ...);
 .........
 }

else if(my_id==P2) { /* Execute tasks mapped onto P2 */
 .........
 /* QRD0 */
 QRD(...);
 MPI_Recv(...,P0, ...);
 MPI_Recv(...,P1, ...);
 .........
 }

else if(my_id==P3) { /* Execute tasks mapped onto P3 */
 .........
 /* QRD1 */
 QRD(...);
 MPI_Recv(...,P0, ...);
 MPI_Recv(...,P1, ...);
 .........
 }
}
```

Figure 8: Parallel code for an example ESP application using C and MPI

4.3 Computational Model

A computational model is an analytical representation of a computer system. The performance of an algorithm-architecture pair can be estimated using such a computational model. Recently, the General-purpose Distributed Memory (GDM) model has been used in developing ESP applications on HPC platforms [14, 18]. Explicit message passing is assumed in the GDM model for communication among the processors. This model can be used to choose appropriate algorithms to minimize the communication time when an ESP application is parallelized.

Current HPC platforms employ high bandwidth low latency interconnection networks. In these platforms, large software overheads are incurred as a message traverses through various software layers and the network interface to its destination [18]. The software overheads dominate the hardware switch latencies in current HPC platforms. For algorithm design and analysis, the networks of such platforms can be modeled as a "flat" network (i.e. its connection forms a complete graph and hence, the communication time is topology independent). Figure 9 depicts user level communication in such HPC platforms.

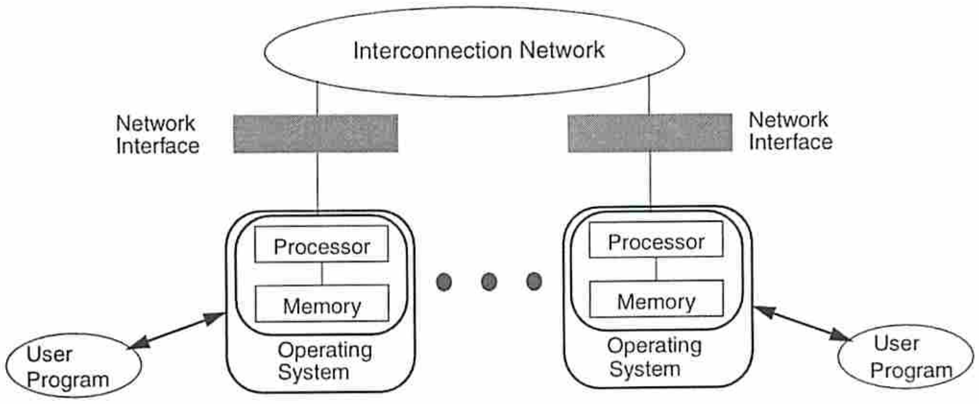


Figure 9: Communication in HPC platforms

To model such communication, the GDM model uses two parameters, T_s and τ_d, to represent the communication time [16]. T_s denotes the start-up cost which includes the software and communication protocol overheads. τ_d denotes the unit data transmission time over the network. In this model, point-to-point communication of a message (of size m) between a pair of processors is assumed to take $T_s + m\tau_d$ time. A permutation of data elements among the processors, assuming that each node has m units of data for another node, also takes $T_s + m\tau_d$ time.

Table 3 shows the observed values of the above parameters for various HPC platforms. The values of

T_s and τ_d shown in Table 3 were obtained based on our measurements on SP-2 using MPICH and T3E using MPI(EPCC). On the Paragon, values were obtained using Basic Linear Algebra Communication Subprograms (BLACS) [3]. It should be noted that these numbers vary depending on the version of the software environment used for message passing.

An example illustrating the use of the model is discussed in Section 4.4.

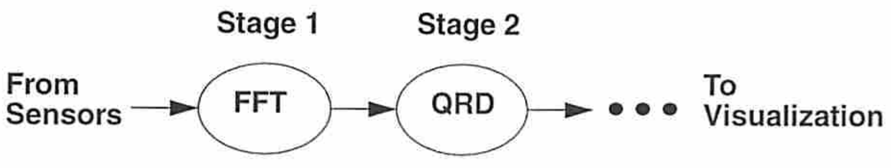
Table 3: Communication features of some HPC platforms

Machine	T_s(μsec)	τ_d (μsec/byte)	Bandwidth $1/\tau_d$ (MB/s)
SP-2	39 ~ 46	0.035	28.57
T3E	28	0.006	166.67
Paragon	37.7	0.0075	133.60

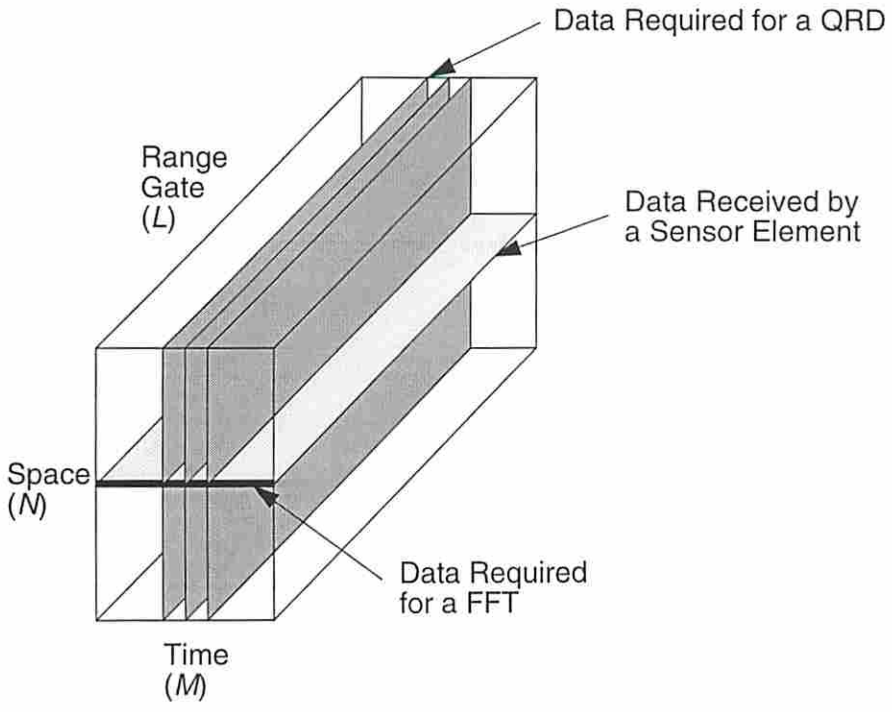
4.4 Data Layouts

The choice of data layouts among the processing stages strongly influences the performance of the application software running on HPC platforms [15, 16]. When an ESP application is parallelized, the data access patterns usually change from stage to stage. One approach to developing parallel software is to fix the data layout during the entire execution. Using this approach, a single task (e.g. a FFT or a QRD) may require data stored in several processors. This results in large communication overhead due to remote data access. An alternate approach is to change the data layout between successive stages so that each processor contains all the data needed for executing a task. Thus, coarse-grain parallelism can be exploited, i.e., each task is executed within a single processor.

In the following, the importance of choosing efficient data layouts is illustrated by a Space Time Adaptive Processing (STAP) application. This application performs adaptive filtering in both spatial and temporal domains. Figure 10 (a) shows the computationally demanding stages (FFT and QRD stages) of this application. The input data set can be represented using a data cube as shown in Figure 10 (b). The data cube is accessed along the space domain for performing FFTs. It is, then, accessed along the time domain for performing QRD's. In this figure, N denotes the number of delays (the time dimension), M denotes the number of channels (the space dimension), and L denotes the number of snapshots (the range gate dimension).



(a) Processing stages of a STAP application



(b) STAP data cube and the data access pattern of each processing stage

Figure 10: A STAP application

An earlier approach for implementing the above application [21] developed a parallel algorithm that uses a fixed data layout throughout the computation and hence does not perform any data remapping. The experimental results in [21] show that the communication time of their implementation increases rapidly as the number of processors increases. On a 32 node SP-1, the communication time is more than 50% of the total execution time [21]. The remote data access overhead severely degrades the overall performance and the speed-up tapers off.

In contrast, our work [15] performs the above application using data remapping between adjacent stages. Each task is performed on a single processor. Data layout is changed before executing tasks in Stage 2 so that there is no remote data access when QRD tasks are performed.

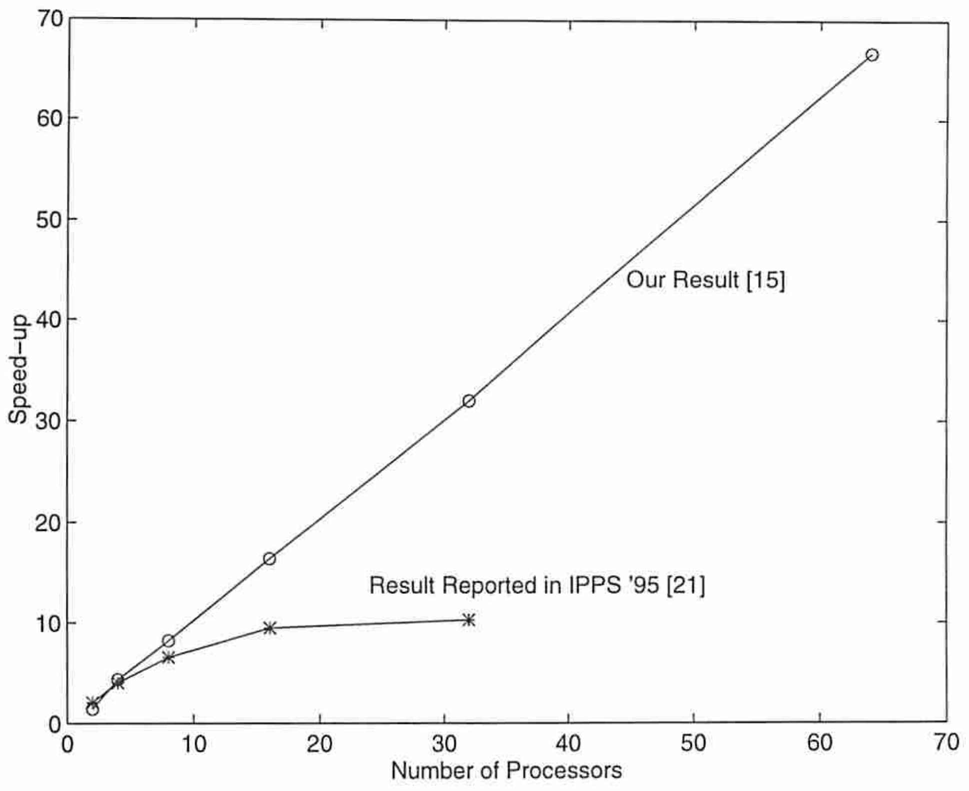


Figure 11: Parallel implementation of Higher-Order Post-Doppler (HOPD) STAP on IBM SP

Figure 11 compares the speed-ups achieved by these two parallel implementations. In these experiments, following parameters were used: N=64, M=64, and L=4096. Linear speed-up was achieved using data remapping. The sub-linear speed-up of the implementation in [21] is due to the communication overhead encountered in performing parallel QRD operations.

The GDM model defined in Section 4.3 can be used to estimate the cost of remapping in this example. Let N=64, M=64, and L=4096. Each data item is assumed to be 8 bytes (single precision, complex numbers). Initially, the data cube shown in Figure 10 (b) is partitioned among the available processors along the space dimension. Then, data is re-mapped (fed back) to these processors so that the data cube is partitioned along the time dimension. Figure 12 shows the estimated communication time for data remapping using 4 to 64 processors on a SP-2 using our model. The values of the message start-up cost and the unit data transmission time shown in Table 3 is used in our analysis.

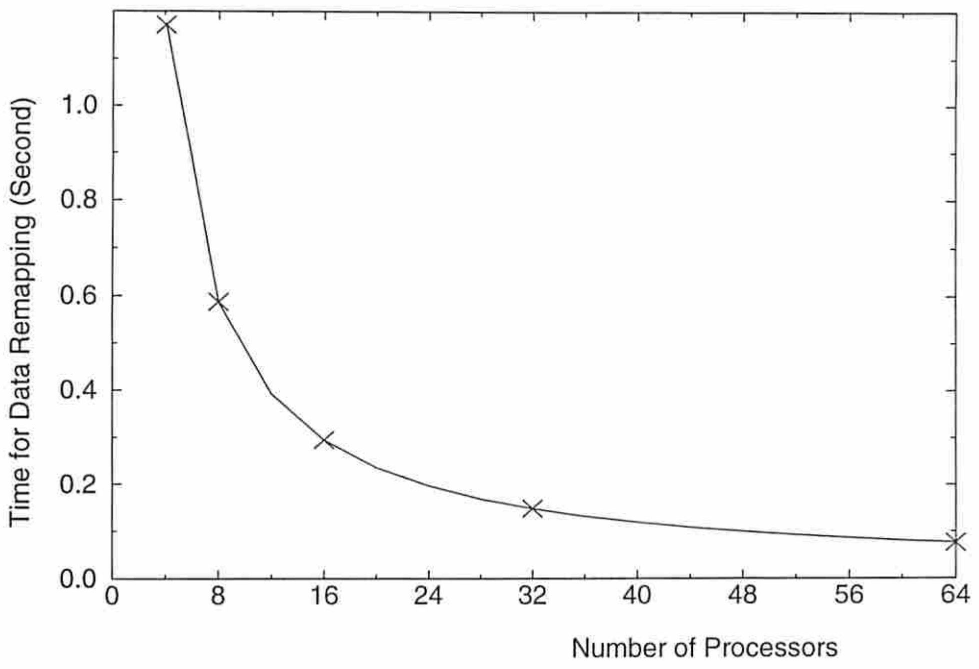
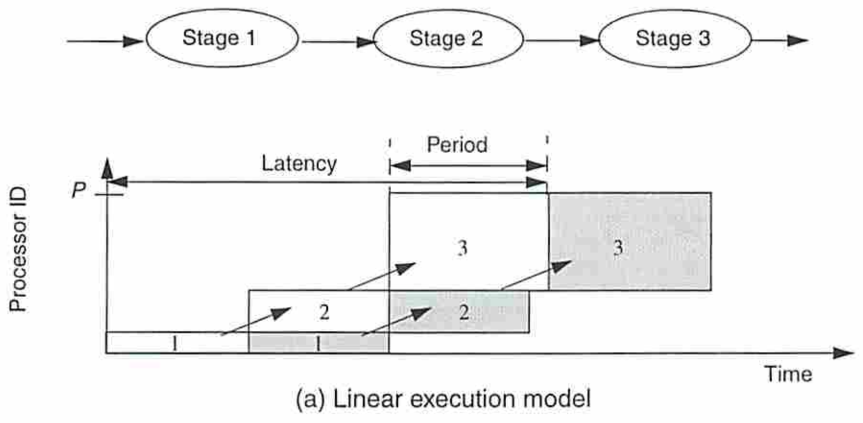


Figure 12: Cost of data remapping between FFT and QRD stages in HOPD STAP on SP-2

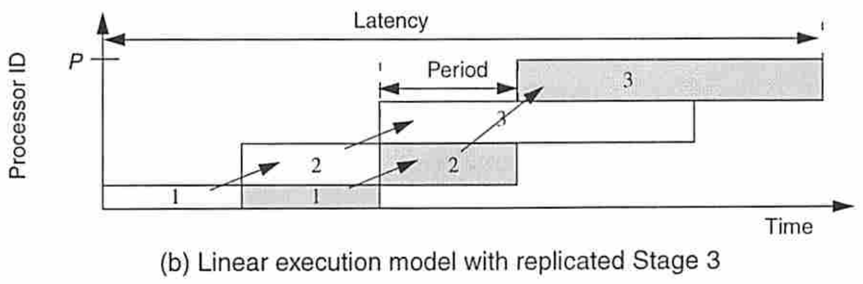
4.5 Task Mapping

Task mapping refers to the assignment of the tasks in an application to the available processors in a HPC platform. Most signal processing problems have regular computation and communication characteristics that are known at compile time. This permits static task mapping (i.e. task mapping is determined at compile time).

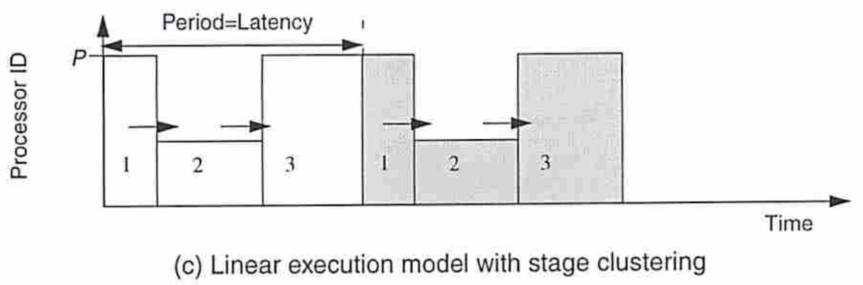
Choudhary et. al. [1] considered an optimal processor assignment problem when a sequence of data sets is processed by a collection of processing stages (called tasks in their notation). The problem is to optimize the latency (throughput) with a given throughput (latency) requirement. They use a task model in which the inter-task data dependence forms a series-parallel partial order. They also assume that each stage is mapped onto a disjoint set of processors, and each stage can be parallelized. The execution time of a stage is given as a function of the size of the processor set allocated to that stage. When an ESP application with a linear chain of stages is parallelized using this approach, the execution can be abstracted as a *linear execution model.* In this model, each processing stage is mapped to a disjoint set of processors (a pipeline stage). The processed data from a stage is fed forward to the nodes allocated to the next stage. Using this model, data is processed in a *pipelined* fashion. Thus, concurrency among the processing stages can be exploited. Figure 13 (a) shows an example task mapping using the linear execution model.



(a) Linear execution model



(b) Linear execution model with replicated Stage 3



(c) Linear execution model with stage clustering

Figure 13: Examples of linear execution models

The linear execution model has also been employed in [23, 24]. The work in [23, 24] considers task mapping for applications that consist of a sequence of processing stages (called data parallel tasks in their notation). The execution time of a stage is assumed to be a function of the number of processors allocated to that stage. Also, the communication time between two adjacent stages is assumed to be a function of the number of processors allocated to the two stages. A dynamic programming approach is used for optimizing the throughput (or latency) based on this model. Algorithmic techniques including *stage clustering* and *replication of pipeline stages* have been proposed for task mapping [23, 24].

The *replication of pipeline stages* technique is used to improve the performance of an application software using the linear execution model. In this technique, a stage is replicated such that a sequence of data sets can use the replicated instances of a stage in a round robin fashion. Figure 13 (b) illustrates this technique. Here, Stage 3 is replicated. There are two reasons for using this technique. In ESP applications, the computational requirements of adjacent stages can be significantly different. To equalize the throughput of those pipeline stages, the number of nodes allocated to those stages may vary from one another. This may cause large communication overhead between these stages. Replication of pipeline stages can be used so that the number of nodes allocated to adjacent stages is balanced. Some computationally demanding stages may have a low degree of parallelism. The processors allocated to such a stage may not be effectively utilized. Replication of pipeline stages can be used such that a data set is processed by a subset of these nodes. This can improve the processor utilization. For a given number of processors, replication of pipeline stages can increase the system throughput; however, it results in larger latency compared with the case where replication is not employed.

The *stage clustering* technique can also be used to improve the performance of linear pipelines. This technique clusters adjacent stages into a module. Such a module is mapped onto a set of processors. If the clustered stages have the same data layout, data transfer between these stages is not needed. This reduces the communication overhead compared with the case when these stages are mapped onto disjoint sets of processors. On the other hand, if the data layout changes between adjacent stages, the processed data in a stage is fed back (re-mapped) to the same set of nodes for the execution of the next stage. Note that, when all the processing stages are clustered, it results in a *feedback* software task pipeline (as shown in Figure 13 (c)). In this approach, at any time, only one processing stage is executed by all the nodes. Thus, parallelism within a stage is exploited. This approach is desirable when low latency is the main concern. However, this approach can result in low utilization of the available computational power if there is not enough parallelism (independent parallel tasks) in a processing stage.

A special case of task mapping is obtained when all the computation stages are clustered into a single module and mapped onto a single processor. By replicating the same design, the throughput can be made to scale with the number of processors. However, the design has several disadvantages: 1) the resulting latency can be too high. 2) the input data collected by a number of sensor elements corresponding to an initiation must be sent to a single processor. This incurs large communication overhead due to the limited bandwidth. 3) The entire input data must be stored in a single node. If the main memory is not large enough, the performance will suffer due to excessive disk access. Embedded systems are constrained in size, power, and weight. The amount of memory in such systems is limited. Therefore, this design is not considered further in this paper.

In the linear execution model discussed above, all the tasks of a stage are mapped to a disjoint set of processors. Besides, any two adjacent stages are executed either in two disjoint sets of processors or executed in the same set of processors (when clustering is used). Thus, the linear model results in restricted task mapping. In Section 5, we describe a novel task mapping methodology based on a new execution model.

5 A Design Methodology

In this section, we describe a task mapping methodology for designing ESP application software. We consider the problem of maximizing the throughput of the resulting software task pipeline given a fixed number of processors. Since the resulting application software is performed in a pipelined fashion, it is called a *Software Task Pipeline* (STP). This section is based on our research described in [12, 15, 16].

First, we define an execution model for developing the design methodology.

5.1 A New Execution Model

Based on the task model defined in Section 3.1, we define an execution model for mapping an ESP application onto a HPC platform. In the linear execution model [23], each processing stage is mapped onto a disjoint set of processors. In our execution model, a stage can be partitioned. A subset of tasks in a stage can be mapped onto the processor set allocated to one of its adjacent stages. Note that each task is mapped onto exactly one processor in our execution model. Therefore, only task level, coarse-grain parallelism is exploited. Stage partitioning is used to exploit independent activities within each processing stage. This technique relaxes the restricted choices for task mapping allowed in the linear execution model.

In Figure 14, the linear execution model and our new execution model are illustrated using a two-stage application. Five processors, P0 through P4, are used. There are 6 parallel tasks in Stage 1 and 19 parallel tasks in Stage 2. On a single processor, the execution time of a task in Stage 1 (Stage 2), denoted as c_1 (c_2), is 3 seconds (2 seconds). A task mapping assuming a linear execution model is shown in Figure 14 (a). The system throughput is determined by Stage 2, since it has the longest execution time (T_2). Using our execution model, the throughput can be improved by partitioning the stage with the longest execution time. In this example, Stage 2 is partitioned: one task in Stage 2 is re-distributed to processor P0, while the remaining 6 tasks are still mapped onto P2 through P4 (see Figure 14 (b)). This results in a shorter period (thus, higher throughput). Our execution model also

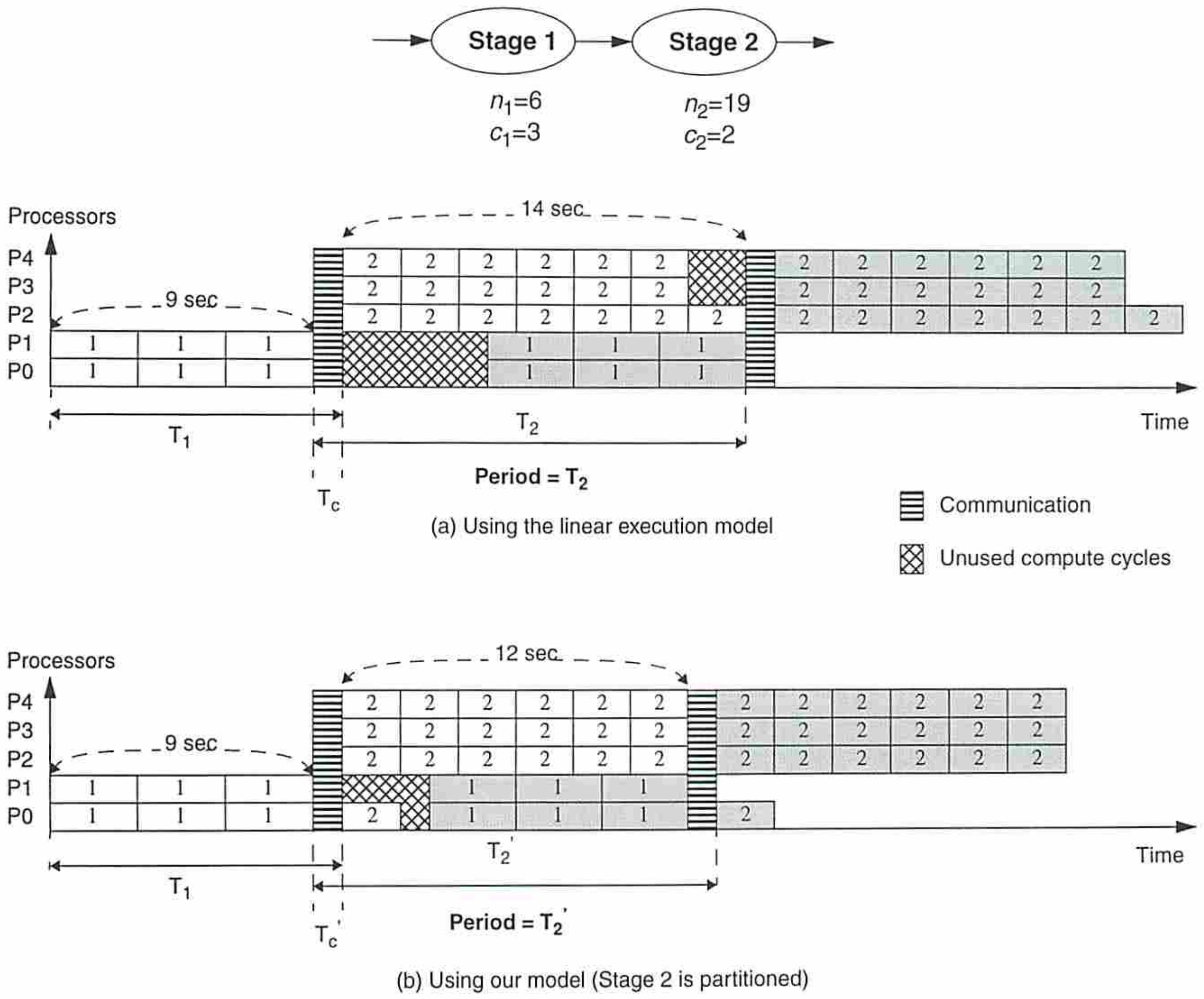


Figure 14: An illustrative example showing task mapping using the linear execution model and our model

considers the inter-stage communication as shown in Figure 14. Note that $T_c^{'}$ can be smaller than T_c. Indeed, in the mapping shown in Figure 14 (b), the amount of data communicated from P0 and P1 to P2, P3, and P4 is smaller than in the case of Figure 14 (a).

5.2 A Three-Step Task Mapping Methodology

Based on our execution model, we describe a three-step task mapping methodology for maximizing the throughput of an ESP application. The application to be parallelized is represented using our task model defined in Section 3.1. The details of these steps are described in the following subsections. In Section 5.3, this methodology is illustrated using an example.

5.2.1 Step 1: Data Remapping

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

Data remapping is a critical 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 their 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* [15, 16] 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 data elements to be sent to the same destination.

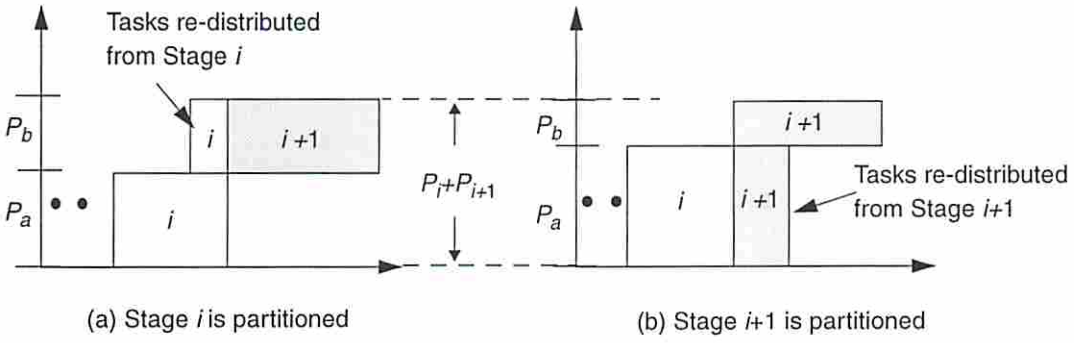
5.2.2 Step 2: Coarse Resource Allocation

This step performs an initial task mapping to generate a linear software pipeline. The processing stages are mapped onto disjoint sets of the available processors. A task in each processing stage is executed on a single processor. Its execution time (i.e., t_i for each task in Stage i) is estimated based on its computational complexity, c_i, and the computing power of each processor. Then, the number of processors allocated to Stage i is determined in proportion to the time complexity of that stage (i.e., $n_i \times t_i$). If P_i processors are allocated to Stage i and $P_i > n_i$, then, Stage i is replicated $\lfloor \frac{P_i}{n_i} \rfloor$ times. Each replicated stage has n_i processors. The remaining unused processors (i.e., $P_i - n_i \times \lfloor \frac{P_i}{n_i} \rfloor$ processors) are designated as free processors. Replication of pipeline stages guarantees that sufficient degree of parallelism exists in each stage.

After the initial task mapping, the free processors are allocated to a bottleneck stage (a stage with the longest execution time) in this step.

5.2.3 Step 3: Fine Performance Tuning

In Step 3, the performance of the linear software pipeline obtained in Step 2 is improved using our execution model. This step realizes stage partitioning by using a heuristic algorithm. This heuristic reduces the period of the pipeline in an iterative manner. In each iteration, a bottleneck stage is identified. Let this stage be Stage i. Based on our execution model, the algorithm locally minimizes the period of the pipeline by 1) re-assigning the processors allocated to stage i and Stage $i+1$ (Stage $i-1$), and 2) re-distributing the tasks in Stage i and Stage $i+1$ (Stage $i-1$) to minimize the period. A new task mapping which results in the largest reduction of the period is chosen. This step is repeated until the period cannot be further reduced.



Tasks re-distributed from Stage i
P_b
P_a
i
$i+1$
P_i+P_{i+1}
Tasks re-distributed from Stage $i+1$
(a) Stage i is partitioned
(b) Stage $i+1$ is partitioned

Figure 15: Examples of stage partitioning

Figure 15 shows stage partitioning between Stage i (the bottleneck stage) and Stage $i+1$. Let P_i and P_{i+1} denote the number of processors allocated to Stage i and Stage $i+1$ respectively in Step 2. These processors are re-assigned. For instance, in Figure 15, P_a and P_b processors are allocated to Stage i and Stage $i+1$ respectively. There are $P_i + P_{i+1}$ ways to re-assign these processors. Thus, the complexity of re-assigning processors is $O(P)$, where P is the total number of processors allocated to this application. For a given P_a and P_b, the period of the software pipeline is minimized by partitioning Stage i or Stage $i+1$ as shown in Figure 15. Note that, if the stage to be partitioned is replicated in Step 2, the same number of tasks from each such replicated stage is considered for re-distribution. Thus, the resulting periods of these replicated stages remain the same. The time complexity of one iteration of stage partitioning is $O(P)$.

5.3 An Illustrative Example

In the following, the effectiveness of the above mapping methodology is illustrated by an example. Consider a two-stage application that is mapped onto 6 processors. Let $n_1 = 5, n_2 = 7$. Let $c_1 =$ 1 second and $c_2 = 1$ second. For the sake of illustration, we assume that the communication cost between the adjacent stages to be zero. Step 1 determines the data layouts of each of the stages and the data remapping to be performed between adjacent stages. In this step, the communication

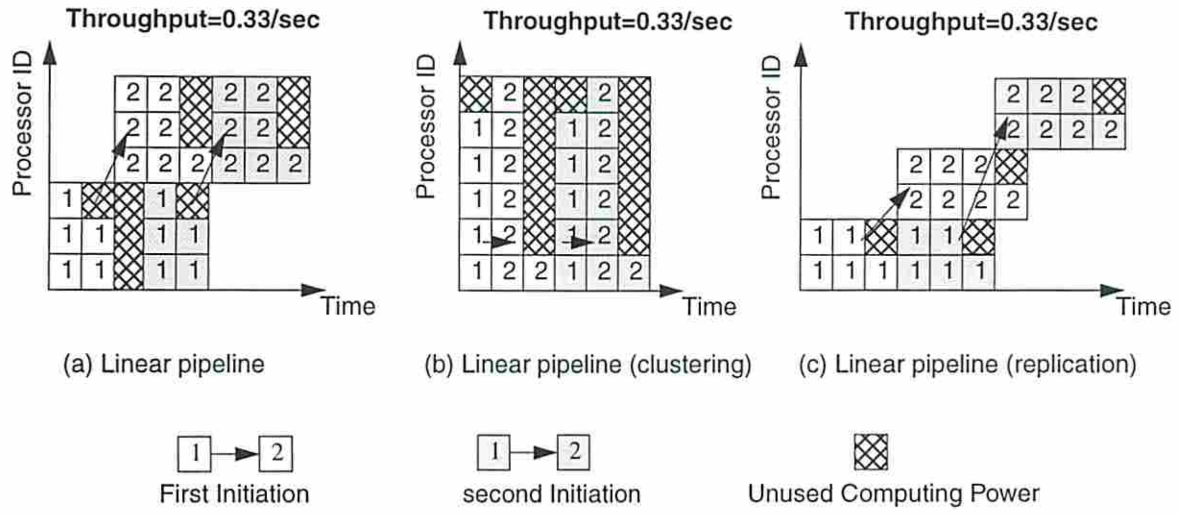


Figure 16: Various software task pipelines using the linear execution model (with clustering and replication)

between the stages is explicitly performed by remapping the data. As discussed in Section 4.5, the solution that maps all the 12 tasks in one initiation onto a single processor is not considered. Based on the linear execution model used in [1], the "optimal" throughput achieved is 0.33/sec (as shown in Figure 16 (a)). Clustering or replication, as proposed in [23], does not improve the throughput. Two "optimal" mappings using clustering and replication are shown in Figure 16 (b) and (c). Using our mapping methodology, the throughput can be significantly improved. The coarse resource allocation in Step 2 results in an initial mapping such that three processors are allocated to each stage. Then, fine performance tuning is performed. Figure 17 shows possible processor allocations and task redistributions using stage partitioning.

The optimal throughput occurs when P_a=5 and P_b=1 as shown in Figure 17. In this case, Stage 2 is partitioned. Five of the tasks in Stage 2 are mapped to the 5 processors allocated 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 $\frac{P}{n_1 c_1 + n_2 c_2} = 0.5$. Thus, for this example, our mapping methodology results in optimal throughput performance.

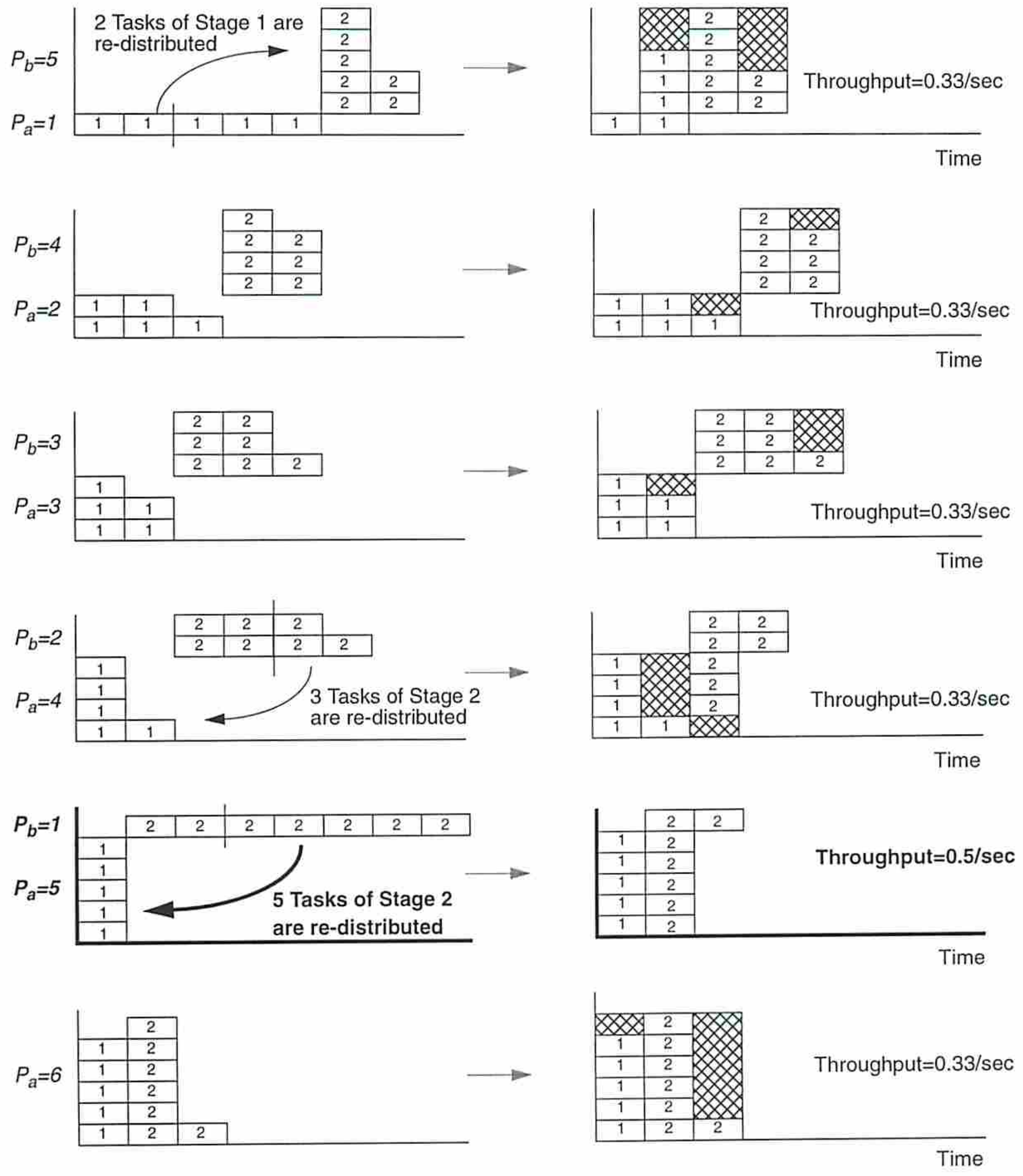


Figure 17: Illustration of stage partitioning for a two-stage application

6 A Case Study: Parallelization of a Frequency-Domain, Fully Adaptive Sonar Beamformer

We illustrate various implementations for an adaptive sonar beamforming benchmark on an IBM SP-2. This sonar benchmark performs adaptive beamforming in frequency domain. The incoming time-domain signals are first converted to frequency domain using FFTs. Then, the resulting frequency-domain signals are linearly combined with fully adaptive weight matrices. Details of this benchmark can be found in [13, ?]. The resulting implementations are denoted $A1$ through $A3$. Both $A1$ and $A2$ use the linear execution model defined in Section 4.5. However, no stage clustering or stage replication is used in $A1$. On the other hand, $A2$ allows clustering and replication of pipeline stages. $A1$ and $A2$ are the best performance that can be achieved by using techniques in [1] and [23] respectively. In [23], it is not stated whether data remapping is explicitly used or not. Thus, $A1$ (as well as $A2$) can use a fixed data layout throughout the computation. Since data remapping can significantly improve performance, the performance improvement using $A3$ can be due to the use of data remapping. To expose the performance improvement using our task model and stage partitioning, we also perform data remapping between adjacent stages for both $A1$ and $A2$. Therefore, in the following, the observed performance improvement can be attributed to various task mapping techniques alone.

Two experiments were performed on an IBM SP-2. In Experiment 1, the beamformer uses 64 sensor elements to sample the signals. 64 frequency bins were chosen with 128 beams formed in each frequency bin. In Experiment 2, the beamformer uses 128 sensor elements to sample the acoustic signals. 128 frequency bins were chosen with 128 beams formed in each frequency bin. See Table 4 for details.

Table 4: Characteristics of the two MVDR sonar beamformers

Processing stage	Functionality	Experiment 1		Experiment 2	
		T (μsec)	N	T (μsec)	N
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	16,384

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

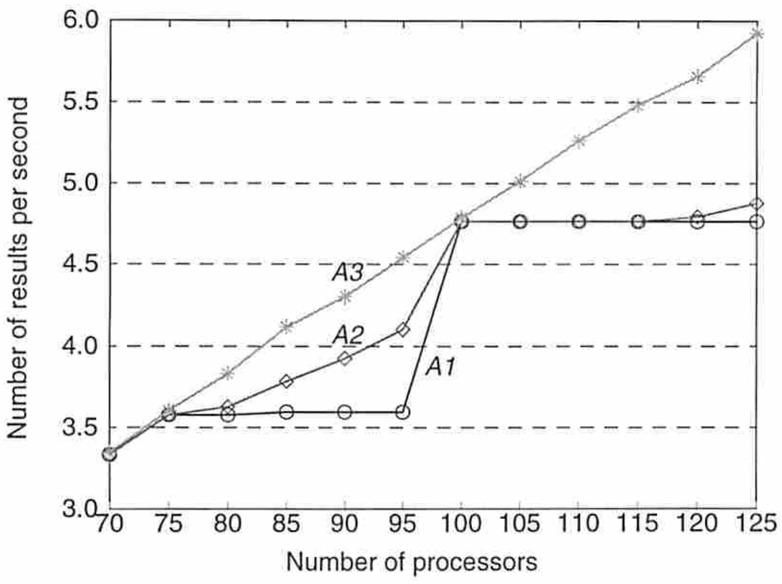


Figure 18: Performance Comparison of $A1$ through $A3$ in Experiment 1

Figures 18 and 19 show the throughput performance on an IBM SP-2. The number of processors in these experiments is varied from 70 to 125 (in increments of 5 processors). In Experiment 1 (2), a practical latency constraint of 1 (10) second(s) is assumed to meet the real-time processing requirement. In Experiment 1, when 75 or 100 processors are used, $A3$, as well as $A1$ and $A2$ can effectively utilize the computing power based on their corresponding execution models. The resulting throughput, in these cases, is about the same using $A1$ through $A3$. However, in most of the other cases, the resulting software pipelines using $A1$ and $A2$ are not scalable. In these cases, our design methodology results in superior performance over $A1$ and $A2$. For instance, in Experiment 1 (2) our approach increases the throughput by approximately 24.3% (8.2%) compared with $A1$ when 125 processors are used. This figure also shows that our approach increases the throughput by approximately 21.4% (29.3%) compared with $A2$ when 125 processors are used.

7 Concluding Remarks

This paper considered algorithmic issues in mapping ESP applications onto homogeneous HPC platforms. It proposed a simple task model to represent ESP applications to facilitate task mapping. A general execution model was proposed to effectively utilize the available computing power in HPC platforms. There are a number of important research areas that are worthy of exploration:

1. Many of the embedded HPC platforms integrate various COTS components with heterogeneous

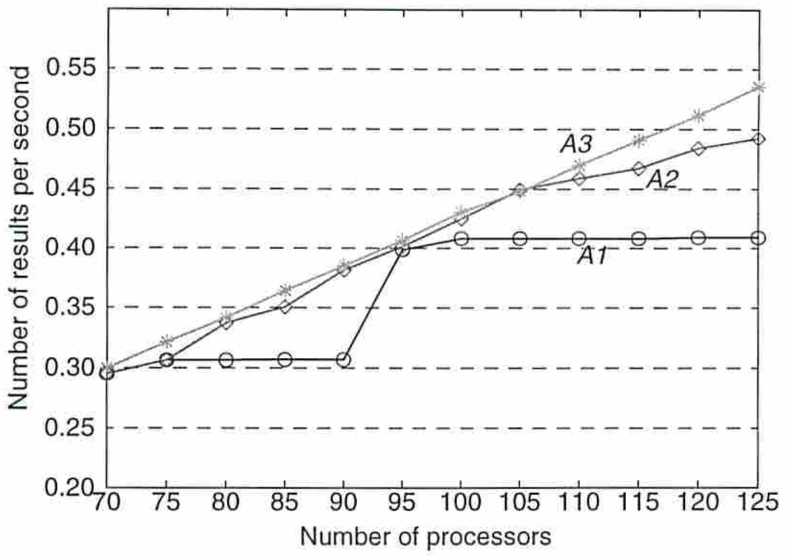


Figure 19: Performance Comparison of $A1$ through $A3$ in Experiment 2

computational features. 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. We focussed on scalable and portable algorithm design; issues related to size, weight, and power were completely ignored in the model. Additional work is needed to develop a comprehensive framework to aid system designers in developing ESP applications.

3. During the past few years, several new architectures have been proposed and implemented for signal 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 a sonar application to illustrate our ideas. Our related results in using HPC for signal processing can be found in [25, 26, 17].

8 Acknowledgments

We would like to thank Young-Won Lim and Myungho Lee for helpful discussions.

References

[1] 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.

[2] R. Bernecky, "Sonar Beamforming Challenge Problems," presented at the DARPA/ITO Embeddable Systems PI Meeting, San Diego, June 1996.

[3] L. S. Blackford, et. el., "ScaLAPACK Users' Guide," published by the Society for Industrial and Applied Mathematics, ISBN 0-89871-397-8, May 1997.

[4] 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.

[5] J. Choi, J. Dongarra, S. Ostrouchov, A Petitet, D. W. Walker, and R. C. Whaley, "The Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines," in proceedings of Scientific Programming, Vol. 5, pp. 173-184, 1996.

[6] "Embeddable Systems Home page,"
URL: http://www.ito.arpa.mil/ResearchAreas/Embeddable.html.

[7] R. A. Games, "Benchmarking Methodology for Real-Time Embedded Scalable High Performance Computing," MITRE Technical Report MTR 96B0000010, March 1996.

[8] "High Performance Fortran Forum,"
URL: http://www.crpc.rice.edu/HPFF/home.html.

[9] K. Hwang and Z. Xu, "Scalable Parallel Computers for Real-Time Signal Processing," IEEE Signal Processing Magazine, July 1996.

[10] 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.

[11] E. Anderson, et. al., "LAPACK Users' Guide - Release 2.0,"
URL: http://www.netlib.org/lapack/lug/lapack_lug.html.

[12] 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.

[13] M. Leonhardt, "Implementation of Minimum Variance Distortionless Response (MVDR) Adaptive Beamforming Algorithm," NUSC Technical Document 8453, July 1989.

[14] Y. W. Lim and V. K. Prasanna, "Efficient Algorithms for General Block-Cyclic Redistribution," Technical Report, Department of EE-Systems, USC, Aug. 1996.

[15] 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.

[16] 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.

[17] Y. W. Lim, P. B. Bhat, and V. K. Prasanna, "Efficient Algorithms for Block-Cyclic Redistribution of Arrays," Algorithmica to appear.

[18] 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.

[19] "Message Passing Interface Standard,"
URL: http://www.mcs.anl.gov/mpi/index.html.

[20] J. S. McMahon, and K. Teitelbaum, "Space-Time Adaptive Processing on the Mesh Synchronous Processor," in proceedings of the 10th International Parallel Processing Symposium (IPPS '96), Apr. 1996.

[21] S. J. Olszanskyj, J. M. Lebak, and A. W. Bojanczyk, "Parallel Algorithms for Space-Time Adaptive Processing," in proceedings of the 9th International Parallel Processing Symposium (IPPS '95), Apr. 1995.

[22] A. Skjellum, "Embedded, Real-time MPI and MsgWay," URL: http://www.ito.arpa.mil/ResearchAreas/Embeddable.html.

[23] 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.

[24] 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.

[25] J. Suh and V. K. Prasanna, "Portable Communication Algorithms for Implementing SAR," First Annual High-Performance Embedded Computing Workshop, Lexington, MA, September 1997.

[26] 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.

[27] B. D. Van Veen and K. M. Buckley, "Beamforming: A Versatile Approach to Spatial Filtering," IEEE ASSP Magazine, Apr. 1988.

[28] 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.

[29] J. Ward, "Space-Time Adaptive Processing for Airborne Radar," Technical Report 1015, Massachusetts Institute of Technology, Lincoln Laboratory, Dec. 1994.