Appears in International Parallel Processing Symposium 1998

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

Alok Choudhary
ECE Department
Northwestern University
Evanston, IL 60208
email: choudhar@ece.nwu.edu

Wei-keng Liao, Donald Weiner, and Pramod Varshney
EECS Department
Syracuse University
Syracuse, NY 13244

Richard Linderman and Mark Linderman
Air Force Research Laboratory
Information Directorate
Rome, NY 13441

Abstract

This paper presents performance results for the design and implementation of parallel pipelined Space-Time Adaptive Processing (STAP) algorithms on parallel computers. In particular, the paper describes the issues involved in parallelization, our approach to parallelization and performance results on an Intel Paragon. The paper also discusses the process of developing software for such an application on parallel computers when latency and throughput are both considered together and presents tradeoffs considered with respect to inter and intra-task communication and data redistribution. The results show that not only scalable performance was achieved for individual component tasks of STAP but linear speedups were obtained for the integrated task performance, both for latency as well as throughput. Results are presented for up to 236 compute nodes (limited by the machine size available to us). Another interesting observation made from the implementation results is that performance improvement due to the assignment of additional processors to one task can improve the performance of other tasks without any increase in the number of processors assigned to them. Normally, this cannot be predicted by theoretical analysis.

1 Introduction

Space-time adaptive processing (STAP) is a well known technique in the area of airborne surveillance radars, which is used to detect weak target returns embedded in strong ground clutter, interference, and receiver noise. Data processing for STAP refers to a 2-dimensional adaptive filtering algorithm which attenuates unwanted signals by placing nulls in the frequency domain with respect to their directions of arrival and/or Doppler frequencies. Most STAP applications consume great amounts of computational resources and are also required to operate in real time. High performance computers are becoming mainstream due to the progress made in hardware as well as software support in the last few years. They can satisfy the STAP computational requirements of real-time applications while increasing the flexibility, affordability, and scalability of radar signal processing systems. However, efficient parallelization of STAP, which consists of several different algorithms is challenging, and requires several optimizations.

This paper describes our parallel pipelined implementation of a PRI-staggered post-Doppler STAP algorithm. The design and implementation of the application is portable. Performance results are presented for the Intel Paragon at the Air Force Research Laboratory (AFRL), Rome, New York. AFRL has successfully implemented this STAP algorithm onboard an airborne platform and performed four flight experiments in May and June 1996 [8]. In that real-time demonstration, live data from a phased array radar was processed by Intel Paragon machine and results showed that high performance computers can deliver a significant performance gain. However, that implementation only used compute nodes of the machine as independent resources in a round robin fashion to run different instances of STAP (rather than speeding up one instance of STAP.) Using this approach, the throughput may be improved, but the latency is limited by what can be achieved using one compute node. The algorithm consists of the following steps: 1)Doppler filter processing, 2)weight computation, 3)beamforming, 4)pulse compression, and 5)CFAR processing.

For our parallel implementation of this real application we have designed a model of parallel pipeline system where each pipeline is a collection of tasks and each task itself is parallelized. This parallel pipeline model was applied to the STAP algorithm with each step as a task in a pipeline. This permits us to significantly improve latency as well as

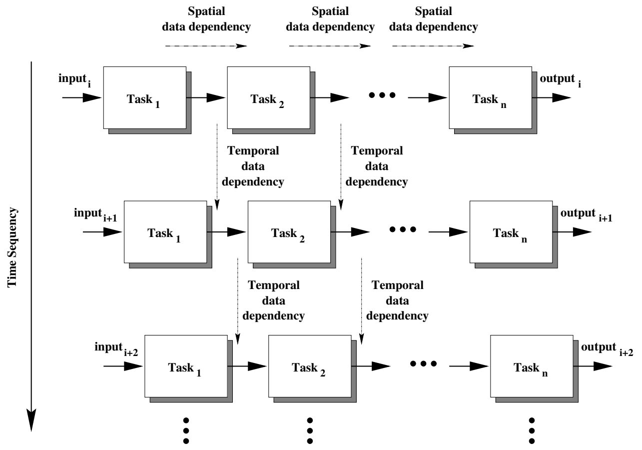

Figure 1. Model of the parallel pipeline system. (Note that $Task_i$ for all input instances is executed on the same number of processors.)

throughput. In this paper we present results from this implementation. Furthermore, we present the process of parallelization and software design considerations including those for portability, task mapping, parallel data redistribution, parallel pipelining and issues involving in measuring performance in implementations when not only the performance of individual tasks is important, but overall performance of the integrated system is critical. We demonstrate the performance and scalability for a large number of processors.

The rest of the paper is organized as follows: in Section 2, we present the parallel pipeline system model and discuss some parallelization issues and approaches for implementation of STAP algorithms. Section 3 presents the implementation. Performance results and conclusions are presented in Section 4 and Section 5 respectively.

2 Model of the parallel pipeline system

The system model for the type of STAP applications considered in this work is shown in Figure 1. This model is suitable for the computational characteristics found in these applications. A pipeline is a collection of tasks which are executed sequentially. The input to the first task is obtained normally from sensors or other input devices and the inputs to the rest of the tasks in the pipeline are the outputs of their previous tasks. The set of pipelines shown in the figure indicates that the same pipeline is repeated on subsequent input data sets. Each block in a pipeline represents one parallel task, which itself is parallelized on multiple (different number of) processors.

In such a system, there exist both spatial and temporal parallelism that result in two types of data dependencies and flows, namely, spatial data dependency and temporal data dependency [4, 6]. Spatial data dependency can be classified into inter-task data dependency and intra-task data dependency. Intra-task data dependencies arise when a set of subtasks needs to exchange intermediate results during the execution of a parallel task in a pipeline. Inter-task data dependency is due to the transfer and reorganization of data passed onto the next parallel task in the pipeline. Temporal data dependency occurs when some form of output generated by the tasks executed on the previous data set are needed by tasks executing the current data set. We will later see that STAP has both types of data dependencies.

2.1 Parallelization issues and approaches

Applications such as STAP entail multiple algorithms (or processing steps), each of which performs particular functions, to be executed in a pipelined fashion. Each task needs to be parallelized for the required performance, which, in turn, requires addressing the issue of data distribution on the subset of processors on which a task is parallelized to obtain good efficiency and incur minimal communication overhead.

2.1.1 Inter-task data redistribution

In an integrated system which implements several tasks that feed data to each other, data redistribution is required when it is fed from one parallel task to another. This is because the way data distributed in one task may not be the most appropriate distribution for another task for algorithmic or efficiency reasons. Data redistribution also allows concentration of communication at the beginning and the end of each task. We have developed runtime functions and strategies that perform efficient data redistribution [10]. These techniques reduce the communication time by minimizing contention on the communication links as well as by minimizing the overhead of processing for redistribution (which adds to the latency of sending messages). We take advantage of lessons learned from these techniques to implement the parallel pipelined STAP application.

2.1.2 Task scheduling and processor assignment

An important factor in the performance of a parallel system, is how the computational load is mapped onto the processors in the system. Ideally, to achieve maximum parallelism, the load must be evenly distributed across the processors. The problem of statically mapping the workload of a parallel algorithm to processors in a distributed memory system, has been studied under different problem models, such as [1, 2]. These static mapping policies do not model applications consisting of a sequence of tasks (algorithms),

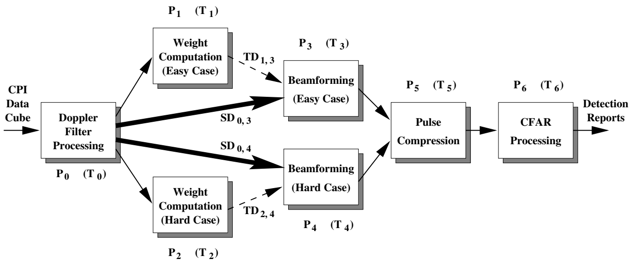

Figure 2. Implementation of parallel pipelined STAP. Arrows connecting task blocks represent data transfer between tasks.

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

Optimal use of resources is particularly important in high-performance embedded applications due to limited resources and other constraints such as desired latency or throughput [5]. When several parallel tasks need to be executed in a pipelined fashion, tradeoffs exist between assigning processors to maximize the overall throughput and assigning processors to minimize a single data set's response time (or latency.)

3 Design and implementation

The design of the parallel pipelined STAP algorithm is shown in Figure 2. The parallel pipeline system consists of seven basic tasks. We refer to the parallel pipeline as simply a pipeline in the rest of this paper. Both the weight computation and the beamforming tasks are divided into two parts, namely, "easy" and "hard" Doppler bins. The hard Doppler bins are those in which significant ground clutter is expected and the remaining bins are easy Doppler bins. The main difference between the two is the amount of data used and the amount of computation required. The input data set for the pipeline is obtained from a phased array radar and is formed in terms of a coherent processing interval (CPI). Each CPI data set is a 3-dimensional complex data cube. The output of the pipeline is a report on the detection of possible targets. Each task i, $0 \leq i < 7$, is parallelized by evenly partitioning its work load among P_i processors. The execution time associated with task i, T_i, consists of the time to receive data from the previous task, computation time, and time to send results to the next task.

For the computation of the weight vectors for the current CPI data cube, data cubes from previous CPIs are used as input data. This introduces temporal data dependency. Temporal data dependencies are represented by arrows with dashed lines, $TD_{1,3}$ and $TD_{2,4}$, in Figure 2 where $TD_{i,j}$ represents temporal data dependency of task j on data from task i. In a similar manner, spatial data dependencies $SD_{i,j}$ can be defined and are indicated in Figure 2 by arrows with solid lines.

Throughput and latency are two important measures for performance evaluation on a pipeline system. The throughput of our pipeline system is the inverse of the maximum execution time among all tasks. The latency of this pipeline system is the time between the arrival of the CPI data cube at the system input and the time at which the detection report is available at the system output.

$$throughput = \frac{1}{\max\limits_{0 \leq i \leq 6} T_i}. \tag{1}$$

$$latency = T_0 + \max_{i=3,4} T_i + T_5 + T_6. \tag{2}$$

The temporal data dependency does not affect the latency because weight computation tasks use data from the previous instance rather than current CPI. The filtered CPI data cube sent to the beamforming task does not wait for the completion of its weight computation. This explains why equation (2) does not contain T_1 and T_2. A detailed description of the STAP algorithm we used can be found in [3, 7].

4 Performance results

The implementation of the STAP application based on our parallel pipeline system model was done on the Intel Paragon at the Air Force Research Laboratory, Rome, New York. All the parallel programs development and their integration was performed using C language and message passing interface (MPI) [9]. This permits easy portability across various platforms which support C language and MPI. In our implementation, asynchronous send and receive function calls were used in order to overlap communication and computation.

4.1 Computation costs

The task of computing hard weights is the most computationally demanding task. The Doppler filter processing task is the second most demanding task. Naturally, more processors are assigned to these two tasks in order to obtain a good performance. For each task in the STAP algorithm, parallelization was done by evenly dividing computational load across processors. Figure 3 gives the computation performance results as functions of numbers of processors and the corresponding speedup on the AFRL Intel Paragon. For each task, we obtained linear speedups.

4.2 Inter-task communication

Inter-task communication refers to the communication between sending and receiving (distinct and parallel) tasks.

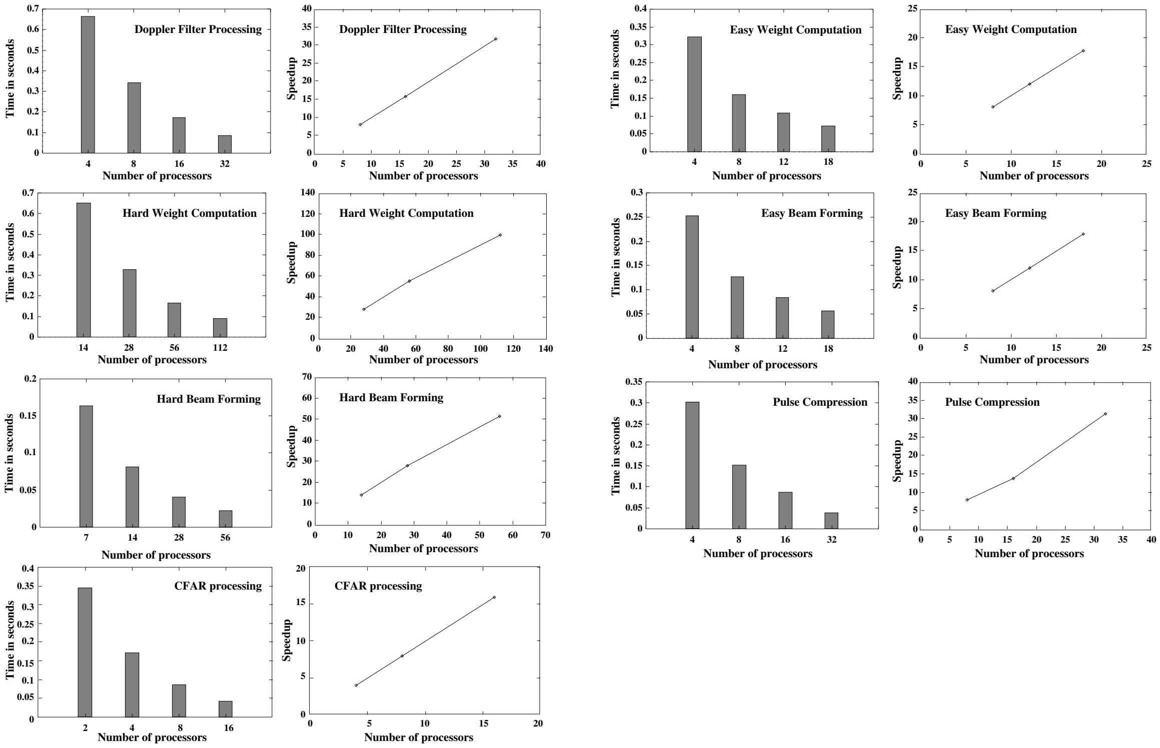

Figure 3. Performance of computation as a function of number of processors.

Table 1. Timing results of inter-task communication. Time in seconds. # proc: number of processors.

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

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

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

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

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

This communication cost depends on both processor assignment for each task as well as on the volume and extent of data reorganization. Table 1 presents the inter-task communication timing results. Each sub-table considers pairs of tasks where the number of processors (# proc) for both tasks are varied. In some cases timing results shown in the tables contain idle time for waiting for the corresponding task to complete. This happens when receiving task's computation part completes before the sending task has generated data to send.

From most of the results the following important observations can be made. First, when the number of processors

Table 2. Performance results for 3 cases with different processor assignments.

case 1: total number of processors = 236 — Time in seconds

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

case 2: total number of processors = 118 — Time in seconds

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

case 3: total number of processors = 59 — Time in seconds

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

is unbalanced, the communication performance is not very good. Second, as the number of processors is increased in the sending and receiving tasks, communication scales tremendously. This happens for two reasons. One, each processor has less data to reorganize, pack and send and each processor has less data to receive; and two, contention at sending and receiving processors is reduced. Thus, it is not sufficient to improve the computation times for such parallel pipelined applications to improve throughput and latency.

Because of the asynchronous send used in the implementation, the results shown here are visible sending time and the actual sending action may occur in other portions of the task. Similar to the receiving time, sending time may also contain waiting time for the completion of sending requests in the previous loop. With large number of processors, there is tremendous scaling in performance of communicating data as the number of processors is increased. This is because the amount of processing for communication per processor is decreased (as it handles less amount of data), amount of data per processor to be communicated is decreased and traffic on links going in and out of each

Table 3. Performance results for adding 4 more processors to case 2 in Table 2.

total number of processors = 122 — Time in seconds

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

Table 4. Performance results for adding 16 more processors to the case in Table 3.

total number of processors = 138 — Time in seconds

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

processor is reduced. This model scales well for both computation and communication.

4.3 Integrated system performance

Integrated system refers to the evaluation of performance when all the tasks are considered together. Throughput (CPIs per second) and latency (seconds per CPI) are the two most important measures for performance evaluation in addition to individual task computation time and inter-task communication time. Table 2 gives timing results for three different cases with different processor assignments. From these 3 cases, it is clear that even for latency and throughput measures we obtain linear speedups from our experiments. Given that this scale up is up to 236 processors (we were limited to these number of processors due to the size of the machine), we believe these are very good results.

As discussed in section 2, tradeoffs exist between assigning processors to maximize throughput and to minimize latency, given limited resources. Using two examples, we illustrate how further performance improvements may (or may not) be achieved if few extra processors are available. We now take case 2 from Table 2 as an example and add some extra processors to tasks to analyze its affect to the throughput and latency. Suppose that case 2 has fulfilled the minimum throughput requirement and more processors can be added. Table 3 shows that adding 4 more processors to Doppler filter processing task not only increases the throughput but also reduces the latency. This is because the communication amount for each send and receive between Doppler filter processing task to weight computation and to beamforming tasks is reduced (Table 3). So, clearly adding processors to one task not only affects that task's performance but has a measurable effect on the performance of other tasks. By increasing the number of processors 3%, the improvement in throughput is 32% and in latency is 19%. *Such effects are very difficult to capture in purely theoretical models because of the secondary effects.*

Since parallel computation load may be different among tasks, bottleneck problems arise when some tasks in the pipeline do not have proper numbers of processors assigned. If the number of processors assigned to one task with heavy work load is not enough to catch up the input data rate, this task becomes a bottleneck in the pipeline system. Hence, it is important to maintain approximately the same computation time among tasks in the pipeline system to maximize the throughput and also achieve higher processor utilization. One bottleneck task can be seen when its computation time is relatively much larger than the rest of the tasks. The entire system's performance degrades because the rest of the tasks have to wait for bottleneck task's completion to send/receive data to/from it no matter how many more processors assigned to them and how fast they can complete their jobs. Therefore, poor task scheduling and processor assignment will cause significant portion of idle time in the resulted communication costs. In Table 4 we added a total of 16 more processors to pulse compression and CFAR processing tasks to the case in Table 3. Comparing to case 2 in Table 2, we can see that the throughput increased. However, the throughput did not improve compared to the results in Table 3, even though this assignment has 16 more processors. In this case, the weight tasks are bottleneck tasks because their computation costs are relatively higher than other tasks. We can see that the receiving time of the rest of tasks are much larger than their computation time. A significant portion of idle time waiting for the completion of weight tasks is in the receiving time. On the other hand, we observe 23% improvement in the latency. This is because the computation time is reduced in the last two tasks with more processors assigned. T_5 and T_6 in equation (2) decrease and therefore the latency is reduced.

5 Conclusions

In this paper we presented performance results for a PRI-staggered post-Doppler STAP algorithm implementation on the Intel Paragon machine at Air Force Research Laboratory, Rome, New York. The results indicate that our approach of parallel pipelined implementation scales well both in terms of communication and computation. For the integrated pipeline system, the throughput and latency also demonstrate the linear scalability of our design. Our design and implementation not only shows tradeoffs in parallelization, processor assignment, and various overheads in inter and intra-task communication etc., but it also shows that accurate performance measurement of these systems is very important. Consideration of issues such as cache performance when data is packed and unpacked, and impact of the parallelization and processor assignment for one task on another task are crucial. This is normally not easily captured in theoretical models. In the future we plan to incorporate further optimizations including multi-threading, multiple pipelines and multiple processors on each compute node.

6 Acknowledgments

This work was supported by Air Force Materials Command under contract F30602-97-C-0026. We acknowledge the use of the Intel Paragon at Caltech for initial development.

References

[1] M. Berger and S. Bokhari. "A Partitioning Strategy for Nonuniform Problems on Multiprocessors,". *IEEE Trans. on Computers*, 36(5):570–580, May 1987.

[2] F. Berman and L. Snyder. "On Mapping Parallel Algorithms into Parallel Architectures,". *Journal of Parallel and Distributed Computing*, 4:439–458, 1987.

[3] R. Brown and R. Linderman. "Algorithm Development for an Airborne Real-Time STAP Demonstration,". *IEEE National Radar Conference*, 1997.

[4] A. Choudhary. *Parallel Architectures and Parallel Algorithms for Integrated Vision Systems*. Kluwer Academic Publisher, Boston, MA, 1990.

[5] A. Choudhary, B. Narahari, D. Nicol, and R. Simha. "Optimal Processor Assignment for Pipeline Computations,". *IEEE Trans. on Parallel and Distributed Systems*, Apr. 1994.

[6] A. Choudhary and R. Ponnusamy. "Parallel Implementation and Evaluation of a Motion Estimation System Algorithm using Several Data Decomposition Strategies,". *Journal of Parallel and Distributed Computing*, Jan. 1992.

[7] M. Linderman and R. Linderman. "Real-Time STAP Demonstration on an Embedded High Performance Computer,". *IEEE National Radar Conference*, 1997.

[8] M. Little and W. Berry. "Real-Time MultiChannel Airborne Radar Measurements,". *IEEE National Radar Conference*, 1997.

[9] M. Snir and et. al. *MPI The Complete Reference*. The MIT Press, 1995.

[10] R. Thakur, A. Choudhary, and J. Ramanujam. "Efficient Algorithms for Array Redistribution,". *IEEE Trans. on Parallel and Distributed Systems*, 1995.