Data and Processor Mapping Strategies for Dynamically Resizable Parallel Applications

Malarvizhi Chinnusamy

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

Examining Committee

Dr. Calvin J. Ribbens

Dr. Eunice Santos

Dr. Srinidhi Varadarajan

June 18th, 2004

Blacksburg, Virginia

Keywords: Grid Computing, Heterogeneous resources, Remapping, Processor allocation, MPI, ScaLAPACK, dynamic resizable applications

Data and Processor Mapping Strategies for Dynamically Resizable Parallel Applications

Malarvizhi Chinnusamy

Abstract

Due to the unpredictability in job arrival times in clusters and widely varying resource requirements, dynamic scheduling of parallel computing resources is necessary to increase system throughput. Dynamically resizable applications provide the flexibility needed for dynamic scheduling. These applications can expand to take advantage of additional free processors, or to meet a Quality of Service (QoS) deadline, or can shrink to accommodate a high priority application, without getting suspended.

This thesis is part of a larger effort to define a framework for dynamically resizable parallel applications. This framework includes a scheduler that supports resizing applications, an API to enable applications to interact with the scheduler, and libraries that make resizing viable. This thesis focuses on libraries for efficient resizing of parallel applications – efficient in terms of minimizing the cost of data redistribution, choosing and allocating the right set of additional processors, and focusing on the performance of the application after resizing. We explore the tradeoffs between these goals on both homogeneous and heterogeneous clusters. We focus on structured applications that have 2D data arrays distributed across a 2D processor grid.

Our library includes algorithms for processor selection and processor mapping. For homogeneous clusters, processor selection involves selecting the number of processors that needs to be added and processor mapping decides the placement of the new processors in the context of the given topology such that it minimizes the amount of data that is to be redistributed. For heterogeneous clusters, since the processing powers of the processors vary, there is also an additional problem of choosing the right set of processors that needs to be added. We also present results that demonstrate the effectiveness of our approach.

Acknowledgements

I would like to thank my advisor, Dr. Calvin Ribbens, for his cooperation, support, patience and guidance not only through my research but also throughout my stay here at Virginia Tech. I would also like to thank Dr. Eunice Santos for her valuable comments on the algorithms and for helping me take the right approach in tackling research problems. I would also like to thank Dr. Srinidhi Varadarajan for being part of my thesis committee. Finally, I would like to thank all my friends especially Gautam, Sudhagar and Anbu for their care and support while pursuing this work.

Table of Contents

ABSTRACT II
ACKNOWLEDGEMENTS III
TABLE OF CONTENTS IV
LIST OF FIGURES V
LIST OF TABLES VI
1 INTRODUCTION 1
1.1 RESEARCH PROBLEM AND APPROACH 1
1.2 MOTIVATION 2
1.3 ASSUMPTIONS AND RESEARCH GOALS 4
2 SURVEY OF RELATED LITERATURE 6
2.1 MODELS FOR DYNAMIC RESOURCE MANAGEMENT 6
2.2 SIMILAR APPROACHES 7
3 GENERAL OVERVIEW AND BACKGROUND INFORMATION 11
3.1 GENERAL ARCHITECTURE 11
3.2 LIBRARIES USED 16
3.3 EXTENDING BLACS TO SUPPORT MPI-2 18
3.4 REDISTRIBUTION ALGORITHMS 21
3.5 HETEROGENEOUS MAPPING ALGORITHMS 24
3.6 EXPERIMENTAL SETUP 26
3.7 EXPERIMENTAL APPLICATIONS 26
4 BASIC CONCEPTS 30
4.1 RESEARCH PROBLEM 30
4.2 GLOBAL REMAPPING 30
4.3 LOCAL REMAPPING 30
4.4 COMPARISON OF THE TWO REMAPPING STRATEGIES 31
4.5 PLACEMENT OF THE NEW PROCESSORS IN LOCAL REMAPPING 32
4.6 EXPERIMENTS 35
5 HOMOGENEOUS ENVIRONMENT 40
5.1 INITIAL MAPPING OF PROCESSORS 40
5.2 EXPANDING THE APPLICATION 44
5.3 PERFORMANCE METRICS 45
5.4 EXPERIMENTAL RESULTS 45
6 HETEROGENEOUS ENVIRONMENT 52
6.1 NEED FOR HETEROGENEOUS ALLOCATION OF DATA 52
6.2 INITIAL MAPPING OF PROCESSORS 52
6.3 RESIZING IN HETEROGENEOUS ENVIRONMENT 54
6.4 EXPERIMENTS 58
6.5 ANALYSIS OF LOCAL REMAPPING ALGORITHM 68
7 CONCLUSIONS AND FUTURE WORK 70
7.1 SUMMARY 70
7.2 FUTURE WORK 71
8 REFERENCES 72

List of Figures

Figure 3.1: Architecture diagram of our framework 12
Figure 3.2: U-curve exhibited by LU factorization 15
Figure 3.3: ScaLAPACK software hierarchy 17
Figure 3.4: Communication between the processes in the first four steps 22
Figure 3.5: Rules in Game of Life application 28
Figure 4.1: Arrangement of the processors in Global remapping scheme 30
Figure 4.2: Arrangement of the processors in Local remapping scheme 31
Figure 4.3: Placement of new processors in Local1 and Local2 algorithms 32

List of Tables

Table 4.1: Amount of data transmitted in Local1 Vs Local2 35
Table 4.2: Redistribution times for Local1 and Local2 remapping algorithms 36
Table 4.3: Redistribution times for Global and Local2 remapping algorithms 36
Table 4.4: Split-up of the redistribution time in Global remapping 37
Table 4.5: Split-up of the redistribution time in Local2 remapping 38
Table 4.6: Split-up of the redistribution time in local1 remapping 38
Table 5.1: Performance of PDGETRF on various processor grid topologies 42
Table 5.2: Performance of PDGEMM on various processor grid topologies 43
Table 5.3: Redistribution times with global and local remapping strategies 46
Table 5.4: Execution time for Game of Life Application 46
Table 5.5: Payoff point for Game of Life Application 47
Table 5.6: Resize cost for Game of Life Application 47
Table 5.7: Execution time for PDGEMV Application 48
Table 5.8: Payoff point for PDGEMV Application 48
Table 5.9: Resize cost for PDGEMV Application 49
Table 5.10: Execution time for PDGEMM Application 49
Table 5.11: Payoff point for PDGEMM Application 49
Table 5.12: Resize cost for PDGEMM Application 50
Table 6.1: Arrangement of processing powers in a processor grid 53
Table 6.2: Different data distributions for the same processor grid 59
Table 6.3: Need for heterogeneous allocation of data to processors 60
Table 6.4: Initial data distribution 61
Table 6.5: T_{opt} *for Initial data distribution* 61
Table 6.6: Performance of the initial processor grid 61
Table 6.7: Performance for Global, Local and Mixed algorithms 62
Table 6.8: Data distribution for global remapping 63
Table 6.9: T_{opt} *for global remapping* 63
Table 6.10 Data distribution for global remapping with replacement 64
Table 6.11: Data distribution for local remapping 64
Table 6.12: T_{opt} *for local remapping* 64
Table 6.13: Data distribution for mixed remapping 65
Table 6.14: T_{opt} *for mixed remapping* 65
Table 6.15: Initial data distribution 65
Table 6.16: T_{opt} *for initial data distribution* 66
Table 6.17: Performance for Global, Local and Mixed algorithms 66
Table 6.18: Initial data distribution 67
Table 6.19: T_{opt} *for initial data distribution* 67
Table 6.20: Performance of the application after series of iterations 68

1 Introduction

1.1 Research problem and approach

Many scientific applications have huge computational requirements and take an exorbitant amount of time when run on a single computer. Though a supercomputer seems to be the ideal solution, not everyone can afford it. Cluster computing offers a low cost alternative to these massive number crunching applications. Personal computers, built from commodity off-the-shelf components and interconnected by high-speed networks like Gigabit ethernet or Myrinet, can deliver the same performance as a supercomputer for many scientific applications. Grid computing, which is considered to be the next wave of the internet, also extends the concept of cluster computing to enable sharing of geographically distributed resources.

As cluster computing becomes pervasive, there is a need to manage the resources efficiently. Until now, static sizing of applications has been a standard approach, i.e., once started the applications run on a constant number of processors till they finish execution. But this approach has several drawbacks with respect to efficient usage of resources (Section 1.2 provides a list of some scenarios). Dynamically resizable applications offer a promising alternative. These are applications that can dynamically vary, at specific points, the number of processors on which they are executing and allow the data to be remapped at these points, without affecting the correctness of execution. While we recognize that not all applications can take advantage of this approach, there are many applications that will benefit greatly. Resizing affects application specific parameters like the number of processors used and the topology of the processor grid, which significantly influence the performance of the application. Hence for efficient resizing, the application should work in close cooperation with the scheduler. The application might even need to negotiate with the scheduler in deciding the amount by which it has to shrink or expand.

This thesis is a part of a larger effort to define a framework for dynamically resizable parallel applications. This framework includes a scheduler that supports resizing applications [1], an API to enable the application interaction with the scheduler, and libraries that make resizing viable. This thesis focuses on libraries for efficient resizing of

parallel applications – efficient in terms of minimizing the cost of data redistribution, choosing and allocating the right set of processors that needs to be added, and focusing on the performance of the application after resizing. We explore the tradeoffs between these sometimes conflicting goals on both homogeneous and heterogeneous clusters. We propose algorithms for processor selection and processor mapping. For homogeneous clusters, this involves just selecting the number of processors that needs to be added and deciding where to place the new processors in the context of the given topology such that it minimizes the amount of data that is to be redistributed. For heterogeneous clusters, since the processing powers of the processors are different, there is also an additional problem of choosing the right set of processors that needs to be added. We also present some results that demonstrate the effectiveness of our approach.

1.2 Motivation

For efficient resource and job scheduling, parallel clusters usually employ some kind of queuing system policy (e.g., priority or best-fit) that maximizes the throughput of the system. These queuing systems usually penalize applications that need large number of processors. Priority queuing systems used on parallel machines often favor jobs requiring few processors, by giving higher priority to jobs that require few processors than those that need many processors. In a best-fit queuing system, processor availability constrains the scheduling of large applications. Hence to avoid a long delay in scheduling large applications, users often request only as many processors as they think may be available. Once started, the application cannot make use of any additional processors that become available, even when other applications have finished and most of the system is idle.

Or, consider another scenario. Suppose the cluster is very lightly loaded. The scheduler is not concerned about the utilization of the processors. It lets the applications underutilize the processors and allocates as many processors as the user asks. But as the load increases, the scheduler tries to achieve maximum processor utilization. So an application started in a lightly loaded environment, which is not utilizing the processors efficiently may get suspended or terminated, when the load on the system increases. Otherwise, the new applications have to wait, when most of the processors are

underutilized, decreasing the throughput of the system.

Thus with static processor allocation, the system cannot be fully utilized and the fairness among the jobs cannot be maintained. Dynamically resizable applications seek to alleviate this problem. For example, applications can start early with a minimum number of processors available and later can increase their number of processors, when more processors become idle. The system throughput is maximized, while the application runs faster, thus meeting the conflicting demands of both the user and scheduler.

The following list summarizes several potential benefits of resizable parallel applications

1. Usually applications are classified into different priority classes, based on several criteria; for example jobs of a high paying customer are allocated the highest priority class. When a high priority application arrives, the low priority applications may be suspended or checkpointed to be resumed later. But with resizability, these low priority applications can shrink, releasing processors for the high priority applications. Thus the low priority jobs can continue to execute at a slower rate till more processors become available. An example of this would be in a cycle stealing environment, where the jobs are run at a low priority compared to that of the user. When a user arrives and there are not enough processors to migrate to, instead of the whole job getting suspended, the job can shrink and still continue to execute.
2. Resizability helps in providing quality of service (QoS) in terms of the time taken by an application to finish. If the scheduler has information on how well the application performs (iteration rate) with different number of processors, then depending on the number of iterations remaining, the scheduler can resize the application so that the application finishes before its deadline.
3. Resizability helps in achieving increased system throughput. Idle processors are often present in the system, because they are not enough to schedule a new application. This situation arises even if the scheduler uses backfilling strategies. If the applications are resizable, these processors can be used for expanding the applications that are currently executing or for starting a new application with these processors, leading to an increased system throughput.
4. Resizability also helps in dynamic resource management based on the nature of the application, i.e., some applications have several stages in their execution, with each

stage having a different degree of parallelism. Dynamic resource management helps in dynamically allocating the appropriate amount of resources required for each stage.

This research seeks to apply dynamic resource management techniques by allocating processors dynamically for parallel applications.

1.3 Assumptions and Research Goals

In this thesis, we deal with structured applications where the data is distributed across processors arranged in a two-dimensional (2D) grid. There are different partitioning strategies [2] available to distribute data across a 2D grid. But most scientific applications partition their data such that each processor has only four direct neighbors. This constraint helps them to simplify the communication overhead and also to use existing linear algebra kernels like ScaLAPACK [3] for their computation. These applications typically run for many iterations and consume hundreds or thousands of CPU hours. As mentioned earlier in Section 1.1, we consider only those applications that lend themselves to resizing i.e., they must be able to dynamically vary the number of processors in which they are executing at specific points and allow the data to be remapped at these points, without affecting the correctness of execution.

Application resizing must be performed efficiently to obtain better system and job performance. For example, the topology to which the application expands may greatly influence the performance of the application. We measure the benefit of choosing the appropriate topology in the homogeneous case and provide support for the user to specify some constraints for choosing the topology.

Resizability comes with its own price. The benefits of resizing may not be realized if this overhead is not minimized. Moreira

et al. [4] showed that this overhead could be close to 10% of the execution time on large sets of processors. The major part of this overhead occurs in redistributing the data to the new set of processors, a cost which scales with the problem size per processor. Therefore an efficient redistribution strategy will cause a significant reduction in the reconfiguration overhead. We consider strategies for placing the new processors into an existing 2D topology and measure the benefit of a good placement of processors for the

homogeneous case. For the heterogeneous case, we propose and evaluate some algorithms for selecting the processors such that a good computation rate results on the new set of processors while reducing the cost of data redistribution.

Though the algorithms we have proposed deal with 2D arrays on 2D grids, they can be extended easily to multidimensional arrays and grids. The algorithms are meant for block-partitioned distributions. Even though block-cyclic distributions can take advantage of some of these algorithms, the algorithms make the biggest difference for block-partitioned distributions. We use ScaLAPACK as the underlying framework for our applications; but the concept is not specific to ScaLAPACK and can be used to enhance the performance of any library that supports resizing. Since ScaLAPACK deals with block-cyclic matrices only, in our experiments we adjust the block size in ScaLAPACK to simulate block-partitioned behavior. To implement these algorithms, we extended BLACS, the underlying library used by ScaLAPACK for communication, to support the dynamic process management functionality of MPI-2. We have also extended the redistribution library of ScaLAPACK to support heterogeneous redistributions.

1.4 Reader's Guide:

Chapter 2 presents a survey of related work done in the field of dynamic resizing of applications. Chapter 3 gives some background information on ScaLAPACK, heterogeneous mapping and redistribution algorithms. Chapter 4 discusses basic concepts of global and local remapping. The applicability of these remapping techniques on a homogeneous set of processors is discussed in Chapter 5, and Chapter 6 presents the same for heterogeneous processors. Chapter 7 concludes with future work and research directions.

2 Survey of related literature

2.1 Models for dynamic resource management

There has been significant work done in the field of dynamic resource management systems. Moreira and Naik classify them into different models.

- **Workers model:** A central entity, which may be an active master task or a central pool, defines the tasks that must be executed on the workers and also collects the data from the workers. Piranha [5], which is based on this model, uses tuplespaces (a central pool) for coordinating between the workers. Though this model may suit a lot of applications, scientific applications that involve large distributed data structures (e.g., for matrix operations) give poor performance with the workers model because of the amount of data moved. These scientific applications only require course grained resource management, which allows them to make use of facilities like caching to improve their performance.
- **Fork-join model:** This is basically meant for shared memory multiprocessors. A few (variable number) kernel threads are scheduled for execution on the underlying processors. They act as virtual processors for the user level threads. [6,7,8,9] are based on this model. These threads may be migrated from one virtual processor to another based on the load on the systems.
- **HPF model:** (High Performance Fortran): This model [10] does not support resizing directly and hence does not provide a flexible environment for resizing. For example, there are no constructs to specify directly the number of desired tasks. However resizing can be achieved by making use of the functionality that allows the specification of virtual processor grids and distribution of data onto the processor grids.
- **AMP model** (Adaptive Multiblock PARTI): [11] The application is spawned on the maximum number of processors that the application can execute. Only tasks on certain processors, called active tasks, are involved in intensive computation. The other tasks, called skeleton tasks, take part in the execution but do not perform rigorous computations. The disadvantages of this model are that it has a

fixed number of executing tasks and that the co-existence of skeleton tasks and the active tasks of different applications may affect the performance of the active task as the operating system has to work in a timesharing mode. This also affects the scalability of the system.

- **SOP model:** This model is basically for SPMD applications. The parallel program identifies certain points as Schedulable and Observable Points (SOPs), which are the only points when a resize can happen. At these points, the application contacts the scheduler and based on the reply from the scheduler it may expand, shrink or proceed without any change. During resizing, there can be a change in the number of processes or change in the data associated with the processes or both. This resizing can be due to external change (based on scheduler's decision) or due to an internal change (based on the nature of the application). Our library falls under this category. Dynamic Resource Management System (DRMS) [4], a similar library that supports resizing also operates on this model.

Now let us look in detail at some frameworks that are similar to our approach. We will compare and contrast these frameworks with our approach in the subsequent sections.

2.2 Similar Approaches

2.2.1 Dynamic Resource Management System (DRMS)

DRMS [4] is based on the SOP model. Though our approach is very similar to their approach, we differ in certain aspects from them. For example, during resizing in DRMS, the whole application gets suspended by the runtime system until new tasks are created and is then continued execution on this new set of tasks. None of the old tasks are involved in the execution after resizing. In contrast to this, in our approach, we expand the application by just adding the extra tasks to the already existing set. The old tasks are also involved in the execution after resizing. This helps us to devise algorithms that reduce the overhead of resizing, by retaining most of the data with the old tasks.

In DRMS, the user has to manually specify the valid sizes for the application expansion. The scheduler remaps to one of these sizes depending on the processors available. In most situations, there are a lot of sizes to which the application can expand;

it would be impossible for the user to list them all. Further, the user might not be aware of the performance of the application on a particular size. Since the application might use some underlying libraries, it may be impossible to model theoretically the application to obtain its performance. A better strategy would be to learn from the past executions of the applications or to allocate certain iterations in the current execution for collecting the application's performance data. Though DRMS has a performance data gatherer component (PDG), which assists the users and system administrators to understand the characteristics of the user programs, the data collected is not used automatically by the application or the scheduler in deciding the partition sizes.

Given the total number of processors to which the application has to expand, DRMS has some intelligence [12] incorporated in the system in choosing the processors that should be allocated to each dimension. Given the total number of processors as p, and if a d dimensional processor grid needs to be constructed, they choose the processor vector ($p_1, p_2,.., p_d$), such that

$$\frac{n_1}{p_1} \simeq \frac{n_2}{p_2} \simeq \dots\simeq \frac{n_d}{p_d}$$

where $p_1, p_2,.., p_d$ denote the number of processors in the 1, 2,…, d^{th} dimension respectively satisfying the condition $p_1 \times p_2 \ldots \times p_d = p$ and $n_1, n_2, \ldots n_d$ represent the data in the 1, 2 … d^{th} dimension, respectively. This reduces the communication overhead for applications like Jacobi where the data exchanged is proportional to their area of contact. But not all scientific applications follow this model. Some applications might have more communication in one grid dimension compared to others. Hence the processor grid selection must be based on the nature of the application. In our approach, the application with the help of the scheduler can intelligently choose the topology that suits it the best based on its history. Further, it also allows the user to make a topological preference, if he chooses to do so.

DRMS employs a very simple redistribution scheme, which does not optimize on the parallelism in the communication schedule of the processors. Optimizing on the

communication schedule can significantly reduce the overhead of redistribution. We will see the performance difference optimization offers in the subsequent chapters.

2.2.2 CHARM and Adaptive MPI

Adaptive MPI [13] works on the top of the Charm++ framework [14], which supports processor virtualization. When the application is started in parallel on multiple processors, several Virtual MPI processes (VPs) may get mapped to a single physical processor. When more processors become available during program execution or when the load on one of the processors decreases, the VPs from the heavily loaded processors migrate to the lightly loaded processors. Thus the applications can dynamically expand or shrink the number of processors on which they execute (resize themselves) or can automatic load-balance to achieve better performance. Applications that can make use of this fine-grained parallelism work very efficiently in this framework. However, this approach increases the overhead in applications that use linear algebra kernels for computation. These applications typically have large amounts of data to operate on and make use of locality of data to yield better performance. Hence they prefer a more coarse-grained parallel environment.

2.2.3 Distributed object migration environment (DOME)

The Dome project [15] does not focus on dynamic resizing but provides a dynamic load balancing framework for parallel programs to execute over a heterogeneous network. It provides an API that allows users to write parallel programs that automatically distribute their data over a heterogeneous network and dynamically load balance as the program runs. It is available as a library of C++ classes. Since it uses PVM for process control and communication, DOME can support dynamic resizing easily.

The frameworks discussed above dynamically resize the applications either by spawning a new set of tasks, or by migrating the processes between the processors based on their load, or by activating the idle processes in the processor when the processor becomes available. In any case, they have to redistribute the data during this resizing step to balance the load on the processors. However these frameworks provide very generic

resizing libraries. Structured applications incur a lot of overhead if a generic resizing library is used. The decision made on the number of processors chosen and the placement of the new processors in the processor grid can significantly influence the performance of the application. In this thesis, we investigate the challenges involved in efficient resizing of structured applications and propose algorithms to tackle these challenges.

3 General Overview and Background Information

In this chapter, we shall discuss in detail the framework we developed to support dynamic resizing of applications.

3.1 General Architecture

The framework we have built for supporting resizable applications includes the following components

1. A scheduler that supports resizing applications [1]. It is responsible for determining which applications should expand or shrink and the amount by which they should do so, such that the overall throughput of the system is increased.
2. Resizing affects a few application specific parameters that influence the performance of the application greatly. For example, there might be a pattern on the number of processors for which the application gives great performance. The scheduler is not aware of these values as they depend on the nature of the application. The performance of the application can be increased significantly, if the application can cooperate with the scheduler in its decision making process. Hence, an API that enables application interaction with the scheduler also forms a part of this framework.
3. When the application is resized, the application has to choose the number of processors that are to be added, map the processors to the original processor grid and redistribute data from the original processors to the new set of processors. To make resizing easier, a library that performs all these operations efficiently is included in the framework. The existing applications can support resizing with minimal changes to their code. This library was implemented on the top of ScaLAPACK. BLACS [17], the communication layer of ScaLAPACK [3] was modified to support dynamic process management. However, the algorithms are not specific to ScaLAPACK and can be easily implemented on the top of other frameworks.

Figure 3.1 clearly shows the interaction between these components. The DQ scheduler [16], developed at Virginia Tech was extended to support resizable applications. A detailed explanation of the scheduler is provided in [1]. Our research deals with providing efficient algorithms for the resize library.

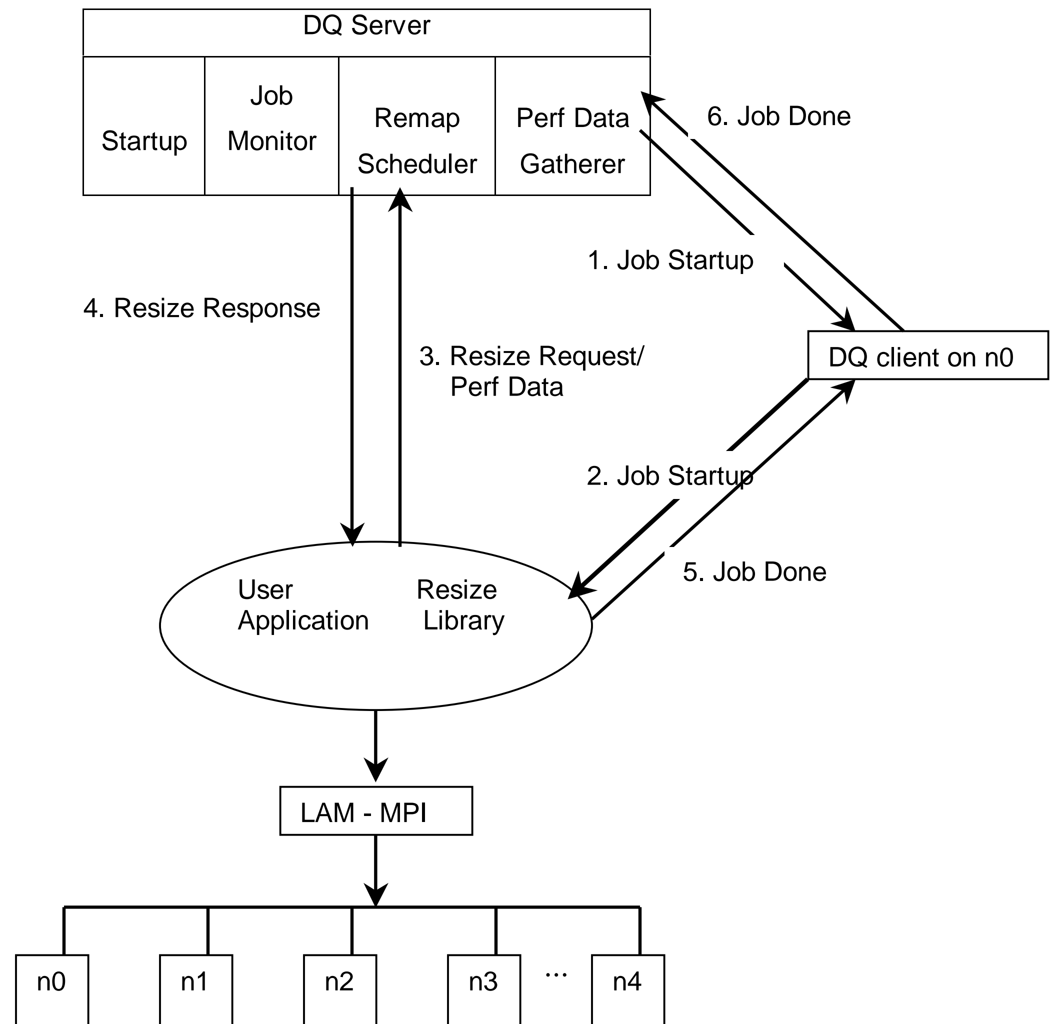


Figure 3.1: Architecture diagram of our framework

We consider only SPMD applications, where a copy of the program runs on each node of a parallel machine. The job startup on the DQ server invokes the job startup of the DQ client on node 0. The DQ client launches the application on the various processors as shown in Figure 3.1.

The application identifies certain points as remap points at which it can remap its data and still maintain correctness of execution. At these points, the application contacts the scheduler to check if it can expand or if it has to shrink. The application is never interrupted between these remap points for resizing. These remap points have to be chosen carefully. If the remap points occur often, it might affect the performance of the application as all the processes have to synchronize at the remap points. But on the other

hand, if the remap points are very wide apart, the application may get suspended if the scheduler decides to shrink an application and the application does not respond within a timeout.

Once the job is started, at every remap point, the application sends a resize request to the DQ server. Based on the availability of processors, performance of the application on different processors, number of jobs in the queue and several other criteria, the scheduler responds with one of the following messages: shrink, expand or no change. The resize library, to which the user application links, provides the application the capability to expand or shrink based on the instruction from the scheduler. In case of shrinking or expanding, the application may enter into a negotiation phase with the scheduler in deciding the amount by which it has to shrink or expand, so that the application can perform better on the new set of processors. Once the processors are allocated, the application uses the resize library to remap its global data distributed across processors to its new processor set.

The following pseudocode shows how a typical application takes advantage of resizability

1. *Do in a Loop*
    a. *Compute*
    b. *Send resize request to the scheduler with performance data for the previous compute step*
        i. *if (response is no change)*
            1. *continue*
        ii. *if(response is expand)*
            1. *Get the maximum number of processors that the application can expand to*
            2. *Calculate the number of processors within the available limit in which the application performs its best. Negotiate with the scheduler if any performance data is necessary for this calculation*

3. *Indicate to the scheduler the number of processors that it has decided to expand to*
4. *Obtain the schema file for expansion from the scheduler and expand.*
5. *Redistribute data to the new processor set*

iii. *if (response is shrink)*

1. *Get the minimum number of processors that the application has to relinquish.*
2. *Calculate the minimum number of processors beyond or equal to this limit, which the application can relinquish.*
3. *Indicate to the scheduler the number of processors that the application has decided to give up.*
4. *Redistribute the data to the new set of processors.*
5. *Signal the scheduler on completion of the redistribution, so that the scheduler can free the resources and allocate them to some other job.*

2. *End loop*

3.1.1 Gathering performance data

At the remap points, the application sends the performance data for the previous step to the scheduler along with the resize request. The scheduler stores this information along with the number of processors and topology of the processor grid. The first few iterations of the program are used for experimentation, during which the scheduler gathers data about the application performance. Performance is measured in terms of iteration rate, which is the inverse of the time taken to execute an iteration. Performance can be also obtained from past executions of the program as well as from the iterations which the application executes after experimentation. The scheduler uses this information to determine the number of processors by which the application has to expand or shrink. Finding the sweetspot of the application is a simple example that demonstrates how the scheduler makes use of this performance data in its decision making.

3.1.2 Determining the sweetspot of the application

As the number of processors used by the application increases, the execution time for the application exhibits a U curve (as shown in Figure 3.2), i.e., the execution time decreases upto a certain point after which it remains constant for sometime and then starts to increase. This is because after that point, the communication overhead starts to dominate over the computation gain obtained and hence the application is not benefited by an increase in the processors. This is the “sweetspot” of the application. Once the scheduler finds from the performance data that the application has reached its sweetspot, then even when processors are available, it would not expand the application.

3.1.3 Experiment: Finding the sweetspot of the application

Figure 3.2 shows how the execution times of LU factorization application with N x N data matrix, where N=4800, run on different number of processors exhibit a U curve.

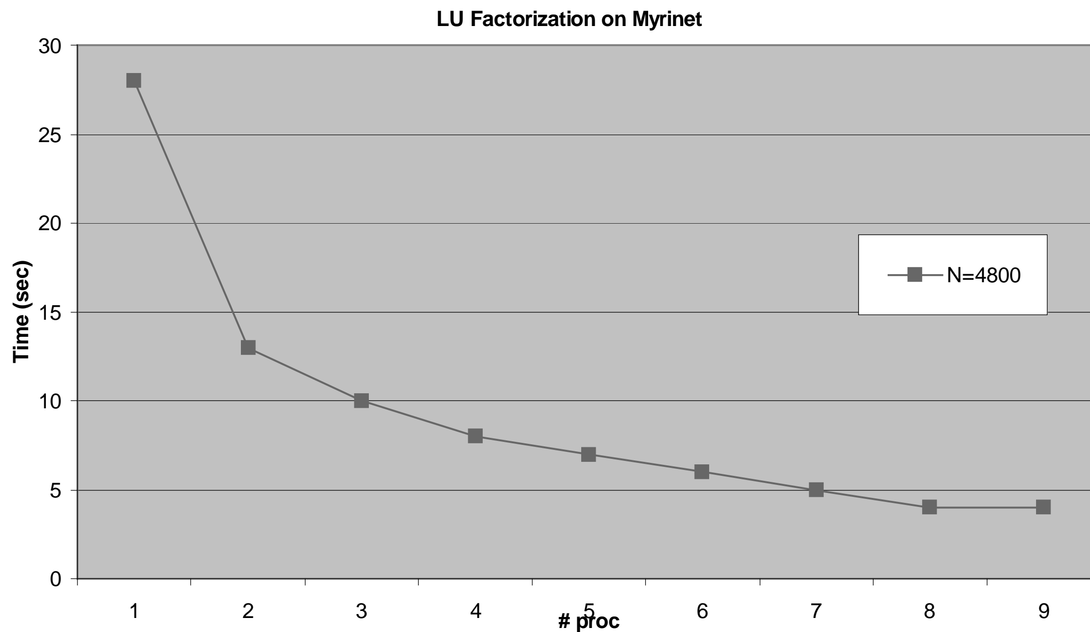


Figure 3.2: U-curve exhibited by LU factorization

In this thesis, we assume that only one process runs on a processor. So, we use the term process and processor interchangeably.

3.2 Libraries Used

3.2.1 LAM-MPI

LAM-MPI [18] is an open source implementation of the Message Passing Interface (MPI) specification, which includes MPI-1 and much of MPI-2. Since it supports dynamic process management, we used this as our underlying message passing library for ScaLAPACK.

3.2.2 ScaLAPACK

ScaLAPACK [3], an acronym for Scalable Linear Algebra PACKage, or Scalable LAPACK, is a library of high-performance linear algebra routines which can solve systems of linear equations, linear least square problems, eigen-value problems, singular value problems, do matrix factorizations or estimate condition numbers for distributed memory message passing MIMD applications. Both dense and band matrices are supported. It uses block-partitioned algorithms in order to minimize frequency of data movement between different levels of memory hierarchy. ScaLAPACK uses distributed memory versions of the Level 1, Level 2, and Level 3 BLAS, called the Parallel BLAS or PBLAS [19], and a set of Basic Linear Algebra Communication Subprograms (BLACS) [17] for communication tasks that arise frequently in parallel linear algebra computations. Figure 3.2.2 shows the software hierarchy of ScaLAPACK.

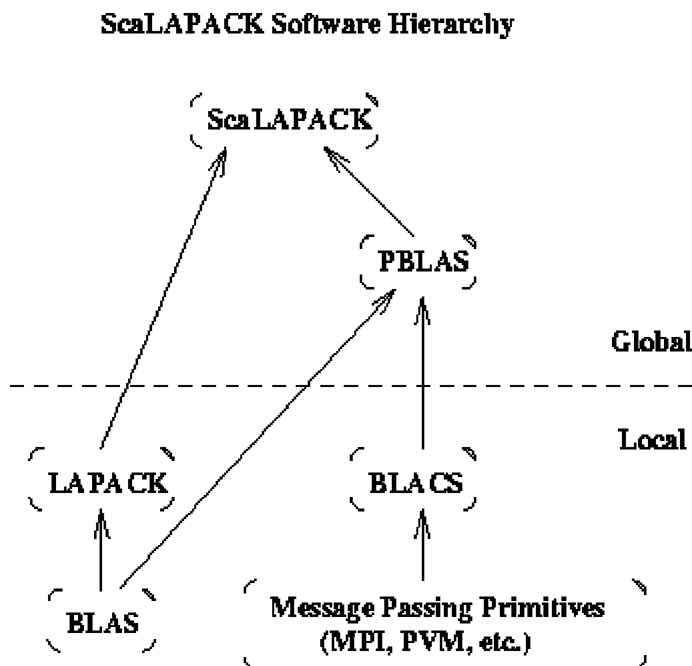


Figure 3.3: ScaLAPACK software hierarchy

In Figure 3.3, the components below the line, labeled Local , are called on a single processor, with arguments stored on single processors only. The components above the line, labeled Global, are synchronous parallel routines, whose arguments include matrices and vectors distributed across multiple processors.

3.2.3 BLACS

BLACS[17], an acronymn for (Basic Linear Algebra Communication Subprograms), is a portable, linear algebra specific layer for communication. The computational model consists of a one- or two-dimensional process grid, where each process stores pieces of the matrices and vectors. The BLACS include synchronous send/receive routines to communicate a matrix or submatrix from one process to another, to broadcast submatrices to many processes, or to compute global reductions (sums, maxima and minima). The underlying communication is through PVM [20] or MPI (MPICH [21] and LAM-MPI [18]). The MPI BLACS supports only the MPI-1 standard, i.e., it has no support for dynamic process creation and management; but the PVM

version of BLACS supports dynamic process creation. Hence it was necessary to extend MPI BLACS to support dynamic process creation and management.

3.3 Extending BLACS to support MPI-2

3.3.1 Blacs_setup

In PVM BLACS, a call to blacs_setup (iam, nprocs) is used to allocate the virtual machine and spawn off processes. This function reads in parameters like the executable, number of processors, and some debugging flags from a file called blacs_setup.dat and spawns processes. Since PVM supplies a dynamic system, allowing processes to be added to the system on the fly, this routine accomplishes meaningful work only in PVM BLACS. For MPI BLACS it is equivalent to BLACS_PINFO, which returns my process id and the total number of processes available. We extend this BLACS_SETUP function in MPI BLACS to provide dynamic process creation.

3.3.2 Application support

We shall discuss the sequence of steps that a typical application has to follow to use ScaLAPACK functions. The steps are listed in detail below along with the corresponding function call in C (in italics). A detailed explanation of the functions has not been provided. Readers are asked to refer to [17] for a detailed explanation of these function calls.

A ScaLAPACK application based on MPI, first initializes MPI execution environment by calling

MPI_Init(&argc, &argv);

It then calls blacs_pinfo to get its processor id and the total number of processors used.

blacs_pinfo(&mypnum, &nproc);

The existing context is obtained by a call to blacs_get. The application then calls either blacs_gridinit or blacs_gridmap to define the size and dimensions of the process grid. This identifies the processes that are involved in the communication. BLACS allows the application to reinitialize at various points in the program to redefine the process grid.

i = 0;

```
blacs_get(&i, &i, &ictxt);
blacs_gridmap(&ictxt, proc, &ldurow, &nprow, &npcol);
blacs_gridinfo(&ictxt, &nprow, &npcol, &myprow, &mypcol);
```

Here proc is the two-dimensional array containing process numbers which are mapped onto the new process grid, ldurow is the leading dimension of the proc array, nprow and npcol are the number of rows and columns of the processor grid. The row and column of the current process in the process grid is given by myprow and mypcol.

The application finally invokes the blacs_exit(&continue) routine to release all the BLACS contexts and the memory allocated by the BLACS. Only when continue = 0, MPI_Finalize is called to exit from MPI. There can only be one call to MPI_Finalize in a program. So at intermediate steps, blacs_exit is called with continue = 1 and at the end of the program, blacs_exit is called with continue = 0 or MPI_Finalize is called directly.

3.3.3 For expanding

To add a new set of processes dynamically, the application has to exit the existing context and then call blacs_setup(). Blacs_setup reads from a file blacs_setup.dat, the schema file for spawning. In the new context, the old parent processes form the first part of the communicator followed by the process ids of the children.

The corresponding C code is

```
i = 1;
blacs_exit(&i);
blacs_setup(&mypnum, &nproc);
```

The application then proceeds as usual with a call to blacs_get to get the new context and then to blacs_gridmap to define the size and dimensions of the new process grid.

3.3.4 Extended blacs_setup of MPI_BLACS

We extended this function to support dynamic process creation for MPI. A flag newprocess is used to identify if the process existed already or if it has been newly created. We could have used MPI_Comm_get_parent to do the same, but blacs_setup can be called repeatedly, in which case we cannot identify a newly spawned process from a process spawned earlier. For example, consider an application started initially with three

processes. Two processes are spawned in the second step and in the third step three more add in. The parent is not NULL even for the second set of processes and we cannot distinguish the second set from the newly spawned set. The MPI_Comm_spawn function call is used to spawn the new processes. This function is a collective operation over all the existing processes of the application. At the end of this call, a child group is started up just like any ordinary MPI application. They have a separate MPI_COMM_WORLD which is distinct from that of the parent processes. An intercommunicator between the two groups is returned by the MPI_Comm_spawn call. The children can get this intercommunicator by calling MPI_Comm_get_parent. MPI_Comm_merge can be used on the intercommunicator to merge the two groups. It has parameters that decide on the ordering of the processes in the two groups.

Pseudocode for extended blacs_setup function

1. If (!newprocess)
    a. Read in schema file blacs_setup.dat and spawn processes
    b. Merge the intercommunicator to form a new intracommunicator. The parent processes form the former part of the new intracommunicator.

   else

   Merge the intercommunicator to form a new intracommunicator. The children form the latter part of the new intracommunicator.
2. Duplicate the communicator and store it in a new handle BI_MY_MPI_COMM.

BI_MY_MPI_COMM is a blacs internal variable that is created to store a copy of the current communicator. Since all the communicators created earlier are destroyed when blacs_exit is called and a copy of the communicator is needed for the collective call MPI_Comm_spawn which is called in blacs_setup, whenever a new communicator is created, it is duplicated and stored in BI_MY_MPI_COMM.

3.3.5 For Shrinking

A stack of communicators along with the corresponding processor grid sizes is maintained. Every time a new communicator is created, BI_MY_MPI_COMM, which

contains the duplicate of the new communicator is stored on the stack. When we need to revert back to the previous processor grid, we copy the communicator in the stack to BI_MY_MPI_COMM, call blacs_exit with continue=0 for those processes that are no longer part of the application and terminate them, and blacs_exit with continue =1 for other processes. blacs_pinfo copies the communicator in BI_MY_MPI_COMM to the current context. Hence, the application reverts back to the old communicator when the application executes the calls to blacs_pinfo, followed by calls to blacs_get and blacs_gridinit or blacs_gridmap that follow the blacs_exit call as usual.

3.4 Redistribution Algorithms

When applications are resized, data in the initial set of processors has to be redistributed to the new set of processors. We shall see below some redistribution algorithms available in the literature.

3.4.1 Caterpillar algorithm

ScaLAPACK provides a redistribution routine that copies a submatrix of a matrix A onto a submatrix of another matrix B. It is very flexible and allows A and B to have different distributions, to be on different processor grids, to have different block sizes and with the beginning of the area to be copied at different places on A and B.

Redistribution is based on the caterpillar algorithm proposed in [22]. The algorithm assumes a block-cyclic data distribution on a virtual grid of processors. It uses a simple round robin scheme similar to a rolling caterpillar to communicate between the processors. It communicates in p steps. At step d, each process P_i, $0 \leq i < p$ exchanges its data with the process at index position $(p - i - d) \mod p$ in parallel. Figure 3.4 shows how the processors communicate in the first four steps of the redistribution algorithm, when eight processors are used.

Step 1	Step 2	Step 3	Step 4
0 1 2 3 7 6 5 4	0 1 2 7 3 6 5 4	7 0 1 2 6 5 4 3	7 0 1 6 2 5 4 3

Figure 3.4: Communication between the processes in the first four steps

In each step, the processors in the upper row of each step communicate with the processor that is directly below it in parallel. That is in step 1, processors 0 and 7 exchange their data. Simultaneously, processors 1 & 6, 2 & 5, 3 & 4 also exchange their data. Similarly in step 2, data exchange takes place in parallel between processors 0 & 6, 1 & 5, 2 & 4, 3 & 7.

Though this algorithm reduces node contention, it does not fully utilize the network bandwidth. For the above example, consider the scenario where processors 0 & 7 have no data to be exchanged between them, but 0 has to exchange data with 6 and 5. Processors 5 and 2 also have no data to be exchanged between them. In this algorithm, 0 waits till 1 and 6 finish exchanging their data. It could have exchanged with 5, but according to the communication schedule it has to exchange with 6 before it can exchange with 5. Thus the communication schedule is not optimized and so processors spend a lot of time in waiting to send to or receive from another processor. Note that this situation arises in spite of the fact that there are no barriers at the end of each step.

3.4.2 Bipartite matching algorithm

The bipartite matching algorithm of Desprez et al. [23] uses graph theory and modern algebra techniques to derive optimal schedules for the communication needed to redistribute arrays. Let P, Q denote the processors in the initial processor set and final processor set respectively. It considers remapping to be similar to constructing a bipartite graph with edges from a vertex in P to a vertex in Q. Now the problem reduces to obtaining disjoint edge matchings from this graph. An edge matching M in a graph G(V,E) is a subset of the edge set E such that no two edges in M are incident on the same

vertex, i.e., if $\{w,x\},\{y,z\} \in M$, then the vertices w, x, y, z are distinct. The algorithm repeatedly extracts from the edge set a maximum matching that saturates all maximum degree nodes. These matchings represent the communication steps of the algorithm. The algorithm reduces the data transfer cost because of this efficient scheduling. But the scheduling cost is high, $O((|P| + |Q|)^4)$, and sometimes can be larger than the data transfer cost.

3.4.3 Circulant matrix algorithm

The circulant matrix algorithm [24] is very efficient for block cyclic data redistributions. It reduces the overall time for communication by reducing data transfer, communication schedule and index computation costs. It generates a schedule that minimizes the communication steps, eliminates node contention in each step and utilizes the network bandwidth fully. The computation of this schedule is very fast, $O(\max(P,\ Q))$, as against the Bipartite matching algorithm, $O((|P| + |Q|)^4)$, where P, Q denote the processors in the initial processor set and final processor set respectively. Since it optimizes on the block cyclic property, it is not very efficient for block-partitioned data redistributions.

For our experiments, we have used an implementation of the caterpillar algorithm that comes with ScaLAPACK. Implementing the circulant matrix algorithm for the block cyclic case and bipartite matching algorithm for local remapping of block matrices is mentioned as future work in Chapter 7.

3.4.4 Extension for heterogeneous clusters

We modified the implementation of the caterpillar algorithm to support heterogeneous redistributions. ScaLAPACK packs the information about the distribution of each matrix in a data structure called DESC comprising of 8 integers. It stores the context, which specifies the processor grid over which the matrix is distributed, the total number of rows (m) and columns (n) of the data matrix, the block sizes used for distributing the row (nbrow) and column (nbcol), the process row over which the first row of the array is distributed (sprow), the process column over which the first column of

the array is distributed (spcol) and the leading dimension of the local array. The structure is shown below.

```
typedef struct {
  int   desctype;
  int   ctxt;
  int   m;
  int   n;
  int  nbrow;
  int  nbcol;
  int   sprow;
  int   spcol;
  int   lda;
}     DESC;
```

To extend the redistribution library available with ScaLAPACK to support heterogeneous redistributions, we modify DESC to accept an array of blocksizes corresponding to each row(column) instead of a single blocksize common to all the rows(columns). The heterogeneous caterpillar algorithm is similar to that used for homogeneous redistribution with a slight modification to support varying blocksizes.

3.5 Heterogeneous mapping algorithms

Consider a heterogeneous network of workstations (HNOWs) where processors are of varying processing power. The Master-Worker model seems to be an ideal solution. It uses dynamic strategies for allocating data to the processors. Since these scientific applications usually operate on large matrices, dynamic strategies for allocating data to the processors are not suitable as they incur a large amount of data transfer and control overhead. Hence the processors have to be mapped on to the processor grid statically. If a uniform block/block cyclic data distribution is used for data allocation, the application executes at the speed of the slowest processor. Hence to balance the load on the processors, block sizes have to be allocated such that the computation time required for processing the block is proportional to the relative performance of the processor. Since the processors are of varying processing powers and all the processors in the same row (column) of the processor grid are assigned the same number of data rows (columns), the placement of the processors in the processor grid influences the utilization of the

processors. Beaumont et al. [25] have proved that the problem of placing the processors in the processor grid such that all the processors are used efficiently is NP complete.

Hence heterogeneous mapping involves solving two problems

1. Optimal mapping of processes into processor grid
2. Determination of block-sizes according to the relative performance of the processor.

Though several scheduling strategies [26, 27] for mapping task graphs onto HNOWs exist, only singular value decompositon [25] and natural block decomposition [28] algorithms deal with 2D processor grids where each processor communicates with four direct neighbors.

3.5.1 Singular Value Decomposition

Beaumont et al. [25] considered the mapping of the processors as an optimization problem and proved that it is NP hard for 2D grids. In a p x q processor grid, assume that each processor P_{ij}, with cycle time t_{ij}, is allocated r_i rows and c_j columns of data. For each processor, $r_i \times t_{ij} \times c_j \leq 1$ to ensure that P_{ij} can process its block within one cycle. Since the total number of data elements being processed is $(\sum_{i=1}^{p} r_i) \times (\sum_{j=1}^{q} c_j)$, they get the optimization problem as

$$\textbf{Objective } Obj_2 : \max_{r_i \times t_{ij} \times c_j \leq 1} \left\{ \left(\sum_i r_i \right) \times \left(\sum_j c_j \right) \right\}.$$

Since the solution to this problem is NP complete, they have proposed a heuristic solution. The heuristic uses the singular value decomposition (svd) to compute a rank-1 approximation to a p x q matrix containing the processors' cycle times. This yields row and column assignments such that idle time of the processors is minimized. This algorithm is used for our experiments and hence we discuss this in detail in Chapter 6.

3.5.2 Natural Block Decomposition

While the singular value decomposition algorithm restrains itself to 2D grids, the natural block decomposition [28] can be extended to multidimensional grids. This

algorithm considers the m-dimensional data distribution as a combination of m data distributions each of which distributes data along a dimension of the processor grid. If s is the size of data matrix along a dimension, r is a vector that contains the relative performances of the processors in the grid along that dimension, then the amount of data allocated to j^{th}-process with relative performance r_j corresponding to that dimension is given by,

$$a_j = \left\lfloor \frac{r_j}{\sum_{i=0}^{\sigma-1} r_i} \cdot s \right\rfloor,$$

3.6 Experimental setup

For our experiments, we used a homogeneous cluster comprising of 64, 1.4MHz AMD Opteron processors. These processors are connected in a fat tree topology by a hierarchy of Gigabit Ethernet switches. This reduces the collision when data travels in parallel between the machines.

3.7 Experimental Applications

The PBLAS (Parallel Basic Linear Algebra Subprograms) [19] is a collection of software for performing linear algebra operations on distributed-memory concurrent supercomputers, networks of computers, and any system on which MPI or PVM is available. It is the distributed-memory analog to the BLAS, hence is a major component of ScaLAPACK, the distributed-memory analog to LAPACK. We have used some calls of PBLAS to determine the overhead for reconfiguration with respect to computation time. To demonstrate the effectiveness of our algorithms, we ran our algorithms with applications of different computation complexity. The applications use functions from different levels of PBLAS. The following explains in detail the different function calls used.

3.7.1 PDGEMM

This is a Level 3 PBLAS routine that does matrix-matrix multiplication of general matrices. PDGEMM is a standard benchmark that is used for assessing floating point

performance of computers as it measures the Flops (floating point operations per second) accurately. Its complexity is of order n^3.

PDGEMM operates on matrices distributed in a 2D block cyclic layout. Given two matrices A and B, block-cyclically distributed in a p x q grid and two scalars ALPHA and BETA, PDGEMM computes C = BETA * C + ALPHA * op(A) * op(B), where op(A) is either A or transpose of A, and op(B) is similar. Each matrix is described by the DESC data structure discussed earlier. Since we are concerned with 2D block distributions, we adjust the blocksizes of these matrices so that they behave like block distributions.

3.7.2 PDGEMV

This is a Level 2 PBLAS routine that computes the matrix-vector product of a general $n \times n$ matrix or its transpose. Its complexity is of order n^2. Given a matrix A, block-cyclically distributed in a p x q grid, a vector X and two scalars ALPHA and BETA, PDGEMV computes C = BETA * C + ALPHA * op(A) * X, where op(A) is either A or transpose of A. Here again, we modify the blocksizes so that it behaves as a block distribution.

3.7.3 PDGETRF

This ScaLAPACK function was used to demonstrate that some applications are very sensitive to the topology chosen by the processors. PDGETRF computes an LU factorization of a general distributed M-by-N matrix using partial pivoting with row interchanges. The factorization has the form $A = P \times L \times U$, where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$), and U is upper triangular (upper trapezoidal if $m < n$). L and U are stored in A.

3.7.4 Game of Life

Cellular automata [29] are dynamical systems, which are discrete in space and time, whose behavior is specified by a local relation. The automaton consists of a uniform grid of cells each containing a few bits of data. The laws for all the cells are uniform and are given in a lookup table. Based on this table, at each step, the cell computes its new

state from that of its close neighbors (eight neighbors are considered). Each cell is in one of the two states, alive or dead. The Game of Life is a 'cellular automaton', invented by Cambridge mathematician John Conway.

Rules

For a cell that is alive:

Each cell with one or no neighbors dies.

Each cell with four or more neighbors dies

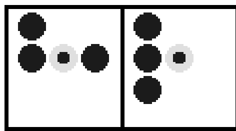
Each cell with two or three neighbors survives.

For a cell that is empty or unpopulated

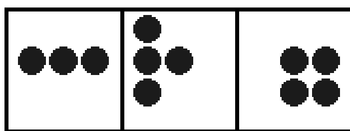
Each cell with three neighbors becomes populated.

The following figures from [30] illustrates the rules

- A dead cell with exactly three live neighbors becomes a live cell (birth).



- A live cell with two or three live neighbors stays alive (survival).



- In all other cases, a cell dies or remains dead (overcrowding or loneliness).

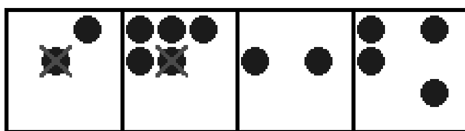


Figure 3.5: Rules in Game of Life application

Depending on the initial conditions, the cells form various patterns throughout the course of the game.

Algorithm

The *n x n* matrix is block distributed in a *p x q* grid.

At each step,

1. The edges of the grid are communicated with its nearest neighbors (left, right, up, down, northwest, northeast, southwest and southeast).
2. The new state for each cell is calculated based on the old state of the cell and its neighbors. Neighboring cells exchange their borders.

The algorithm is $O(n^2)$.

4 Basic concepts

4.1 Research problem

Given n initial processors arranged in a p x q processor grid, if r more processors are added to form the final grid p' x q', the problem is to find a way of placing these additional processors such that the overhead for redistribution is minimum.

We propose a strategy called local remapping which minimizes the redistribution time. We also compare the performance of this strategy with a simple remapping strategy called global remapping.

4.2 Global remapping

This is the easiest way to construct the final set of processors. This method is independent of the arrangement of the processors in the initial set. The processors in the final grid are ordered by their processor ids into a p' x q' processor grid.

Figure 4.1 shows a simple example for global remapping. Fig 4.1a shows the initial processor set containing 4 processors arranged in a 2 x 2 processor grid. Figure 4.1b shows the grid when the application expands to 9 processors with global remapping.

0	1
2	3

Initial processor grid

(a)

0	1	2
3	4	5
6	7	8

Final processor grid

(b)

Figure 4.1: Arrangement of the processors in Global remapping scheme

4.3 Local remapping

In this remapping strategy, the new processors are added as new rows or columns in the processor grid. Figure 4.2 shows the initial processor set with 4 processors and the

final processor set after the application has expanded using local remapping. Note that the processors in the initial grid (0, 1, 2, 3) have not changed their relative positions in their final grid, i.e., processors which were in the same row (column) in the initial grid still remain in the same row (column) in the final grid.

0	1
2	3

a. Initial processor grid

0	1	4
2	3	5
6	7	8

b. Final processor grid

Figure 4.2: Arrangement of the processors in Local remapping scheme

4.4 Comparison of the two remapping strategies

1. Global remapping is simple. Given the total number of processors to which the application has to expand, the final processor grid is easy to construct, as it is independent of the initial processor set. With local remapping, the initial processor grid needs to be kept track of while new processors are added to form the final grid.
2. In local remapping, since most of the processors in the initial grid retain their position in the final grid, the processors retain a large portion of their initial data during remapping. By contrast in global remapping, most processors have to send their data to the new set of processors and should receive a new set of data corresponding to their location in the new processor grid. In short, the amount of data that has to be redistributed is more in global remapping than local remapping.
3. In local remapping, there is sufficient parallelism in data transfer as it requires data to be locally redistributed mostly, i.e., data has to be redistributed only between the processes in the same row or column. In global remapping there exists no such pattern for redistribution.
4. With global remapping, more number of processors can be chosen compared to that of local remapping as the final processor set is not dependent on the number of rows or columns present in the initial processor grid. For example, if the initial processor grid is 10 x 10 and eight more processors can be added, global

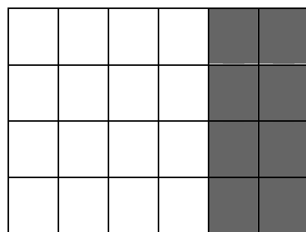
remapping can be done to obtain a 12 x 9 processor grid, whereas with local remapping, we are not able to make use of the computational power of these eight additional processors.

4.5 Placement of the new processors in local remapping

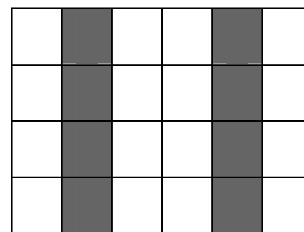
Consider the local remapping strategy. Since the new processors are added only as new rows or columns, these can be efficiently placed in the processor grid such that the redistribution time is minimized. We shall consider a simple way and an optimal way of placing the processors and we shall evaluate the performance of both the strategies.

In the first case (Local 1), the new rows/columns of processors are placed at the ends of the processor grid. In the second case (Local2), they are interspersed between the rows/columns of processor grid. Suppose p is the number of columns (rows) in the initial processor grid and r columns (rows) are added ($r \leq p$). Then p is divided into r divisions and the new set is placed at the center of each division. i.e., the first new column is placed in $\frac{p}{2r}$, the second in $\frac{p}{r}+\frac{p}{2r}$, the third in $2\frac{p}{r}+\frac{p}{2r}$ and in general the nth is placed in the $(n-1)\frac{p}{r}+\frac{p}{2r}$ th position.

Figure 4.3 shows these two different ways of placing new processors in the final processor grid, when p = 4 and r = 2. The gray columns correspond to the new columns of processors.



a. Local1



b. Local2

Figure 4.3: Placement of new processors in Local1 and Local2 algorithms

Let us analyze the amount of data that is transferred in each of these placements. Suppose the initial processor grid is *p x p*. The data present in the matrix is *n x n*. So each processor in the initial grid is allocated a $\frac{n}{p}$ x $\frac{n}{p}$ data matrix. Say, *r* columns (i.e., *p x r* additional processors) are added.

The data columns present in each processor initially is $x = \frac{n}{p}$. After the addition of r columns in the processor grid, the number of columns of data present in each processor is $x' = \frac{n}{(p+r)}$. So, the amount of data columns that have to be moved from each processor is atleast $x - x'$, which is $\frac{n}{p} - \frac{n}{(p+r)}$.

As the problem is symmetrical across the rows, let us consider the first row alone.

Local 1

Consider the Local1 algorithm for the first row of the previous example show in Figure 4.3a.

0	1	2	3	4	5

We assume that if the processor that is responsible for the data before and after remapping are different, then the first processor transfers data directly to the second processor, without going through any intermediate processors.

While redistributing, processor 0 has to transmit $x - x'$ data columns to 1, processor 1 has to transmit 2($x - x'$) columns to 2 and so on till processor *t*, where processor *t* is the first processor that receives x' data from processor *(t-1)* and hence has to transmit its entire *x* columns to the succeeding processors. All the *(p-t)* original processors after *t* have to transmit their entire *x* data columns to the succeeding processors.

So, the total amount of data columns transmitted is $(1+2+3+..+t)\ (x - x') + x(p - t)$, where $t\ (x - x') = x'$. Reducing the above

equation, we get the total amount of data columns transmitted in one row as $\frac{2nr-n}{2r}$.

Therefore, the total amount of data transmitted by all the processors in p rows of the grid is $p.\frac{n}{p}.\frac{2nr-n}{2r} = n.\frac{2nr-n}{2r}$.

Local 2

In the Local2 algorithm, the first row looks as shown.

0	4	1	2	5	3

There are *r* divisions of $\frac{p}{r}$ processors each and each new processor is placed in the $\frac{p}{2r}$ th position of that division. In each division, the processors to the left transmit $x - x'$, $2(x - x')$, $3(x - x')$,... data respectively to the succeeding processors. There are $\frac{p}{2r}$ processors to the left of the new processor. Hence the amount of data transmitted is $(1+2+ \ldots + \frac{p}{2r}) * (x - x')$. Similar is the case for the processors to the right of the new processor. There are r such divisions. Hence the total amount of data columns transmitted by processors in one row of the grid is $2 * r * (1+2+\ldots+\frac{p}{2r}) * (x - x')$, which reduces to $\frac{n}{4} * (\frac{p+2r}{p+r})$. Therefore, the total amount of data transmitted by all the processors in the grid is $\frac{n}{p} * p * \frac{n}{4} * (\frac{p+2r}{p+r})$, which is $\frac{n^2}{4} * (\frac{p+2r}{p+r})$.

It is obvious that the amount of data transmitted in the simple Local1 algorithm will always be greater than the amount of data transmitted in the optimal Local2 algorithm, since $\frac{2nr-n}{2r} > \frac{n}{4} * (\frac{p+2r}{p+r})$, given $p \geq r$.

The following table shows the factor by which the data transmitted by Local1 exceeds that of Local2 for various values of *r* in comparison to *p*.

New columns added	$\frac{Local1}{Local2}$
r = p	2.6
r = p/2	3
r = p/4	3.3
r =1	2

Table 4.1: Amount of data transmitted in Local1 Vs Local2

The factor by which Local1 performs better than Local2 increases up to a certain point as the number of columns that are added is decreased with respect to *p*, and then starts to decrease in the form of an inverted U.

From the above table, we find that the amount of data transmitted in the Local1 algorithm is always more than twice the amount of data transmitted in the Local2 algorithm. In a shared network like ethernet, where only one processor can send data at a time, this factor directly reflects on the time taken for redistribution. That is by choosing the Local2 algorithm over Local1, the time taken for redistribution can be reduced by a factor of two. In other kinds of networks, where data can be transferred in parallel, this factor might not be as directly reflected as in ethernet, but still the difference will be considerable as less data is being redistributed.

4.6 Experiments

4.6.1 Experiment 1: Comparison of Local2 with Local1 and Global

Tables 4.2 and 4.3 compare the redistribution times of Local1 vs Local2 and Global vs Local2 respectively. Time taken to redistribute data of different sizes (represented by N) when the number of processors changes from *p1* to *p2* has been tabulated. The column %imp shows the percentage improvement of Local2 over Local1 in Table 4.2 and over Global in Table 4.3. We have used the Caterpillar algorithm, present in the redistribution library of ScaLAPACK for all our experiments.

Local1 Vs Local2

$p_1 \rightarrow p_2$	N = 12000			N = 18000			N = 24000		
	Loc1	Loc2	%imp	Loc1	Loc2	%imp	Loc1	Loc2	%imp
4 →9	30.982	23.450	32.12	69.177	51	35.64	134	107	25.23
9 →16	27.219	19	43.26	60.512	41.220	46.80	107.48	73.048	47.14
16 →25	21.435	18.117	18.31	47.156	40.101	17.59	83	69.372	19.64
25→36	17	12.694	33.92	37	27.305	35.51	66	48.067	37.31

Table 4.2: Redistribution times for Local1 and Local2 remapping algorithms

Global remapping Vs Local Remapping

$p_1 \rightarrow p_2$	N=12000			N = 18000			N = 24000		
	Glob	Loc2	%imp	Glob	Loc2	%imp	Glob	Loc2	%imp
4 →9	32.240	23.450	37.48	72.401	51	41.96	200.49	107	87.37
9 →16	21.315	19	12.18	47.180	41.220	14.46	84.696	73.048	15.95
16→25	21.723	18.117	19.90	49.662	40.101	23.84	90.145	69.372	29.94
25→36	17.584	12.694	38.52	40	27.305	46.49	71	48.067	47.71

Table 4.3: Redistribution times for Global and Local2 remapping algorithms

Even though the data suggests that Local2 performs better than Global and Local1, we do not get the expected magnitude difference in the redistribution time, because a lot of time is spent in waiting for the other processor to send or receive i.e., on synchronizing (discussed in detail in Section 3.4.1) as demonstrated by the experiment below.

4.6.2 Experiment 2: Time taken for synchronization in Caterpillar algorithm

A 15000 x 15000 data matrix was distributed from 4 to 9 processors, with the processors remapped using Global, Local1 and Local2 algorithms. Tables 4.4, 4.5, 4.6 show the redistribution time taken by each processor (identified by Proc. ID) for Global, Local1 and Local2 algorithms respectively. The time taken for synchronization (i.e., the

time each processor spent in waiting for another processor to send or receive data) is given by T_{sync}. This is the time when no useful work is done. $T_{datatransfer}$ gives the time spent by the processor in transferring data (useful work).

	Global for N= 15000		
Proc. ID	**Tot time Redist(sec)**	**T_{sync} (sec)**	**$T_{datatransfer}$ (sec)**
0	42.4	25.2	15.3
1	41.2	12.3	27.5
2	41.1	0.0	39.8
3	53.9	12.8	39.8
4	19.2	6.3	12.2
5	41.5	28.6	12.2
6	16.5	3.6	12.2
7	41.5	28.4	12.2
8	54.4	41.4	12.2

Table 4.4: Split-up of the redistribution time in Global remapping

Local2 for N =15000			
Proc. ID	**Tot.time Redist(sec)**	**Tsync (sec)**	**$T_{datatransfer}$ (sec)**
0	26.4	9.4	15.3
1	17.2	0.0	15.3
2	16.9	0.0	15.3
3	16.9	0.0	15.3
4	19.4	6.5	12.2
5	25.6	12.7	12.2
6	17.0	4.1	12.2
7	17.4	4.4	12.2
8	14.0	1.0	12.2

Table 4.5: Split-up of the redistribution time in Local2 remapping

Local1 for N = 15000			
Proc. ID	**Tot time Redist**	**Tsync**	**Tdatatransfer**
0	45.2	28.3	15.3
1	57.0	28.0	27.5
2	44.4	15.3	27.5
3	50.6	15.6	33.6
4	25.9	13.0	12.2
5	13.1	0.2	12.2
6	13.3	0.4	12.2
7	57.2	44.2	12.2
8	44.9	31.9	12.2

Table 4.6: Split-up of the redistribution time in local1 remapping

The results show that each processor spends a lot of time in waiting rather than in data transfer. This is because the caterpillar algorithm uses a simple round robin scheme and does not optimize its communication schedule. If the processors did not have to wait for synchronizing, and were involved only in data transfer, then the Global, Local1 and Local2 algorithms would have taken 39.8, 33.6, 15.3 seconds respectively (maximum time in the 4th column), i.e., Local2's performance would have been 160.13% better than Global and 119.61% better than Local1.

We expect to get this kind of performance improvement if the bipartite matching algorithm is used for redistribution. For local remapping, the bipartite redistribution algorithm would be very efficient, as the data in one processor is shared only with a few nodes. There is not much overlap in the communication schedules of the processors and hence a lot of parallelism is possible in data transfer. The main disadvantage of using the bipartite redistribution algorithm is the scheduling cost. During local remapping, the graph tends to be sparse and hence the scheduling cost of the bipartite redistribution algorithm should not be high.

5 Homogeneous Environment

The following scenario occurs while resizing in a homogeneous environment. The application contacts the scheduler at the remap points. Depending on the free processors available and the needs of other applications, the scheduler requests the application to expand or shrink by a certain number of processors. In case of expansion, the application might not be able to make use of all the additional processors available, because of its topological preference. The application calculates the number of processors that it can extend to (based on history from scheduler or from a theoretical model of the application) and intimates the scheduler of its decision. The scheduler creates a schema file corresponding to the new processor set, which the application uses to expand. Similarly, while shrinking, the application might have to release more processors than required. Thus in either case, the application negotiates with the scheduler regarding the number of processors. In case of shrinking, there is a time limit imposed by the scheduler within which the application has to redistribute the data present and release the extra processors. If the application does not release the processors by then, the scheduler may suspend the application.

5.1 Initial mapping of processors

Two decisions have to be made to map the processors to the initial processor grid. They are

1. Topology selection i.e., choosing the topology for the processor grid based on the number of processors available, and
2. Processor mapping which involves mapping the processors onto the processor grid.

5.1.1 Topology selection

Choosing as many processors as possible for a given application will not always lead to reduced computation time. The topology plays a very important role. Table 5.1 clearly evinces this fact. Even a small increase in iteration rate obtained by choosing the appropriate topology will significantly reduce the cost of computation, as the application

goes through a lot of iterations. The choice of the topology depends entirely on the nature of the application.

Some applications work well on any topology. Their performance depends only on the number of processors and hence increases with more processors. For such applications, the topology that selects maximum number of processors is efficient. But there are other applications which are very sensitive to the topology chosen. While some of these might prefer a square processor grid, others which have twice as much horizontal communication as vertical communication might prefer a 1:2 processor grid. Therefore, to maintain a desired aspect ratio, we introduce two variables p, q and constraints such that

$$c \leq p \times r$$

$$r \leq q \times c$$

Depending on the nature of the application, p and q will take different values. For a square processor grid, $p = 1$, $q = 1$. For an arrangement close to square processor grid, if t is the threshold that determines the maximum aspect ratio, then $p = t$ and $q = t$ and the constraint is

$$1/t * c \leq r \leq tc$$

For a 1 : t processor grid, $p = 1/t$, $q = t$,

$$c \leq 1/t * r$$

$$r \leq t * c$$

The values p and q are obtained from the user if the application is simple and the user has information about the nature of the application. Otherwise, they can be determined from the history of the application maintained by the scheduler. The application can enter into a negotiation phase with the scheduler and obtain the iteration rate for the various topologies. The application can then calculate the topology that suits it the best.

Once the topology is chosen, the initial mapping of the processors in the processor grid is simple in the homogeneous cluster case, as the processing power is the same on all machines. The processors are just ordered by their processing ids and placed in the processor grid.

5.1.2 Experiment: Different topologies give different performances

Table 5.1 shows the execution time (in seconds) for the PDGETRF application for a N x N matrix, with different values of N. Assuming the maximum number of processors available is 14, the table shows the performance of the application on 9 processors (exactly square processor grid), on various topologies of 12 processors (as close to square processor grid is possible with 12 processors) and various topologies of 14 processors (all the processors used).

N / *p* x *q*	**3 x 3**	**4 x 3**	**3 x 4**	**6 x 2**	**2 x 6**	**12 x 1**	**1 x 12**	**7 x 2**	**2 x 7**
9000	88	88	77	152	80	369	105	156	74
12000	148	171	156	270	155	654	205	312	162
18000	423	441	430	658	438	1487	541	763	531
24000	916	890	905	1275	907	2711	1123	2069	886
30000	1674	1559	1620	2142	1632	4441	1974	2950	1806
36000	3281	2494	2651	3279	2643	6688	3173	3906	2486

Table 5.1: Performance of PDGETRF on various processor grid topologies

Table 5.1 shows how PDGETRF is sensitive to the different topologies. Neglecting the small differences in execution times, we can see that PDGETRF prefers a close to square processor grid for execution. As we can see from the table 3 x 4 or 4 x 3 topology is preferred over 7 x 2 or 2 x 7, proving the fact that an increase in the number of processors does not necessarily yield an increase in performance. We shall see below factors that influence the decision in choosing the right topology for the PDGETRF application.

Data sizes

From columns 2 and 3 of Table 5.1, we can see that for low matrix sizes, topology 3 x 3 performs better than 4 x 3, because the communication overhead is comparable to the computational gain obtained with 4 x 3. But as data size increases, we find that the computational gain becomes significant and so 4 x 3 performs better.

Aspect ratio of the topology

Aspect ratio is a measure of asymmetricity in the topology. As the aspect ratio increases, the topology becomes more asymmetric. For the same number of processors, PDGETRF prefers processor grids with low aspect ratios. This can be seen from the columns corresponding to 3 x 4, 4 x 3, 6 x 2, 2 x 6, 12 x 1, 1 x 12, topologies in Table 5.1. Twelve processors are used in each case. Topologies 3 x 4 and 4 x 3, with low aspect ratio have low execution times compared other topologies with high aspect ratios.

Comparing topological pairs, 6 x 2 and 2 x 6, 7 x 2 and 2 x 7, 12 x 1 and 1 x 12, we find that PDGETRF prefers a more horizontally dominant grid (has more processors in the horizontal dimension than in vertical dimension) over a vertically dominant grid (has more processors in its vertical dimension than in horizontal dimension).

The same experiment was carried out with the PDGEMM application; the results are given in Table 5.2.

N / *p* x *q*	3 x 3	4 x 3	3 x 4	6 x 2	2 x 6	12 x 1	1 x 12	7 x 2	2 x 7
9000	180	145	159	145	173	197	297	176	167
12000	372	297	332	308	356	394	576	356	341
18000	1244	897	953	916	1025	1610	1542	981	952
24000	2835	1994	2129	3253	2201	3452	3137	2030	2011

Table 5.2: Performance of PDGEMM on various processor grid topologies

We find that PDGEMM does not require an exactly square processor grid, but a close to square processor grid is preferable. 4 x 3 performs better than all other topologies consistently. For topologies with high aspect ratios, PDGEMM prefers a horizontally dominated grid as seen by the execution times of the topological pairs 6 x 2 and 2 x 6, 12 x 1 and 1 x 12, 2 x 7 and 7 x 2.

5.2 Expanding the application

To expand the application, the topology for the final processor grid has to be chosen based on the number of new processors available, and the processors have to be mapped onto the final processor grid.

5.2.1 Choosing the topology

5.2.1.1 Global Mapping

Global remapping is done using the same algorithm used for mapping the initial processors except that the number of processors is now the sum of initial processors and the additional processors available.

5.2.1.2 Local Mapping

Local mapping is an integer non-linear optimization problem. Let r, c be the number of rows and columns of processor grid, x and y be the number of rows and columns of processors that are to be added, n be the additional number of processors that are available, then the optimization problem is to find x, y as solutions to the following problem.

Objective function

Maximize $rx + cy + xy$

Such that

$$rx + cy + xy \leq n$$

As in initial mapping, to maintain a certain aspect ratio, we introduce two variable p, q and constraints such that

$$y <= p * x$$

$$x <= q * y$$

5.2.2 Processor mapping

Once the topology is selected, the new processors are placed as discussed in Chapter 4 for the local and global remapping schemes.

The performance metrics used in evaluating our applications are discussed below.

5.3 Performance metrics

5.3.1 Payoff point

The payoff point [4] for expansion from P_1 to P_2 processors is defined as the number of iterations that the application has to execute to compensate for the cost of resizing.

$$\text{Payoff point} = \left\lceil \frac{t_r}{(t_{p1} - t_{p2})} \right\rceil$$

t_r – Time taken for resizing

t_{p1} - time taken for an iteration on P_1 processors

t_{p2} - time taken for an iteration on P_2 processors

5.3.2 Resize cost

The cost in iterations for resizing, resize cost [4], is the number of iterations that P_2 processors would have executed during t_r. This gives the time when no useful work was done in terms of the number of iterations.

$$\text{Resize cost} = \left\lceil \frac{t_r}{t_{p2}} \right\rceil$$

5.4 Experimental Results

The applications Game of Life, PDGEMV, PDGEMM were run for matrix sizes 21000 x 21000 and 42000 x 42000 initially on a 5 x 5 processor grid. The application was then expanded to 36, 49 and 64 processors. The computation gain obtained by the application by expanding to each of these processor sizes is compared to the redistribution time to find the overhead of resizing. The performance metrics defined above help in evaluating this overhead.

Since the time taken for redistribution is not dependent on the type of application but only on the amount of data that is redistributed and the initial and the final processor grid, we tabulate the redistribution times corresponding to each expansion separately in Table 5.3. The subsequent sections show the time taken for computation by each

application for the different processor sizes. Tglo represents the time taken for global redistribution and Tloc represents the time taken for local redistribution.

p x p → q x q	21000		42000	
	Tglo(sec)	Tloc(sec)	Tglo(sec)	Tloc(sec)
5 x 5 →6 x 6	48.94	41.271	190.90	152.696
6 x 6→7 x 7	53.25	36.652	213.45	142.351
7 x 7→8 x 8	49.98	42.410	199.93	163.122

Table 5.3: Redistribution times with global and local remapping strategies

5.4.1 Game of Life

The Game of life application was run with an N x N matrix distributed across a p x p grid. Table 5.4 shows the execution times for the application with different values of N and *p*.

	Texec (sec)	
p	N=21000	N=42000
5	1.624	6.453
6	1.130	4.473
7	0.833	3.302
8	0.640	2.532

Table 5.4: Execution time for Game of Life Application

Table 5.5 shows the payoff point for Game of Life when the application resizes from *p x p* processors to *q x q* processors using global and local remapping strategies. The percentage improvement offered by local vs global is shown in the column %imp.

p x p → q x q	Payoff point for 21000			Payoff point for 42000		
	Global	Local	%imp	Global	Local	%imp
5 x 5 → 6 x 6	100	84	19.05	97	78	24.36
6 x 6 → 7 x 7	180	124	45.16	183	122	50.00
7 x 7 → 8 x 8	259	220	17.73	260	211	23.22

Table 5.5: Payoff point for Game of Life Application

We find that the payoff point is nearly the same for different matrix sizes. The payoff point for local is significantly less compared to that of global in all cases. The greater the improvement in the redistribution time of the local algorithm, the more will be the percentage improvement in payoff point. Hence by implementing other redistribution algorithms as suggested in Chapter 4, we can expect more improvement in the payoff point.

Table 5.6 shows the resize cost for Game of Life application when the application resizes from *p x p* processors to *q x q* processors using global and local remapping strategies.

p x p → q x q	Resize cost for N = 21000			Resize cost for N = 42000		
	Global	Local	%imp	Global	Local	%imp
5 * 5 → 6 * 6	44	37	18.92	43	35	22.86
6 * 6 → 7 * 7	64	44	45.45	65	44	47.73
7 * 7 → 8 * 8	79	67	17.91	79	65	21.54

Table 5.6: Resize cost for Game of Life Application

Like the payoff point, the cost for resizing also remains the same for different data sizes. Resize cost is directly proportional to the redistribution cost and hence decreases with any improvement to the redistribution algorithm.

5.4.2 PDGEMV

Table 5.7 shows the execution times for the PDGEMV application for different values of N and p.

p	Texec(sec)	
	N=21000	**N=42000**
5	2.1	8.7
6	1.5	5.7
7	1.1	4.2
8	1.8	3.3

Table 5.7: Execution time for PDGEMV Application

The payoff point for each resizing is shown in Table 5.8.

$p \times p \rightarrow q \times q$	**Payoff point for N = 21000**			**Payoff point for N = 42000**		
	Global	**Local**	**%imp**	**Global**	**Local**	**%imp**
5 * 5 →6 * 6	80	68	17.65	64	51	25.49
6 * 6 →7 * 7	144	99	45.45	148	99	49.49
7 * 7 →8 * 8	-68	-58	17.24	203	165	23.03

Table 5.8: Payoff point for PDGEMV Application

The negative values in the payoff point indicate that the execution time after resizing is more than the time on the original processor grid. Resizing has not improved, but on the contrary was detrimental to the performance of the application. The sweetspot of the application (described in Section 3.1.2) for N = 21000, lies between these two processor sizes. But for N = 42000, the sweetspot was not reached and the application continued to offer significant performance gain with increase in the number of processors.

Table 5.9 shows the resize cost for PDGEMV application.

p* x *p* -> *q* x *q	**Resize cost for N=21000**			**Resize cost for N=42000**		
	Global	**Local**	**%imp**	**Global**	**Local**	**%imp**
5 * 5 -> 6 * 6	34	29	17.24	34	27	25.93
6 * 6 -> 7 * 7	50	34	47.06	51	34	50.00
7 * 7 -> 8 * 8	28	24	16.67	62	51	21.57

Table 5.9: Resize cost for PDGEMV Application

5.4.3 PDGEMM

Table 5.10 shows the time taken for the PDGEMM application to calculate the square of a N x N matrix distributed across a *p x p* processor grid, for different values of N and p.

p	**Texec(sec)**	
	N=21000	**N=42000**
5	1146	6778
6	1000	5307
7	1048	4891
8	1116	-

Table 5.10: Execution time for PDGEMM Application

Tables 5.11 and 5.12 show the payoff point and resize cost when the application is resized from a *p x p* → *q x q* processor grid.

p* x *p* → *q* x *q	**Payoff point for N = 21000**			**Payoff point for N = 42000**		
	Global	**Local**	**%imp**	**Global**	**Local**	**%imp**
5 * 5 -> 6 * 6	1	1	0	1	1	0
6 * 6 -> 7 * 7	-2	-1	0	1	1	0
7 * 7 -> 8 * 8	-1	-1	0	1	1	0

Table 5.11: Payoff point for PDGEMM Application

Table 5.11 suggests that the sweetspot of the application lies between 36 and 49 processors. The overhead due to redistribution becomes negligible when compared to the computation gain obtained. The application is able to compensate for the overhead of redistribution within an iteration.

p* x *p* → *q* x *q	**Resize cost for N = 21000**			**Resize cost for N = 42000**		
	Global	**Local**	**%imp**	**Global**	**Local**	**%imp**
5 x 5 → 6 x 6	1	1	0	1	1	0
6 x 6 → 7 x 7	1	1	0	1	1	0
7 x 7 → 8 x 8	1	1	0	1	1	0

Table 5.12: Resize cost for PDGEMM Application

By comparing the payoff point and resize cost for the three applications seen above, we find that for applications with high order of complexity, the performance gain due to resizing is very high when compared to the redistribution overhead. The application can compensate for this overhead within a few iterations. Hence the type of remapping chosen does not affect the performance of the application. However algorithms of lower complexity, e.g., Game of Life and PDGEMV applications which are O(n) algorithms, yield a significant improvement in performance if local remapping is chosen over global remapping. The application has to execute fewer iterations to compensate for the cost of resizing. Hence, the application can resize frequently. In our experiments we added 11, 13, 15 processors in subsequent steps, ie., the processor grid was increased by 44%, 36% and 30% respectively. If a few processors are added to a large initial processor grid, then the payoff point becomes significant as the computational gain that we get due to resizing is not big. Still, we would like to resize and take advantage of the few additional processors that are available. In this scenario, the local remapping algorithm will be of great advantage.

Even though local remapping offers a significant improvement over global remapping, it has its own limitations. With local remapping, sometimes we might not be able to make use of all the processors available, as the number of processors that add on

is based on the number of rows and columns in the existing processor grid. For example, if the number of rows and columns in the initial processor grid is 5 and the additional number of processors that can be added is 3, a local remapping is not possible; but with global remapping we get a 4 x 7 processor grid.

6 Heterogeneous Environment

Consider a heterogeneous cluster environment, where the heterogeneity arises due to the difference in processing capabilities of the machines or due to differences in load on the machines. Now let us examine the challenges that occur while resizing an application in this environment. Processor mapping in heterogeneous clusters is not as simple as in homogeneous clusters, because the processors are allocated a variable amount of data depending on their processing power.

6.1 Need for heterogeneous allocation of data

Most scientific applications are not embarrassingly parallel applications like SETI@home. These applications often require the processors to communicate with each other after each computation step, either for coordinating or sharing data. So, if all the processors are allocated the same amount of data, the performance is limited to the speed of the slowest processor. Experiment 1 (Section 6.4.1) illustrates this scenario. To make use of the high-speed as well as the low speed processors efficiently, the data and work allocated to the processors should be proportional to their processing power.

6.2 Initial mapping of processors

We focus on 2D data arrays distributed across a 2D processor grid. All the processors in a row of the processor grid are allocated the same number of data rows. But the number of data columns allocated to these processors should vary depending on their relative processing power. This is also true for the processors along the same column of the processor grid. So a processor shares the same number of data rows with other processors in its row of the processor grid and the same number of data columns with other processors in its column of the processor grid. Hence, the amount of data allocated might not be exactly proportional to the processing power of the processor. As a result, some of the processors may be underutilized. A mapping algorithm that maps processors to a processor grid should reduce the amount by which processors are underutilized.

Section 3.5.1 provides a description of some algorithms used for mapping

heterogeneous processors in a 2D processor grid. Since we use a modified version of the singular value decomposition algorithm for our experiments, we discuss the algorithm in detail below.

6.2.1 Singular value decomposition (SVD) algorithm

Beaumont et al. [25] have proved that the problem of arranging the processors in a 2D grid such that the idle time of the processors is minimum is NP complete. Given a topology (*p* x *q*) for the processor grid, their heuristic algorithm for placing the processors in the processor grid finds a close to optimal arrangement by arranging the processors into a rank-1 matrix based on their processor powers. The algorithm works as follows:

1. Sort the processors' cycle times and arrange them in the form of a p x q matrix.
2. Given this arrangement of the processors, compute approximate data rows r_i (data columns c_j) that should be assigned to each row (column) of the processor grid.
3. Iteratively refine this arrangement to compute a new matrix that better fits the values of r_i and c_j computed during the second step.

6.2.1.1 Initial arrangement of the processors

The initial matrix (T) is obtained by sorting the cycle times of the processors in ascending order and placing them in a *p* x *q* matrix.

6.2.1.2 Compute approximate values for rows and cols

The matrix rank is the maximum number of linearly independent rows or columns. If T is a rank-1 matrix, it is easy to find a row vector and a column vector such that $r_i \times c_j = t_{ij}$, since now only one independent row (column) of the processor grid has to be considered for allocating the data columns (data rows) to all the processors in the processor grid. Table 6.1 shows an example of a rank-1 matrix.

1	2	3
2	4	6
3	6	9

Table 6.1: Arrangement of processing powers in a processor grid

Since there is only one independent row, the data columns that have to be allocated to each column of the processor grid can be calculated by just considering the processing powers of the processors in the first row. Thus the problem reduces to one-dimension (1D). An efficient dynamic programming approach has been proposed to solve the 1D case by Beaumont et al. [25]. But usually T is not a rank-1 matrix. The singular value decomposition is used to give the best approximation of a given matrix as a rank-1 matrix.

Let $U \sum V^T = S$, be the singular value decomposition of S, where $S_{ij} = \left(\frac{1}{t_{ij}}\right)$.

The best rank-1 approximation of S is given by sab^t, where s is the largest singular value of S, a and b are the left and right singular vectors associated with s. By assigning $r_i = sa_i$ and $c_j = b_j$, the utilization of the processor given by $r_i t_{ij} c_j$ is maintained close to 1. The utilization of all the processors is given by the matrix $U_{ij} = (r_i t_{ij} c_j)$.

6.2.1.3 Iterative refinement

Given an initial arrangement of processors, the above step calculates a close to optimal number of rows and columns that should be allocated to the processors. The refinement step is done to obtain a better arrangement of the processors. The processing power needed to satisfy the row and column vector calculated is obtained by computing the matrix,

$$T_{opt} = \left(\frac{1}{r_i c_j}\right)$$

Based on T_{opt}, the processor cycle times in the initial matrix T are rearranged to better suit the values of r_i and c_j allocated to them. The algorithm converges when no change takes place. Convergence is guaranteed for all inputs.

6.3 Resizing in heterogeneous environment

For expansion in a homogeneous environment, the scheduler specified the total number of additional processes that can be added to the application. Now let us consider

how an application can be expanded in a heterogeneous environment.

For the application to expand, the scheduler has to choose a few processors from the list of free machines that are available and allocate them to the application. As we described earlier, the utilization of a processor depends to a large extent on the other processors in its row and column of the processor grid in general, the other processors already used by the application. The processor grid is known only to the application. So the scheduler is not aware of which set of machines will help the application to run faster. So any help from the application will greatly influence the utilization of the processors chosen, and hence the throughput of the whole system.

Hence to satisfy all these conditions, we propose the following strategy. The scheduler specifies the total processing power (TPP) that can be made use of by the application. It also provides a list of processing powers of the free processors to the application. The application can select any number of processors from the list whose sum of processing powers is less than or equal to TPP. Say TPP is 20. The application can add a few high-end processors, say two processes with processing capability 10, or many low-end processors, say with processing capabilities 2, 5, 5, 8. This gives flexibility to the application in choosing the processors that suit its requirements. The application then specifies the list of processors it has chosen to the scheduler, which allocates these processors to the application. Thus by a simple negotiation step, the scheduler can allocate appropriate resources to the application and increase the overall system throughput as well as the performance of the application.

6.3.1 Selecting the processors

We shall consider how processors can be chosen for each of the strategies, global and local remapping, discussed in Chapter 4. We shall also see how the computation rate of the system varies with the strategy chosen.

6.3.1.1 Global remapping

The application chooses as many high-end machines as possible such that the total processing power of the processors chosen is less than the TPP given by the scheduler. These processors are combined with the initial processor set and the SVD algorithm is

used on the new processor set. The utility matrix (U) defined in Section 6.2.1.2 is calculated. Any new processor that is underutilized, is replaced by another processor of reduced processing power that can equal this efficiency, from the list specified by the scheduler. If more processors can be added now, they are also included and the algorithm is carried out again. The same algorithm, except for the replacement step in the end which checks underutilization of processors, can be used in the scenario where the scheduler allocates processors without application intervention

6.3.1.2 Local remapping

1. Local remapping assumes that the initial processor grid is fairly balanced, i.e., processors are not dramatically underutilized.
2. The new sets of processors are added as new rows or columns in the processor grid.

We have seen that in a row of processors, the data columns allocated are proportional to the processing powers of the processors in each column of the processor grid. Therefore, when a new row of processors is added, their processing powers should also be proportional to the data columns allocated to them. We also have to decide how many rows and columns of processors are to be added and which set of processors from the list are to be chosen.

Step 1: Decision on whether to add along a row or column

When data is allocated to the processors, it is often good to have a low aspect ratio of the data. This is especially useful in applications where processors communicate their borders to the nearest neighbors, as in Game of Life [30]. But some applications might work better in a 2:1 ratio of data rows and columns or in a 1:2 ratio. We shall consider keeping the rows and columns as equal as possible. The other options should be a simple modification to this algorithm.

Algorithm

1. Obtain the data rows assigned to each row of the processors in the processor grid from the SVD algorithm and divide these data rows by the data rows allocated to the

processor that is present in the first column of the processor grid. The ratio is called $colratio$.

2. Calculate the sum of the ratios. Let it be called $colratio_{sum}$.
3. Similarly calculate the *rowratio* and $rowratio_{sum}$ for the processor grid
4. If $rowratio_{sum} > colratio_{sum}$, then the processing capability along the row is greater than that along the column. Hence the new set of processors is added as a new row. Otherwise, the processors are added as a new column.

Step 2: Decision on which set of processors are chosen

Now, we know whether we have to add the new processors as a new row or as a new column. If the processors are added as a new row, then the processors have to be chosen such that their processing powers are proportional to the data columns allocated to each column of the processor grid. So by finding one of the processors in the row, all other processors that should be present in the same row ideally can be found out. We calculate the maximum processor ($proc_{max}$) that can be added to the row and add all other processors based on $proc_{max}$. This is done to ensure that the high-end processors are used effectively. To add a new column of processors, a similar procedure needs to be followed.

Algorithm

1. Sort the list of the processing powers of the processors given by the scheduler.
2. Find the max of the *colratio (rowratio)* and the nearest processor ($proc_{max}$) corresponding to this *colratio (rowratio)*. If the processors are to be added as a new row, *colratio* is considered, otherwise *rowratio* is considered.
3. Find the other processors that have to be added in the same row by calculating $colratio[i] * \left(\frac{proc_{\max}}{colratio_{\max}} \right)$. Processors are added to a column based on the ratio, $rowratio[i] * \left(\frac{proc_{\max}}{rowratio_{\max}} \right)$
4. The processing powers of these new processors are subtracted from the TPP

specified by the scheduler to get the remaining processing power that can be added.

The steps 1 and 2 are repeated till there is no remaining processing power left or until no more processors can be added.

Refinement step

Having obtained the new matrix from the previous steps, the SVD decomposition algorithm is carried out to get the data rows and data columns that are to be allocated to the processors. During the iterative refinement step only the new processors are interchanged.

6.4 Experiments

We modified the Game of Life application for our experiments so that each processor is allocated a different number of rows and columns. An iteration of the Game of Life application consists of a communication step where the neighbors exchange data along their borders and a computation step. To simulate the different processing powers of the processors, the actual computation done by the processors was made a function of its processing power. If PP is the processing power of a processor, then we let the processor execute the iteration $\frac{1000}{pp}$ times in a computation step instead of executing once.

6.4.1 Experiment 1: Motivation for allocating rows and columns according to the processing power

The objective of the experiment is to demonstrate that

1. If all the processors are allocated the same number of rows and columns, the application executes at the speed of the slowest processor, and
2. Low-end processors can be used to significantly reduce the time taken by the application.

A 2000 x 2000 data matrix was distributed across four processors with processing powers 1, 2, 2, 4 respectively. Table 6.2a displays the data distribution when an equal

number of rows and columns are allocated to the processors and Table 6.2b shows the case when processors are assigned rows and columns proportional to their processing power.

Rows \ Cols	1000	1000
1000	1	2
1000	2	4

(a) Equal number of rows and columns allocated

Rows \ Cols	667	1333
667	1	2
1333	2	4

(b) Allocation based on processing power of processors

Table 6.2: Different data distributions for the same processor grid

In Table 6.2a, we can see that all processors have been allocated a 1000 x 1000 matrix (uniform data distribution). Consider the first row in Table 6.2b. The processors are in the ratio 1:2. They are allocated the same number of data rows. But the data columns allocated differ in the ratio 1:2, thus balancing the work done.

The application was run with the above two data distributions. The results are tabulated in Table 6.3. Column 1 in the table displays the processing power of the processors used. Column 2 shows the time taken for an iteration by the application when only processor 4 is used. Columns 3 and 4 tabulate the time taken by the each processor for an iteration corresponding to the first and second distributions.

We can see that for the first distribution, the slowest processor takes 92.30 sec to execute an iteration, whereas processor 4 finishes its execution in 23.32 sec. Though processor 4 finishes before processor 1 in the computation step of this iteration, it has to wait for processor 1 in the communication step of the next iteration. This shows that if the same number of rows and columns are allocated to the processors of different processing powers, the high-end processors are not utilized fully and the application executes at the speed of the slowest processor. Hence, to maintain the efficiency of the application, one would just use the fast processors. But we can see that by assigning a smaller slice of data to the low-end processors compared to their high-end counterparts,

the low-end processors can be used to significantly reduce the execution time of the application. The execution times in columns 2 and 4 clearly bring out the advantage of using low-end processors.

Processing power	Time for an iteration (ms) (proc 4 only)	Time for an iteration (ms) (uniform dist)	Time for an iteration (ms) (nonuniform dist)
4	94.12	23.32	41.548
2	-	46.48	41.256
2	-	46.68	41.556
1	-	92.30	41.356

Table 6.3: Need for heterogeneous allocation of data to processors

6.4.2 Experiment 2: Comparison of different processor mapping strategies

If PP represents the theoretical processing power of the processor, and row, col denotes the number of rows and columns allocated to the processor, respectively, then the time taken for an O(n) algorithm is proportional to the theoretical time T_{Th} = $\left(\frac{row * col}{PP}\right)$. Let P represents the actual processing power of the processor. Since to simulate different processing powers, we executed $\frac{1000}{PP}$ iterations instead of $\frac{P}{PP}$ iterations, we have to also factor $\frac{P}{1000}$ in our theoretical time to get the actual time. This factor is 0.096 corresponding to the processor we used. T_{Th} is used to evaluate the relative performances of the various processor-mapping algorithms. Since this measure reflects directly on the utilization of the processors, it helps us to evaluate how far the algorithms reduce the underutilization of the processors. This measure is valid even if the actual algorithms used for computation are of higher order, except that the differences between the algorithms will now be more pronounced. A random number generator was used to generate the processing powers of the processors.

Experiment 2a:

Processing powers, generated randomly from 10 to 300, were used for this experiment. The initial processor grid had 4 processors The data distribution of the initial processor grid is shown in Table 6.4. Table 6.5 shows the T_{opt} discussed in Section 6.1.1.3.

Rows \ Cols	1115	885
1819	313	279
181	34	25

Table 6.4: Initial data distribution

313	248
31	25

Table 6.5: T_{opt} for Initial data distribution

A 2000 x 2000 data matrix was used. The theoretical time calculated and the actual time taken for the different processing powers are listed below.

Processing power	T_{Th} (sec)	Actual time (sec)
313	622.9	582.7
279	555.8	463.2
34	572.9	549.4
25	621.9	600.4

Table 6.6: Performance of the initial processor grid

We find that the ratios corresponding to the theoretical time and the actual time taken are nearly the same. Hence the theoretical measurement provides a reasonable estimate of the actual time taken.

Expansion with different values of TPP

Thirty more processing powers each in the range 10-300 were generated randomly to obtain the list given by the scheduler. The processing powers generated are 298, 95, 35, 11, 30, 28, 39, 218, 39, 91, 290, 279, 249, 315, 210, 64, 209, 284, 131, 84, 176, 267, 66, 232, 114, 138, 157, 251, 250, 174.

We ran our two remapping algorithms for the list specified above with different values of TPP. Table 6.7 shows the theoretical and actual time taken by the global and local algorithms. The mixed column shows the performance obtained when the global algorithm was run on the processor set obtained from the local algorithm (i.e., the SVD algorithm was executed on the whole processor grid obtained from the local algorithm). The total processing power that was used in each of the cases is also tabulated.

PP	Global			Local			Mixed	
	T_{Th} Time (ms)	Actual time (ms)	TPP used	T_{Th} Time (ms)	Actual time (ms)	TPP used	T_{Th} Time (ms)	Actual time (ms)
1600	188.2	201.1	1597.5	184.0	178.6	1528.2	183.3	179.8
1400	264.9	266.9	1399.8	215.9	229.9	1325.7	214.9	220.5
1200	250.5	243.8	1198.7	217.8	215.6	1180.9	215.9	215.5
1000	274.3	289.8	999.1	273.5	272.6	924.9	272.7	270.5
800	309.6	299.6	799.9	282.1	277.0	760.8	282.1	277
600	357.4	393.6	590.6	326.0	330.7	564	326.2	330
400	488.2	468.9	398.6	396.3	384.2	374.5	396.3	384.2
200	508.5	499.2	186.6	487.3	491.4	161.8	487.3	491.8

Table 6.7: Performance for Global, Local and Mixed algorithms

We find that the local algorithm consistently uses less TPP than the global algorithm, but still gives better performance than the global algorithm. The difference in execution times between the local and mixed algorithms is negligible. This is because all the processors in the initial processor were hardly underutilized and the list given by the

scheduler had the processing powers spread throughout the range.

Let us analyze more closely the case where TPP is 600. In global remapping without replacement strategy, 2 more processors were chosen and a 2 x 3 processor grid was constructed. Table 6.8 shows the distribution for the new processor grid.

Rows \ Cols	1478	302	220
1310	315	284	279
690	313	34	25

Table 6.8: Data distribution for global remapping

Based on the data rows and columns allocated, the actual processing power needed in the processor grid (i.e., T_{opt}) is given in Table 6.9.

315	64.3	47
166	33.9	24.8

Table 6.9: T_{opt} for global remapping

We can find that processors 284 and 279 are underutilized to a very significant extent.

Replacement step of global remapping:

Among the processors that are underutilized (284 and 279), only 284 is a new processor. According to T_{opt}, we find that a processor of PP 64.3 would very well serve the need. Hence we replace 284 with 66, the nearest processing power in the list obtained from the scheduler. For the remaining processing power, two processors (95 and 114) are added instead of one, to avoid having a 7 x 1 grid. The resulting processing grid is shown in Table 6.10.

Cols / Rows	714	683	348	255
1639	313	279	315	114
361	66	34	95	25

Table 6.10 Data distribution for global remapping with replacement

The total processing power used is 590.56 and the time taken is 393.6 sec.

Local remapping:

The algorithm chooses 2 more processors. The final processor grid is shown in the following table.

Cols / Rows	1116	884
950	313	279
95	34	25
955	315	249

Table 6.11: Data distribution for local remapping

Table 6.12 shows T_{opt} corresponding to the processor grid in Table 6.11. As we can see all the processors are nearly utilized to their maximum extent. Time taken for execution is 330.68 ms.

313	247.8
31.3	24.8
314.6	249

Table 6.12: T_{opt} for local remapping

Mixed Algorithm:

Table 6.13 shows the data distribution in a mixed remapping scheme.

Rows \ Cols	1118	882
950	313	249
95	34	25
955	315	279

Table 6.13: Data distribution for mixed remapping

T_{opt} (Section 6.1.1.3) for the above data distribution is given by

313.0	247.3
31.4	24.8
315	248.7

Table 6.14: T_{opt} for mixed remapping

We find that there is hardly any difference between the local and the mixed algorithms and both the remapping techniques take 330 ms.

Experiment 2b

Let us consider a different set of initial processors.

Initial processor mapping

The initial processor mapping considered along with its data distribution and T_{opt} are shown in Table 6.15 and Table 6.16.

Rows \ Cols	1111	889
1197	326	275
803	227	175

Table 6.15: Initial data distribution

325.7	260.4
218.7	174.9

Table 6.16: T_{opt} for initial data distribution

Expansion with different values of TPP

We used the same list of processing powers specified in the previous experiment. Table 6.17 shows the theoretical and actual time taken by global, local and mixed algorithms along with the total processing power that was used in each case.

TPP	Global			Local			Mixed	
	T_{Th} Time (ms)	Actual Time (ms)	TPP used	T_{Th} Time (ms)	Time (ms)	TPP used	T_{Th} Time (ms)	Time (ms)
2000	129.0	123.8	1984.84	133.4	134.8	1975.44	132.2	132.5
1600	160.0	172.7	1599.68	176.4	202.9	1532.38	170.7	197.4
1400	165.0	174.6	1399.82	165.2	174.2	1374.43	165.2	174.2
1200	178.6	188.6	1198.52	184.1	193.6	1152.14	184.1	193.6
1000	297.3	295.2	994.75	204.6	212.8	926.65	204.6	212.8
800	244.7	252.9	794.095	234.8	228.7	725.82	234.8	228.7
600	254.7	256.8	595.81	250.7	246.8	567.88	250.7	246.8
400	299.6	300.4	399.82	298.4	293.7	361.87	298.4	293.7
200	322.2	313.8	193.71	331.9	331.1	189.32	331.9	331.1

Table 6.17: Performance for Global, Local and Mixed algorithms

We find that for TPP values of 2000 (1 x 11), 1600 (5 x 2), 1200 (2 x 4), 200 (1 x 5), global performs better than local. But consider the topology (shown in paranthesis) that global remapping used for each of the TPP. They have high aspect ratios. Local remapping maintains a more or less equal processing power on both dimensions. The gray portion in columns 5 and 8 in Table 6.17 show that the same mapping was obtained for both local and mixed algorithms.

6.4.3 Experiment 3: Application performance across a series of iterations

This experiment was done to see how the application behaves with global remapping and local remapping across a series of iterations. For this experiment, we generated random numbers in the range 0.1 –10 to represent the processing powers of the processors. Twelve processors made up the initial processor grid. The list given by the scheduler consisted of 20 such random numbers.

6.4.3.1 Initial processor mapping

Tables 6.18 and 6.19 show the initial data distribution and T_{opt} of the initial processor grid.

Rows \ Cols	839	695	466
752	9.9	8.8	8.5
735	9.7	8.7	7
358	6.7	3.9	2.6
155	2	1.8	1.7

Table 6.18: Initial data distribution

9.9	8.2	5.5
9.7	8.0	5.4
4.7	3.9	2.6
2	1.7	1.1

Table 6.19: T_{opt} for initial data distribution

6.4.3.2 Expanding the application

The following table shows the performance of the application for 4 iterations. The TPP used in each iteration was generated randomly.

TPP	Global			Local		
	TTh (sec)	**Processor grid chosen**	**TPP used**	**TTh (sec)**	**processor grid chosen**	**TPP used**
30.00	4.2	3 x 5	29.76	4.2	4 x 4	28.25
45.00	2.9	4 x 5	44.71	2.9	5 x 5	43.60
28	2.4	4 x 6	27.98	2.5	5 x 6	22.26
57	1.7	5 x 6	56.77	1.8	6 x 8	54.96

Table 6.20: Performance of the application after series of iterations

We can see that local remapping uses more processors than global remapping. But the TPP values are less than that of global. The section below analyzes the table in detail.

6.5 Analysis of Local remapping algorithm

6.5.1 Advantages

1. We have seen in Chapter 4, that there is a significant reduction in the redistribution time when local remapping is used instead of global remapping.
2. Local remapping makes efficient use of the processors selected. We can see from the above tables that local remapping consistently makes use of lower TPP than global remapping, but still is able to deliver better performance compared to global remapping in most of the cases.

6.6.2 Disadvantages

1. Local remapping uses more processors compared to global remapping. When the processor grid is large and a low TPP has to be added, local remapping chooses many low-end processors, as the number of processors in a row of the initial processor grid is large. This may increase the communication overhead of the application. But since most of the applications are compute-intensive and the communication can be overlapped with computation in most cases, the performance of the application may not be affected greatly.
2. Since processors add only as new rows or columns, in some cases, local

remapping might not be able to take full advantage of the TPP (e.g., 22.26 in row3 of table 6.8). There might be considerable amount of TPP left, but the list specified by the scheduler might not have enough processors to make up a new row with this TPP. Hence the application loses some processing capability, which it could have used. The performance of the application might be impacted as seen in row 4 of Table 6.8. This problem occurs when the processor grid is very large. It will be solved if the scheduler is intelligent and capable of compensating for the missed processing capability of the application by allocating more capability in the next step.

3. To work efficiently, local remapping requires the processing capabilities of the processors to be spread throughout the range so that it can choose the processors corresponding to the ratio of the rows or columns of the previous processor grid.
4. Local remapping works efficiently with a well balanced initial processor grid, i.e., if the processors in the initial grid are only slightly underutilized. If the processors were underutilized to a great extent in the initial processor grid, then a lot of processing power would be wasted if the grid is retained and the processors are added only along a new row or column. Instead a reordering may improve the utilization of the processors.

It is not possible to prejudge the utilization of the processors as processors are selected based on the list obtained from the scheduler. Hence an adaptive scheme that finds out the processor utilization in global remapping and local remapping and chooses the remapping technique that gives the best performance for the application will be very efficient. This would eliminate the problem of local remapping and at the same time would take advantage of local remapping whenever possible.

Choosing the processors that needs to be added to the application is an NP complete problem. In this chapter, we have proposed a heuristic called local remapping, that also helps in reducing the overhead of redistribution. But this heuristic does not give a good performance in few cases. Hence, an adaptive scheme that intelligently shifts between these two remapping techniques would be able to take advantage of both the techniques.

7 Conclusions and Future Work

7.1 Summary

In this thesis, we have provided a library that helps parallel applications to resize efficiently. Only minimal modifications have to be made to the exisiting applications to support resizing. By doing so, these applications can take advantage of the dynamic resource management techniques that help improve the performance of the application as well as system throughput. The close interaction of the application with the scheduler ensures that the scheduler makes a good decision when it decides to shrink or expand the application.

We have implemented our algorithms on top of BLACS. ScaLAPACK calls were used for our experiments. However, the algorithms proposed are not specific to BLACS or ScaLAPACK and can be incorporated on the top of any framework that supports resizing. As MPI-BLACS does not support dynamic process creation, we extended it to support MPI-2. In this thesis, we have considered only structured applications that have 2D arrays distributed across 2D processor grids. The proposed algorithms are simple and can be easily extended to multi-dimensional data arrays and processor grids. Resizing techniques on both homogeneous and heterogeneous clusters have also been discussed. Our results have demonstrated the effectiveness of our approach.

In summary, the contributions of the research described in this thesis include

1. Support provided in BLACS to support dynamic process creation.
2. Efficient algorithms for redistributing data.
3. For a homogeneous environment, algorithms that select the topology of the processor grid that is to be used after resizing and that map the new processors in the existing processor grid such that resizing overhead is minimized.
4. For a heterogeneous environment, algorithms that choose the set of processors that can be added, map this new set in to the existing processor grid and determine the number of rows and columns of data assigned to each processor, such that the processors are utilized to their maximum capacity.

7.2 Future work

We would like to enhance our library in the following areas.

7.2.1 Redistribution algorithm

We have used the implementation of the caterpillar algorithm that is available with ScaLAPACK for redistribution. Chapter 4 discusses the disadvantages of this algorithm. More efficient algorithms like bipartite matching algorithm for block distributions and circulant matrix algorithm for block cyclic distributions have to be implemented to reduce the redistribution time significantly.

7.2.2 Improving topology selection for homogeneous clusters

For a homogeneous environment, we presented a simple heuristic for choosing the topology of the application. The application chooses the topology based on a simple user specified constraint or by a query to the scheduler that gives it the execution time based on its past history. However it is not possible for the scheduler to have information about the performance of the application in all topologies and with all data sizes possible. Intelligent systems like recommender systems can be used to mine the information collected about the performance of the application in various topologies and to predict the performance for a topology that was not considered earlier.

7.2.3 Out of Core

Out of core algorithms have huge amounts of data stored on disks or in other external storage devices. For such applications, the cost of resizing is so high that it can easily offset the performance benefits obtained due to resizing. Our algorithms can significantly reduce the resizing cost of these applications and enable them to take advantage of dynamic resizing.

8 References

[1] G. Swaminathan. Scheduler Support for Dynamically Resizable Applications. Master's Thesis. Virginia Tech, 2004.

[2] T. Schnekenburger and M. Huber. Heterogeneous Partitioning in a Workstation Network. *Proceedings of Heterogeneous Computing Workshop*, pages 72–77, 1994.

[3] ScaLAPACK Project, http://www.netlib.org/scalapack/

[4] J. E. Moreira and V. K. Naik. Dynamic Resource Management on Distributed Systems Using Reconfigurable Applications. *IBM Research Report RC 20890,* 1997. *IBM Journal of Research and Development*, Vol. 41, No. 3, pages 303-330, May 1997.

[5] D. Gelernter and D. Kaminsky. Supercomputing Out of Recycled Garbage: Preliminary Experience with Piranha. *Proceedings of the International Conference on Supercomputing*, ACM, pages 417-427, July 19-23, 1992.

[6] C. McCann, R. Vaswami, and J. Zahorjan. A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors, *ACM Transactions on Computer Systems* 11**,** No. 2, 146-178, May 1993.

[7] A. Gupta, A. Tucker, and L. Stevens. Making Effective Use of Shared-Memory Multiprocessors: The Process Control Approach. *Technical Report CSL-TR-91-475A*, Computer Systems Laboratory, Stanford University, Stanford, CA, 1991.

[8] J. E. Moreira. On the Implementation and Effectiveness of Autoscheduling for Shared-Memory Multiprocessors. Ph.D. thesis. University of Illinois at Urbana-Champaign, 1995.

[9] C. Polychronopoulos. Auto-Scheduling: Control Flow and Data Flow Come Together. *Technical Report 1058*, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, December 1990.

[10] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele, Jr., and M. E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[11] E. Edjlali, G. Agrawal, A. Sussman, and J. Saltz. Data Parallel Programming in an Adaptive Environment. *Proceedings of the 9th International Parallel Processing Symposium*, Santa Barbara, CA, pages 827-832, April 1995.

[12] J. E. Moreira, V. K. Naik, and R. B. Konuru. A System for Dynamic Resource Allocation and Data Distribution. *IBM Research Report RC 20257*, 1995.

[13] C. Huang, O. Lawlor, L. V. Kale. Adaptive MPI. *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing* (LCPC 03), College Station, Texas, Oct 2003.

[14] L.V. Kale and Sanjeev Krishnan. CHARM++ : A Portable Concurrent Object Oriented System Based On C++. *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications,* Sept-Oct 1993.

[15] J. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan. Dome: Parallel Programming in a Heterogeneous Multi-user Environment. *Technical Report CMU-CS-95-137*, School of Computer Science, Carnegie Mellon University, April 1995.

[16] S. Tadepalli. DQ Scheduler. Master's Thesis. Virginia Tech, 2003.

[17] The BLACS Library. http://www.netlib.org/blacs/

[18] LAM-MPI. http://www.lam-mpi.org/

[19] PBLAS. http://www.netlib.org/scalapack/pblas_qref.html

[20] PVM. http://www.csm.ornl.gov/pvm/pvm_home.html

[21] MPICH. http://www-unix.mcs.anl.gov/mpi/mpich/

[22] L. Prylli and B. Tourancheau. Fast Runtime Block Cyclic Data Redistribution on Multiprocessors. *Journal of Parallel and Distributed Computing,* Vol. 45, August 1997.

[23] F. Desprez, J. Dongarra, A. Petitet, C. Randriamaro, and Y. Robert. Scheduling Block-Cyclic Array Redistribution. *IEEE Trans. Parallel and Distributed Systems*, vol. 9, no. 2, Feb. 1998.

[24] N. Park, V.K. Prasanna, C.S Raghavendra. Efficient algorithms for block-cyclic array redistribution between processor sets. *IEEE Trans. Parallel and Distributed Systems*, Vol. 10, No. 12, Dec. 1999.

[25] O. Beaumont, V. Boudet, A. Petitet, F. Rastello, and Y. Robert. A Proposal for a Heterogeneous Cluster ScaLAPACK (Dense Linear Solvers). *IEEE Trans. Computers,* Vol.50, No.10, pages.1052-1070, 2001.

[26] P. E. Crandall, and M. J. Quinn. Block Data Decomposition for Data-Parallel Programming on a Heterogeneous Workstation Network. *Proceedings of the Second*

International Symposium on High Performance Distributed Computing, pages. 42-49, July 1993.

[27] M. Kaddoura, S. Ranka, and A. Wang. Array Decomposition for Nonuniform Computational Environment. *Journal of Parallel and Distributed Computing*, Vol.36, No.2, pages 91-105, 1996.

[28] E. Dovolnov, A. Kalinov, S. Klimov. Natural Block Data Decomposition for Heterogeneous Clusters. *International Parallel and Distributed Processing Symposium* (IPDPS'03).

[29] Cellular Automaton: http://www.brunel.ac.uk/depts/AI/alife/al-ca.htm

[30] Game of Life: http://www.math.com/students/wonders/life/life.html