RICE UNIVERSITY

Mapping HPF onto the Grid

by

Anirban Mandal

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Dr. Ken Kennedy (Chair),
Professor, Computer Science

Dr. Keith D. Cooper,
Professor,
Computer Science

Dr. William Symes,
Professor,
Computational and Applied Mathematics

Houston, Texas

November, 2002

Mapping HPF onto the Grid

Anirban Mandal

Abstract

For this thesis, we have developed a tool for mapping HPF applications onto the Grid using the GrADS [ea02] infrastructure. To build the mapper, the tool makes use of SPMD taskgraph representation of the application. Using the mapper generated, we have been able to launch an HPF application, namely tomcatv, on the Grid. To our knowledge this is the only instance of an HPF application running on the Grid. We have compared the generated mapper with the generic site-aware mapper in GrADSoft. The results show that, for Grid runs of the application on 16 processors distributed over three clusters, our mapper has performance comparable to the generic site-aware mapper in GrADS. On the average, the mapper generated by the tool performs about 5% better than the generic site-aware mapper.

Acknowledgments

I would like to thank my advisor Dr. Ken Kennedy for his excellent guidance throughout the project. I would also like to thank Charles Koelbel for his feedback and insights. My fellow graduate student, Daniel Chavarria, helped me immensely on most aspects of the dHPF compiler. I would also like to thank Dr. Vikram Adve for his help with the taskgraph code. Next, I would thank Mark Mazina for helping me out in most aspects of the GrADSoft software. I would also like to thank the entire GrADS team, especially Holly Dail and Asim Yarkhan for their help in remote system administrator issues. I would especially like to thank my parents and uncle who have always supported and encouraged me in all my academic endeavors.

Contents

Abstract ii

Acknowledgments iii

List of Illustrations vii

List of Tables viii

1 Introduction 1

1.1 Contributions 2

1.2 Organization 2

2 Background 3

2.1 GrADSoft Architecture 3

2.1.1 PPS 3

2.1.2 PES 4

2.2 GrADSoft Application Execution Scenario 5

2.3 Place of GrADS Builder Tools 5

3 Design of GrADS Builder Tool for Mapping HPF 8

3.1 Application Representation and Performance Modeling 8

3.1.1 Task Graph Model 9

3.1.2 Synthesizing the STG 10

3.1.3 Communication Performance Modeling 11

3.2 Mapping 12

3.2.1 Modeling Network Characteristics 13

3.2.2 Mapping processes to processors 13

4 Implementation **18**
4.1 Implementation Flow 18
4.2 Using dHPF compiler and Task Graph Generator 19
4.2.1 dHPF Compiler 19
4.2.2 Task graph generator 20
4.2.3 Omega Calculator 20
4.2.4 Using LEDA 20
4.3 Integration with GrADSoft and Launching HPF 20
4.3.1 Porting dHPF runtime libraries to Linux 20
4.3.2 Using classAds builder tool and Launching HPF 21

5 Experiments and Validation **22**
5.1 Experimental Methodology 22
5.1.1 Testbed 22
5.1.2 HPF Application 23
5.1.3 Generic site-aware mapper in GrADSoft 23
5.2 Validation and Results 24
5.2.1 Proof of Concept 24
5.2.2 Validation of Mapper 25
5.3 Analysis of the Results 25
5.4 Summary of the Results 27

6 Previous and Related Work **29**
6.1 Previous Work on Multiprocessor Scheduling 29
6.1.1 Multiprocessor Scheduling 29
6.1.2 Mapping Task Parallel Flow Applications 31
6.2 Related Work 32

7 Conclusion and Future Work **34**

Illustrations

2.1 Overview of GrADS 4
2.2 Execution Scenario in GrADS 6
2.3 Place of Builder Tools in GrADSoft 6

3.1 An example of a portion of a static task graph 10
3.2 Example: Mapping Algorithm 16

4.1 Implementation Flow 19

5.1 Plot of Total Execution time for the two mappers 26
5.2 Plot of Application Manager Overhead 27

Tables

5.1 Testbed description 23
5.2 Overall Execution Times 25

Chapter 1

Introduction

The Grid is an emerging infrastructure that will fundamentally change the way we think about and use computing [FK99]. This emerging paradigm known as "Grid Computing" — the massive integration of computer systems, will offer overall performance and capabilities exceeding that attainable by any single machine. The Grid vision is that the Grid users will experience the Internet as a seamless computational universe.

Despite this promise, the dynamic and complex nature of the Grid environment poses daunting challenges for application developers. Because of these challenges, applications have been written and run on the Grid only by experts in distributed computing. The principal motivation for the GrADS project [ea02] is to address this issue of making development of Grid programs easier and accessible to people having minimal Grid expertise. The goal of GrADS is to supply a set of tools and infrastructure that will enable easier development of Grid programs. One component of the GrADS effort is developing a set of tools that can discover, without much user intervention, the nature and demand of the application and a strategy to map it to available resources. We call this set of tools in GrADS as the Builder tools.

In this thesis, we propose tools that generate strategies for mapping an HPF application onto the Grid. This is achieved by modeling performance using a SPMD task graph representation of the application. The generated Mapper is used by the scheduler to decide what set of resources the application should run on. We also evaluate the quality of mapping generated by this tool by integrating it with the GrADS infrastructure and running the application on the Grid with the generated

mapping.

1.1 Contributions

The main contributions of the thesis are the following:

- ⋄ Design and implementation of a tool that generates a mapper for HPF applications.
- ⋄ Demonstrating the feasibility of launching an HPF application onto the Grid using the mapping strategy and the GrADSoft infrastructure.
- ⋄ Validating the mapper.

1.2 Organization

The remainder of the thesis is organized as follows. Chapter 2 discusses the background introducing the GrADS framework. It also sets the context of the HPF Mapper Builder tool. Chapter 3 describes the algorithms and approaches used to model application performance model. It also describes the mapping algorithm in detail. Chapter 5 describes the implementation details of the mapper builder tool and its integration with the GrADS code base. In Chapter 6 we discuss the experimental methodology, experiments for validation of the mapper and results of the experiments. Chapter 7 describes related work and Chapter 8 discusses future directions and concludes the thesis.

Chapter 2

Background

Despite the fact that Grid computing promises to be the next big step toward achieving ubiquitous and uniform accesses to computation, data, sensors and other resources, it still is a long way away from gaining widespread acceptance. The primary reason is that current Grid applications are developed on existing software infrastructures like Globus by developers who are experts on Grid Software implementation. The Grid Application Development Software (GrADS) project aims to address this challenge. The vision for GrADS is that the end user should be able to specify applications in high-level, domain specific problem-solving languages and expect these applications to seamlessly access the Grid to find the required resources when needed. To realize this ambitious vision the GrADS project has proposed this software architecture that we will call from now on as GrADSoft architecture.

2.1 GrADSoft Architecture

The following figure 2.1 shows a high level view of the GrADSoft architecture. There are two main subsystems in the architecture — the Program Preparation System(PPS) and the Program Execution System(PES).

2.1.1 PPS

The PPS mainly consists of components having intimate knowledge of the application. The key component in the PPS is the Configurable Object Program (COP) which consists of an AART (Application Abstract Resource and Topology) model, a performance model of the application called the RSE (Resource Selection Evaluator),

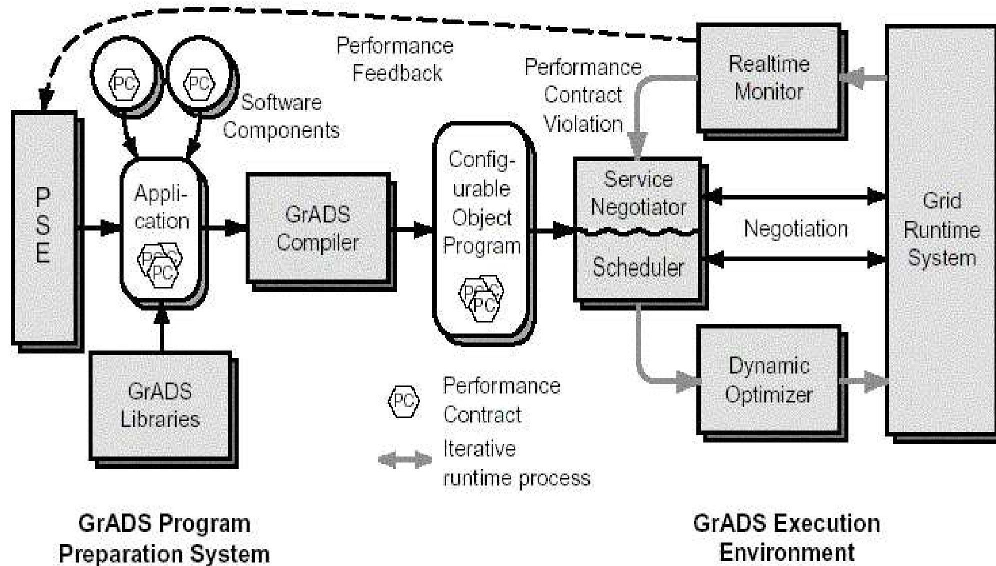

Figure 2.1 : Overview of GrADS

a Mapper and the IR code of the application. When an application is supplied by the end user, the tools in the PPS are responsible for generating the COP. We call these tools collectively as the Builder.

2.1.2 PES

The PES is responsible for launching and monitoring the application running on the Grid. The main components of the PES are the Scheduler/Resource Negotiator, the Binder, the Contract Monitor and the Grid Runtime System. The scheduler uses the COP and information from the Grid Runtime System to select an appropriate resource subset and the Binder is invoked to perform a final resource-specific compilation and insertion of contract monitor sensors. Once the application is launched, the Contract Monitor oversees the progress of the application and reports any violation of performance guarantees.

2.2 GrADSoft Application Execution Scenario

In this section, we describe the application execution flow in the GrADSoft infrastructure. The end user provides the application source with optional annotations for resource selection and run-time behavior to the Builder. Builder is responsible for producing the COP. Next, the user starts the Application Manager. In a nutshell the Application Manager is the process that initiates resource selection, launches the problem run and sees the execution through completion. The Application Manager retrieves the pieces of the COP and invokes the Scheduler. The Scheduler is responsible for choosing Grid resources appropriate for a particular problem run. The scheduler runs an optimization procedure that uses the RSE and the Mapper to find the resources that best fit the application's needs. Once the collection of resources is identified, the Application Manager begins the launch sequence. It invokes the Binder to perform a final compilation and insertion of sensors needed by the performance monitoring component. The Application Manager then starts the Contract Monitor and launches the binaries by invoking the GrADS launcher, a service that is constructed on top of Globus middleware layer. The Contract Monitor reports any violation of performance guarantees to the Application Manager. If the Application Manager determines that the application is not making enough progress, the Rescheduler is invoked to change the course of execution like changing the selected resources or re-assigning workloads. The following figure 2.2 illustrates the application execution flow in GrADSoft.

2.3 Place of GrADS Builder Tools

In this thesis, we are mainly concerned with the Builder part of the PPS. The notion of COP is at the heart of GrADS execution framework for the basic idea that the program must be configurable to be able to run it on the Grid. So the GrADS framework requires the application include both a mapper and an RSE. The Mapper provides a mapping for application components to a given set of Grid resources with

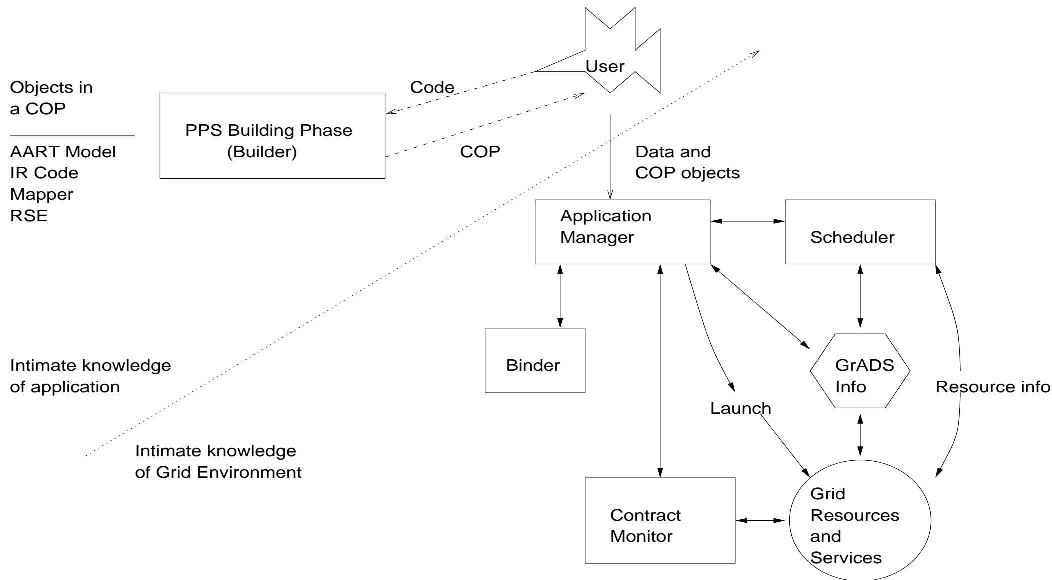

Figure 2.2 : Execution Scenario in GrADS

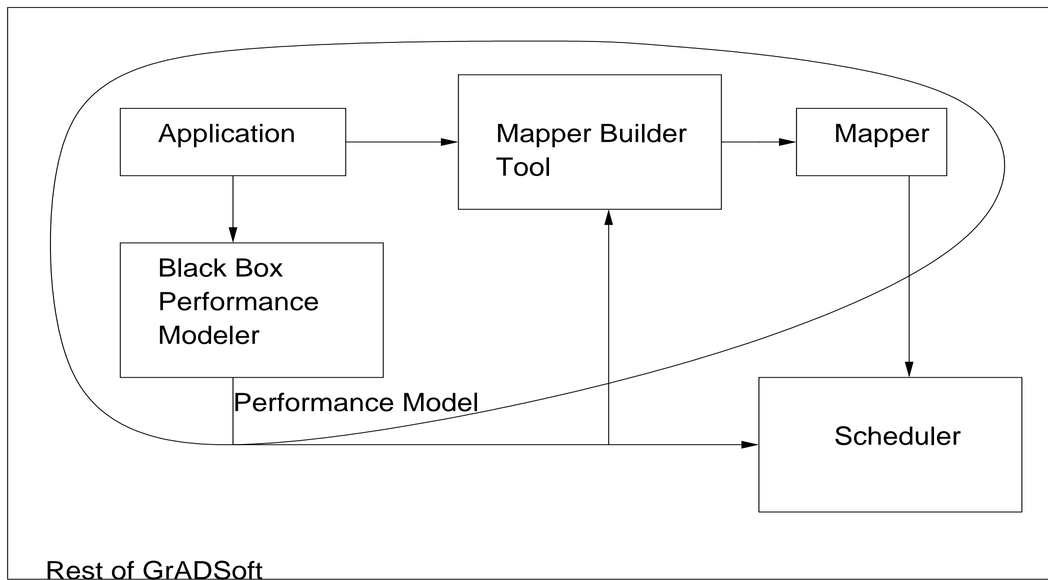

Figure 2.3 : Place of Builder Tools in GrADSoft

the goal of producing good performance. The RSE provides a performance estimate for the application on a given set of resources (assuming the mapping provided by

the mapper) as parts of COP. Since providing the mapper and RSE is a significant additional burden on the developer, constructing tools to ease that burden is an important goal of the Builder Software effort. Figure 2.3 shows the place of the GrADS Builder tool in the GrADSoft framework. The Builder tools are mainly responsible for construction of COP components. The application and optional annotations are shown as input to the Mapper Builder tool and the Black Box Performance Modeler. These two tools are shown to output a mapper and RSE that are used by the GrADSoft Scheduler.

The HPF Mapper has been developed in context of the GrADS Builder Tools. This involves construction of mappers from SPMD task graphs produced from HPF applications. For the purpose of this thesis, we don't use the Black Box Performance Modeler to model application performance. Instead, we use a different performance model derived from the SPMD task graph. We plan to demonstrate this mapper as a proof of concept and would like to accommodate general parallel programs and build mappers for them in future. The next chapter describes our design of the Mapper Builder tool for mapping an HPF application.

Chapter 3

Design of GrADS Builder Tool for Mapping HPF

Application representation and application performance modeling are the two most important prerequisites for designing of mappers for any application. So, at first we describe the application representation that we think is appropriate for mapping an HPF application. Then we describe our algorithm to extract the HPF application performance model from the representation. After that we describe our mapping algorithm, the heuristics we chose to build the mapper and the running time analysis for the same.

3.1 Application Representation and Performance Modeling

There are quite a few general application representations in the literature [Sar93]. Most of them like the Control Flow Graph (CFG) and Static Single Assignment (SSA) representations are targeted toward representing sequential applications. These representations are used extensively for dataflow analysis, different code optimization passes, code generation, code scheduling and interprocedural analysis. These representations are sometimes modified to represent parallel applications [Sar93]. Examples of parallel application representations are task graphs of various flavors. Since, here we are primarily concerned with representation of SPMD code, we chose the static task graph (STG) representation of the application. The static task graph is an SPMD task graph representation which was developed as a part of the POEMS project [AS01, AS00]. In our context, the STG is constructed using parallelizing compiler technologies during the compilation of HPF application to MPI. The Rice dHPF compiler [AJMCY98] constructs the task graph during the SPMD translation

phase of the compilation. Information about the detailed computation partitioning and communication structure of the application is collected in symbolic terms during construction of the STG. The STG is then used to extract the application performance model. We now describe this representation in detail.

3.1.1 Task Graph Model

The static task graph captures the static parallel structure of the application. It is a directed graph representing not only control flow of the application but also communication, synchronization and computation partitioning of the application. Each node of the graph denoting a task may represent one of the following main types: control flow nodes for loops and branches, procedure calls, communication, or pure computation. Edges between nodes may denote control flow within a processor or synchronization between different processors due to communication tasks. A key aspect of the static task graph is that each node in the STG actually represents a set of instances of the task, one per process that executes the task at runtime. Similarly an edge in the STG represents a set of edge instances connecting pairs of dynamic node instances. Symbolic integer sets are used to describe the set of instances for a given node. For example, a task executed by P processes would be described by the set: $\{[p] : 0 \leq p \leq P-1\}$. Similarly, symbolic integer mappings is used to describe edge instances. For example, an edge from a SEND task on process p to a RECV task on process $q = p + 1$ would be described by the mapping: $\{[p] \rightarrow [q] : q = p + 1 \wedge 0 \leq p < N - 1\}$. An example of a portion of a static task graph is shown in figure 3.1. In this example, we have a control flow "DO" node, a pure computation task node, "T" and two communication nodes, "SND" and "RCV". The dark arrows are control flow edges and the dotted arrow is a communication edge. The communication edge is annotated with the symbolic integer mapping corresponding to the send-receive communication event. This kind of mapping enables precise symbolic representations of arbitrary regular communication patterns.

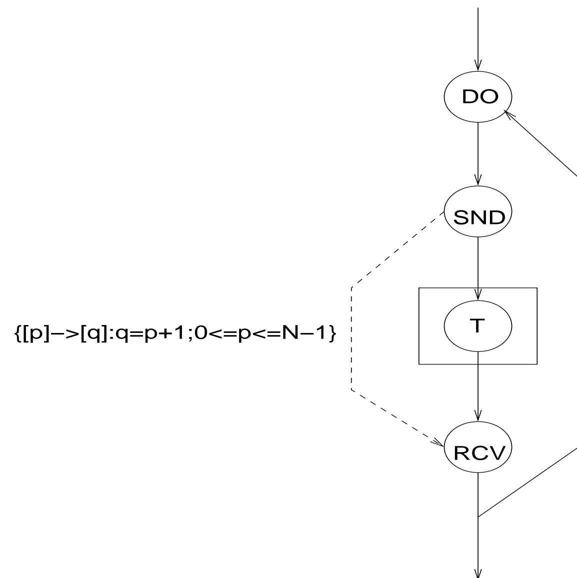

Figure 3.1 : An example of a portion of a static task graph

Irregular patterns are represented as all-to-all communication. To capture high level communication patterns like broadcast etc, communication operations are grouped into related groups, each describing a single "logical communication event". A communication event descriptor, kept separate from the STG, captures all information about a single communication event. This includes the communication pattern, the set of communication tasks involved and a symbolic expression for the communication size.

3.1.2 Synthesizing the STG

Synthesis of a static task graph is carried out in four phases

- Generating computation and control flow nodes: This is done by a single preorder traversal of the AST. Program statements like DO, CALL etc. trigger the creation of STG nodes. Any contiguous sequence of computation statements that are executed by the same set of processes are grouped into a single com-

putation task. To identify statements computed by the same set of processes, information is used from the computation partitioning phase of the dHPF compiler. This information is translated into symbolic integer set that is included with each task

- Generating communication task for each logical communication event

- Generating symbolic sets describing the processes that execute each task, and symbolic mappings describing the pattern of communication edges. The communication pattern information is extracted by recognizing the communication calls syntactically, analyzing their arguments and identifying matching send and receive calls. The symbolic scaling functions for each task and communication event is constructed using direct symbolic analysis of loop bounds and message sizes.

- Eliminating excess control flow edges

3.1.3 Communication Performance Modeling

Once we have the task graph representation for the HPF application, we use it to discover the communication performance model of the application. By communication performance modeling, we mean finding the patterns of the communications in the application and the communication volumes between participating MPI processes for the application.

We represent the application communication characteristics by a complete graph of P nodes where P is the number of processes in the application. We call this graph as the Communication Characterization Graph. Each edge (P_i, P_j) in the graph represents the number of messages transferred from P_i to P_j, the number of bytes per message transferred from P_i to P_j and message startup cost. This graph is constructed using information from the STG. As discussed in the previous sections, the STG for an application contains explicit communication nodes that may represent different

types of communication. The task graph also contains information about communications and communication synchronizations in what is called "Communication Event Descriptors". Each "Communication Event Descriptor" captures a single logical communication event in the application; that is, for each logical communication event in the application, there is a communication event descriptor. This descriptor contains the ID's of the communication task nodes that comprise this logical communication event, the message size and per element size in bytes for the message, the type of communication like pipelined shift, broadcast, all to all communication etc.

The Communication Event Descriptor captures the communication pattern for this event. The edge mappings are key to this since they precisely represent which process is communicating with which other process for that event. We then collect these pieces of information from the descriptor and the corresponding portion of the task graph for each logical communication event. Then we aggregate the information to get the required values for each edge of the Communication Characterization Graph.The following pseudo-code for the algorithm describes our strategy.

```
For each "Communication Event Descriptor" in the Task graph
    Extract CommTaskNodes from the STG;
    Build up the edge-mappings among these CommTaskNodes;
    For each edge mapping
        Find out the sender and receiver processes for the message;
        Aggregate (Psend, Precv) edge in the Comm. Characterization graph;
        Use message size and number of messages to aggregate entry;
    End For
End For
```

3.2 Mapping

The application performance model extracted from the task graph acts as an input to the mapping algorithm. Besides information about the application, the mapper

also needs information about the available resources on which the application will be mapped. More precisely, the mapping algorithm needs to know information about the current set of available processors — at least their available memory and their CPU speed. Also, the network latency and bandwidth between the available processors need to be known. Therefore, before we apply the mapping algorithm, we construct a network characteristic model.

3.2.1 Modeling Network Characteristics

Fortunately, the network and processor characteristics are available almost free of cost from the GrADSoft framework. The latency and bandwidth between pairs of processors are available as two matrices from the GrADSoft framework constructed from Network Weather Service (NWS)and Monitoring and Discovery Service (MDS) data. Each entry in the latency or bandwidth matrix represents the latency/bandwidth between the pair of processors in the network. From these two matrices we construct a Network Characterization graph which is a complete undirected graph whose nodes represent the processors. There are values in each node representing the CPU speed and available memory of the processor. Each edge in this graph denotes the combined latency/bandwidth characteristics for the processor pair. This graph roughly models the network and processor characteristics.

3.2.2 Mapping processes to processors

Once we have modeled both the application and network characteristics the job is to identify which process gets mapped to which processor. We assume for simplicity that the number of processors is equal to the number of processes and the problem is to find a permutation of processes on to the given processor set. We also assume even work allocation among the processors, though the present algorithm can be easily extended to handle work allocation proportional to computational power of the processor.

3.2.2.1 Mapping Philosophy

In Grid environment, high latency between processors belonging to different geographically distributed clusters is one of the greatest problems in mapping tightly coupled applications. We think that latency will be a bigger issue in Grids than bandwidth. So, the motivation behind choosing a good mapping heuristic was that we needed to use the lowest latency links for processes having most communication between them. The best-fit heuristic looked the most obvious in this context. Another approach we considered but didn't implement is to model the mapping problem as a linear programming problem and to maximize some mapping coefficient. But since, the best-fit heuristic was both very intuitive and easy to implement, we chose the same.

3.2.2.2 The best-fit heuristic

We take a graph growing best-fit heuristic strategy to do the mapping of processes on to processors. We choose the process pair having the most stringent communication needs and map them to the processor pair having the best latency/bandwidth characteristics. Now this pair having fixed, we grow the graph with the next most stringent communication need and map it to the next best processor pair satisfying all the present constraints. We expect that this heuristic will automatically map all processes to a single cluster if the number of processes is less than the total number of processors connected in a cluster. Note that these processors will have a very favorable entry in the Network Characterization graph.

3.2.2.3 Mapping Algorithm

We now describe the steps of the mapping algorithm. The inputs to the algorithm are the Communication characterization graph and the Network characterization graph.

- ◇ The combined latency/bandwidth characteristics between a processor pair is calculated in the following way. For each link latency, we normalize it against

the maximum of the link latencies. We call it NL. For each link bandwidth, we normalize it against the link with maximum bandwidth. We call it NB. We have two fractions A and B which sum up to one to give relative importance to bandwidth and latency. The combined metric M is expressed as $M = A*NB + B*(1-NL)$ which is a simple linear model. Higher the value of M, better the communication characteristic between the two processors. We can vary the values of A and B to get different mappers. The default case is a higher value for B since we think that latency is a bigger issue on the Grid than bandwidth.

◇ The edges in the Network characterization graph are sorted according to the metric M. The edges in the Communication characterization matrix are sorted according to the communication volumes of the edges.

◇ Until all processes are mapped onto processors, we pick the process pair with highest communication volume and map it onto the processor pair having the best M value. We keep doing this and at each stage we satisfy the mapping obtained at the previous step.

The output of the algorithm is a permutation of processes onto processors. Here is the pseudo-code of the algorithm.

```
Input: Comm. characterization matrix and network latency and bandwidth
       matrices
Find maximum latency and bandwidth values from the two matrices;
Normalize the latency and bandwidth and find the M values for each edge;
Sort edges of Network characterization matrix on M;
Sort edges on Comm. characterization matrix on communication volume;
Until all processes mapped
    Pick the next edge in Comm characterization graph in sorted list
    If !(both nodes in the edge mapped)
        If both not mapped
```

```
            Map to the next edge in Network characterization graph;
        else if one of them mapped
            Map the the next edge with one end at the mapped node;
    else
        Pick the next edge and continue
End Until
Output: Permutation of processes on to processors
```

The following figure 3.2 illustrates an example of a run of the mapping algorithm on a toy graph. The leftmost graph denotes the input Communication characterization graph. The graph at the center is the Network characterization graph. The rightmost graph shows the output permutation of processes onto processors.

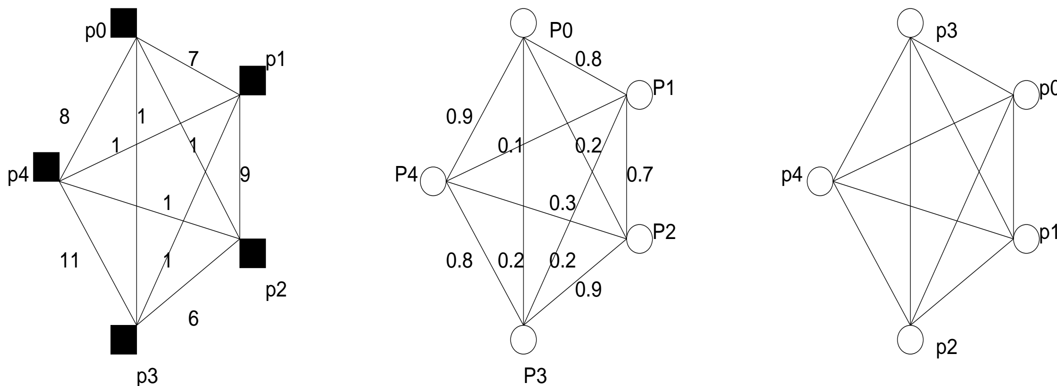

Figure 3.2 : Example: Mapping Algorithm

3.2.2.4 Running Time Analysis

Normalizing the matrices and finding the M values both take $O(E)$ where E is the number of edges in the Network Communication graph. Also the final matching step requires $O(E)$ time. The second sorting step takes $O(E)$ time if we use a bucket sort kind of linear algorithm. We can use bucket sort because we can give as input to

the sorting algorithm the range of the values on the edges. Since E is $O(P^2)$, the overall algorithm is $O(P^2)$. Basically the running time is controlled by the sorting step. So the actual running time of the algorithm depends on the running time of the sort routine in the LEDA library we used to perform the sorting of the edges. It is not clear from the documentation of the LEDA manual which sorting algorithm is used and what is the running time for the same. So, we are not sure whether their implementation achieves that bound or not..

3.2.2.5 Limitations of the Mapping Strategy

The main limitation in our overall mapping scheme is that, we take an aggregate view of the communication in the application and try to map according to that. But, for a more efficient mapping scheme, we also need to look at the temporal aspects of the communications in the application and also interaction between communication and computation. Basically, we lack in a more accurate performance model. The second limitation comes from the assumption that we map a single process onto a single processor. To get a better mapping, we ought to relax this condition. The third limitation is that we don't try to balance computation among the processors. Load balancing is an important aspect and we should incorporate that in the mappng scheme.

Chapter 4

Implementation

In this chapter we describe the implementation of the builder tool that generates a mapper for an HPF application. The tool uses the Rice dHPF compiler [AJMCY98] infrastructure to generate the task graph and the MPI code from the HPF application. Once the mapper has been generated, the GrADSoft infrastructure is used to incorporate the generated mapper and launch the application on the Grid.

4.1 Implementation Flow

The HPF application is compiled to the corresponding MPI program using the dHPF compiler. The libraries responsible for generating the STG are linked during this compilation. This process generates the MPI code with calls to dHPF runtime libraries and also the STG for the application. From the STG of the application, the tool builds up the symbolic integer mappings corresponding to the communications between the MPI processes. These integer mappings are tuples expressed in symbolic terms, which if enumerated, will output the process pairs and the communication volume between them. The integer mappings are fed to the Omega calculator to enumerate the tuples. The tool then uses this information and the network bandwidth/latency information from the GrADSoft code base to generate the mapper. The tool uses LEDA routines to do graph construction and manipulations during the mapping phase. To launch the HPF application, the mapper is integrated with the GrADSoft framework. Also the HPF runtime libraries were ported to the Linux platform. The ClassAds [Uni] COP was modified to incorporate the mapping strategy generated by the tool. By ClassAds COP, we mean the mapper and performance

model obtained using a ClassAds translator tool that translates ClassAds directives to GrADSoft components. Then the application is launched by the GrADSoft Application Manager. We describe each of these steps in details below. The following figure 4.1 illustrates a high level view of the implementation flow.

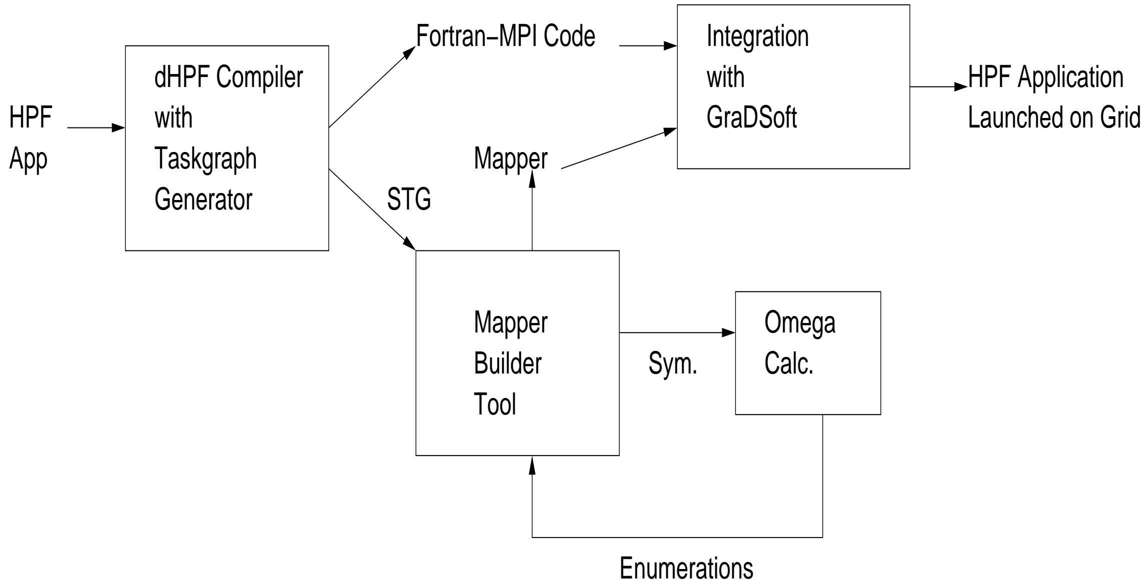

Figure 4.1 : Implementation Flow

4.2 Using dHPF compiler and Task Graph Generator

4.2.1 dHPF Compiler

The Rice dHPF compiler is a parallelizing compiler that has been developed with the aim of narrowing the gap between the performance of data parallel languages like HPF and hand-coded programs. Also, the dHPF project has developed optimization techniques for a range of parallel architecture classes like message passing, shared memory and cluster systems to gain wider acceptance. In our context, we have used the dHPF compiler to compile applications written in HPF to message passing code written in Fortran-MPI.

4.2.2 Task graph generator

The task graph generator is a set of library routines that are linked with the dHPF compiler. The task graph generator uses the parallelizing compiler technologies of the dHPF compiler to generate the STG for the application. More specifically, it uses the computation partitioning and communication generation phases to extract the symbolic integer sets and symbolic integer mappings. In our context, we have used the task graph generator to build the STG of the application.

4.2.3 Omega Calculator

The dHPF compiler is based on a powerful, abstract integer-set framework for analysis and optimization of regular, data-parallel programs. The framework is implemented using the Omega library [KMP+] from the University of Wisconsin, Madison. Omega library also includes an Omega calculator which is useful to generate code that enumerates the symbolic integer mappings. In our context, we have used the Omega calculator to enumerate the tuples of process pairs involved in communication.

4.2.4 Using LEDA

LEDA, Library for Efficient Data structures and Algorithms [MN99], is a library of efficient routines for almost all common data structures and algorithms. In our context, we have used LEDA to construct and manipulate graphs during the mapping process.

4.3 Integration with GrADSoft and Launching HPF

4.3.1 Porting dHPF runtime libraries to Linux

The dHPF runtime libraries had been originally compiled and linked on the Sun machines. But the GrADSoft code base has been developed on Linux at present and

we wanted to launch the HPF application using the GrADSoft infrastructure. So the dHPF runtime routines needed to be ported to Linux.

4.3.2 Using classAds builder tool and Launching HPF

The ClassAds structure is a flexible and extensive data model that can be used to describe an application's resource requirements. In GrADS we have developed the ClassAds builder tool that translates the ClassAds to GrADSoft components. The motivation is to support this widely used, simplistic ClassAds model in GrADS framework. More specifically, the translator converts the ClassAds representation to s-expressions that correspond to functional attributes in AART, like whether resources are sufficient or not etc. The generic COP components, specifically the RSE and the Mapper, use these s-expressions to incorporate the classAds directives. In our context, we have used the classAds tool in two ways. First, to express that we want to run the application on some fixed number of processors, we have specified an additional 'count' attribute in the classAd and forced the mapper to use 'count' number of machines. Second, we have modified the generic COP to incorporate the mapping strategy. The compiled HPF application is then launched onto the Grid using the GrADSoft infrastructure.

Chapter 5

Experiments and Validation

In this chapter, we describe our experimental setup and results of validation of the mapper generated by our tool.

5.1 Experimental Methodology

5.1.1 Testbed

The experimental testbed was the GrADS testbed. GrADS testbed has approximately 40 nodes spread across University of California at San Diego (UCSD), University of Illinois at Urbana Champaign (UIUC), University of Tennessee at Knoxville (UTK), Rice University (Rice), Indiana University (IU) and University of California at Santa Barbara (UCSB). For this thesis, we have run our experiments on clusters at three testbed sites — UCSD, UTK and UIUC. Table 5.1, borrowed from Holly Dail's Master's thesis [Dai02], summarizes the machine characteristics in these clusters. For the Mapper Builder tool, we have used two machines from Rice: basuri.cs.rice.edu, an eight processor, 333 Mhz each, SMP running Sun was used for HPF compilation and taskgraph generation and ural.owlnet.rice.edu was used to develop most of the mapper builder tool components. The required softwares at the clusters at three sites were Globus 2.0, MPICH-G v. 1.2.4 and NWS/MDS services. Omega and LEDA libraries were required in the Rice machines.

For timing the application, we put mpi_wtime() calls at different points of the MPI code. We find iteration times and the whole application running time in this manner. This time does not include the time required by the GrADS services. To capture the total time required from program preparation to completion of program

	UCSD cluster1	UTK cluster1	UIUC cluster1	UIUC cluster2
Size Used	5	5	4	8
Domain	ucsd.edu	cs.utk.edu	cs.uiuc.edu	cs.uiuc.edu
Names	dralion, nouba quidam, soleil saltimbanco	torc0-torc4	opus13-m-opus16-m	[a-h]major
CPU	450 MHz PIII	550MHz PIII	450MHz PII	266 MHz PII
Memory	256MB	512MB	256MB	128MB
OS	Debian Linux	Red Hat Linux	Red Hat Linux	Red Hat Linux

Table 5.1 : Testbed description

execution, we use the Unix 'time' utility.

5.1.2 HPF Application

The HPF application we chose for our experiment was **tomcatv**, a vectorized mesh generation program. It is one of the programs in the Spec CFP'95 benchmark suite. It generates a two dimensional, boundary fitted coordinate system around general geometric domains. The application calculates residuals, finds the maximum value of the residuals and then solves a tri-diagonal system in parallel. It iterates until convergence. The analytical worst case running time for the application is $O(N^2)$, where the matrices are of size $N \times N$. This application can be parallelized reasonably well. tomcatv is very memory intensive . It is very sensitive to the speed of the memory system.

5.1.3 Generic site-aware mapper in GrADSoft

The generic site-aware mapper in GrADSoft groups the available processors into sites they belong to. Sites are generally the domains the machines belong to. Next it maps

the processes (from 0 to number of processes) onto the processor set that has been ordered by sites. This mapper is called generic because it doesn't analyse the communication needs for the specific application that needs to be mapped. It also looks at only one hierarchical level as far as communication characteristics are concerned. But still, this generic mapper does pretty well in applications where there are a lot of nearest neighbor communication. That is because 'nearby' processes are mapped to processors belonging to the same cluster in most cases.

5.2 Validation and Results

5.2.1 Proof of Concept

We are not aware of any other work that has attempted to launch an HPF application on the Grid. We have been able to launch the tomcatv application across three clusters on the GrADS testbed. We have been able to launch the application using 4, 8 and 16 processors across the three clusters. We have also run the experiments with different data sizes — matrices of size 1024, 2048 and 4096. All these experiments were repeated for two mapping cases; one that uses the generic GrADSoft site-aware mapper and other which uses the mapper generated by the Mapper Builder tool. Table 5.2.1 shows the execution times in minutes for each of processor number-datasize cases and uses the Builder Tool mapper. By execution time we mean the total time it takes for the Application Manager to launch the application + execution time of the application. When data size was 4096, experiments with 4 and 8 processors took a long time and sometimes didn't finish. In most cases the reason was that one or some of the processes ran out of memory in the machines they were running on. The Linux kernel would kill those processes in such cases. This lead to the bad situation where the rest of the processes were waiting for some communication from the killed processes. This meant that the other live processes would not terminate.

	1024	2048	4096
4 processors	3.76	11.13	–
8 processors	4.16	12.16	–
16 processors	3.80	7.56	23.68

Table 5.2 : Overall Execution Times

5.2.2 Validation of Mapper

In this section we compare the two mappers. Also we demonstrate that the Application Manager overhead is minimal. Figure 5.1 shows execution time plots for the three problem sizes, each running on 16 processors. For each problem size, the two bar diagrams correspond to two different mappers used. The left bar corresponds to the total execution time for the Builder tool Mapper and the right bar corresponds to the generic GrADSoft Mapper. From the plot, it is clear that the builder tool mapper performs about 5% better than the generic mapper. The next diagram 5.2 shows the Application Manager overhead time. Each bar has two portions. the upper portion is the Application Manager Overhead time and the rest is the actual application execution time. The graph shows that the application Manager overhead is in the range of a constant time of around 2 minutes.

5.3 Analysis of the Results

The results show that the computational complexity of the application runs is $O(N^2)$ where N is the size of the matrix. For a fixed number of processors, when the matrix size is doubled, the running time increases to approximately four times. For example, for matrix size 2048 the application execution time is about 5.84 minutes while for matrix size 4096 the application execution time is about 22 minutes. This is expected, since analytically, the tomcatv application is $O(N^2)$. So the application scales at least

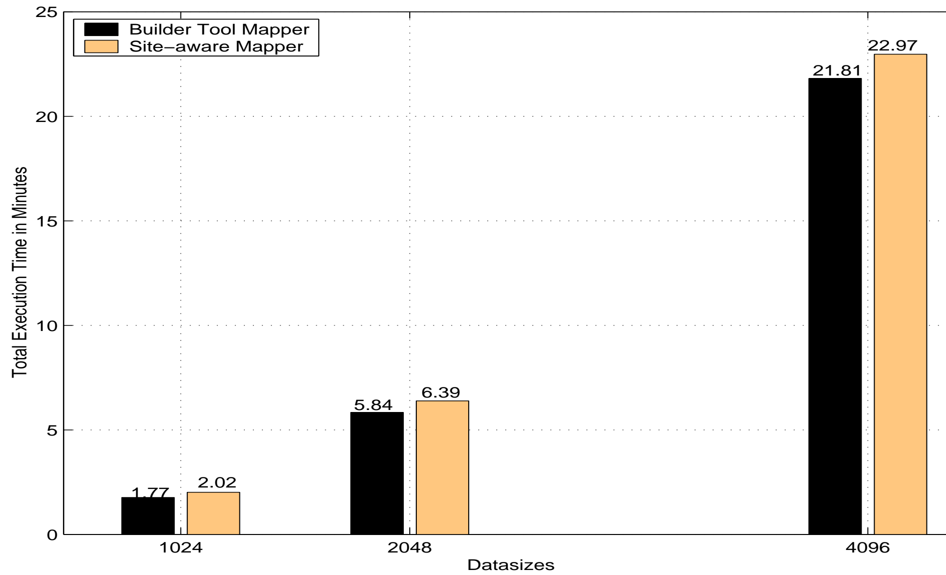

Figure 5.1 : Plot of Total Execution time for the two mappers

for upto 16 processors.

The mapper generated by the tool didn't perform appreciably better than the site-aware GrADSoft mapper mainly because of the following reasons. First, there are limitations in our communication performance model. The problem lies in the step where we are aggregating the communication volume. Because of aggregation, we lose the temporal aspects of communication. We don't perform well when there are many communication steps, each having very small volume. The tomcatv application has a significant number of cases where there are many small volume, nearest neighbor communications. The site-aware mapper does fairly well in such cases. Second, the GrADS testbed on which we ran the experiments consists of machines from three sites. This means there are only three different types of latency/bandwidth in the processor links in the Network Characterization Graph. This does not give much scope to the mapping heuristic. If we had many different types of latency/bandwidth in the testbed, the Builder Tool Mapper should produce permutations that would perform

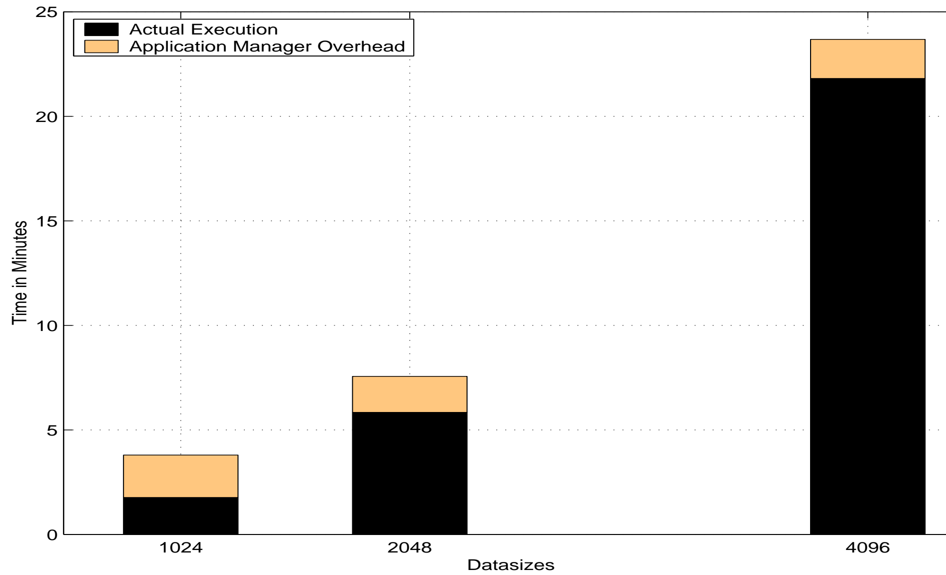

Figure 5.2 : Plot of Application Manager Overhead

much better than the generic mapper. Note that, the site-aware mapper would still group processors into sites and would generate almost the same permutation.

We ran our experiments to include machines from the three clusters. We expect that the running time of the application when it is run on a single cluster with the same number of procesors would be less than the when run on the Grid of three clusters. The comparision makes sense only when we have machines having same CPU speed, memory etc. in both experiments — the intra-cluster one and the inter-cluster one.

5.4 Summary of the Results

As a proof of concept, we were able to prepare an HPF program and launch it on the Grid using a generated mapper and the GrADS execution framework. The mapper generated by the Mapper builder tool has performance comparable with the generic GrADSoft mapper, which is itself fairly intelligent. The results also show that the Application Manager overhead for launching the application on to the Grid is minimal.

This validates the overall GrADS approach.

Chapter 6

Previous and Related Work

6.1 Previous Work on Multiprocessor Scheduling

In this section we look at existing methodologies to map applications to parallel machines. Most of the times, researchers are primarily interested in either of these two metrics when they consider mapping and scheduling an application.

- ◇ Reducing the parallel time of the whole execution of the application. This is true for all the three categories of applications running on the grid we have seen so far.
- ◇ Increasing the throughput of the application. There are certain flow applications that may definitely run on the Grid in very near future, where increasing the throughput is the primary concern.

6.1.1 Multiprocessor Scheduling

There has been a host of research on the topic of multiprocessor scheduling. The main goals of these efforts has been reducing the parallel execution time and minimizing communication. We now study the broad approaches and taxonomy in this field. In most cases the parallel program is represented by a directed acyclic graph (DAG) G = (V,E) where V is the set of nodes and E is the set of directed edges. A node in the DAG represents a task which in turn is a set of instructions which must be executed sequentially on the same processor. Each node has a weight which denotes the computation cost or the processing time. The edges in the DAG correspond to the communication messages and precedence constraints among the nodes. The weight of

an edge is the communication cost incurred if the two tasks are assigned to different processors.

Partitioning and scheduling a DAG is known to be NP-Complete. Hence heuristics are required to find sub-optimal solutions. Based on these, most methods can be classified into the following three categories.

- ◇ *Critical Path Heuristics*: For DAGs with edge weights and node weights, a path weight is defined to be the sum of the weights of both nodes and edges on the path. A critical path is a path of greatest weight from a source node to a sink node. These algorithms try to shorten the longest execution path in the DAG. Paths are shortened by removing communication requirements (zeroing edges) and combining the adjacent tasks into a grain. Important algorithms in this category are DSC(Dominant Sequence Clustering), Linear Clustering and MCP(Modified Critical Path) algorithms.

- ◇ *List Scheduling Heuristics*: These algorithms assign priorities to tasks and schedule them according to a list priority scheme. For example a high priority can be assigned to tasks with large number of incident edges or whose neighbors have already been scheduled. They generally use greedy heuristics to schedule tasks in a certain order. Sometimes task duplication is also used to minimize communication. An important algorithm in this class is the LAST algorithm.

- ◇ *Graph decomposition method*: Based on graph decomposition theory, this method parses the graph into a hierarchy of subgraphs representing the independence and precedence relationship among groups of tasks. Communication and execution costs are applied to the tree to determine the grain size that results in the most efficient schedule. An important algorithm in this class is the CLANS algorithm.

This classification is due to [KMG93]. Another taxonomy based on the context of

scheduling DAGs is given in [Kwo97]. A comprehensive survey of most DAG scheduling algorithms is also available in the same.

6.1.2 Mapping Task Parallel Flow Applications

Another class of applications that are becoming very important are the flow applications like Video compression, Color tracking etc. The goal in scheduling this type of application is to increase the throughput of the application. Here exploiting task parallelism is very important. Inputs in these applications generally consist of a stream of data which undergoes several levels and types of processing. The nodes of a task graph in this case represent a whole task [which may itself be data parallel] and the edges represent the input/output and data sharing relations between different tasks. A point to note here is that the granularity of tasks in this case is much higher than the tasks in the traditional multiprocessor DAG. Also there is the concept of unit progress that can be defined as completion of processing of a discrete stream of data through all levels of processing in the task graph. The goal is to map these tasks on to processors in a multiprocessor system so as to maximize throughput and minimize the response time of unit processing.

An example of such a system is the Smart Kiosk system developed at the Cambridge Research Lab[KRC+99, RKR+98] Smart Kiosk is a computationally demanding multimedia application involving vision, speech and interaction with the real world. It is highly task parallel and requires low latency for good performance. In addition, this application is highly dynamic and some of the tasks also exhibit data parallelism. The abstract execution model of this application is that all tasks in the task graph are running in parallel and they interact through channels which hold unprocessed/partially processed video frames acting as input/output for the tasks.. While mapping a task it is also kept in mind whether the task itself is data parallel or not.

Given this general application structure it is natural to incorporate pipelined

parallelism into the mapping strategy for mapping these flow applications. We call these applications flow applications since a data stream flows and mutates through the task graph while getting processed. Also there has been work on processor assignment for general pipelined computations using series-parallel tasks [CNNS94] that also tries to maximize throughput and minimize response time of pipelined computations.

The existing heuristic strategies[TC00] used to map flow applications try to achieve a good throughput of the whole data processing pipeline, taking both parallelism(load balance) and communication volume(locality) into account. As communication becomes a bottleneck, they trade parallelism for reduction in communication traffic. The general method being clustering the task graph and then ordering the tasks and scheduling them incrementally onto the system graph to obtain an initial mapping. Then the mapping is improved iteratively so as to maximize a throughput function.

6.2 Related Work

The work closest to what we want to achieve is [OD01] that describes a task mapping technique for communication aware scheduling strategies. They describe a mapping technique that takes into account both existing network resources and traffic generated by application. They collect execution traces of the MPI application to model the MPI application's traffic pattern [OAD00] and also model the network resources[AORD99]. They represent these two classes of information in two matrices and use a genetic algorithm to find a good permutation (mapping of processes to processors) that maximizes a Mapping coefficient [a measure of network communication] that is a function of entries in the two matrices. The assumptions that are different in our case are we want to model the application communication traffic from the application source and not from execution traces. We also have a different mapping strategy.

The Information Power Grid project [BBS+] of NASA also aims to map tightly

coupled computational fluid dynamics application to the Grid. But their strategy is very application specific since they modify the application itself to gain latency tolerance and load balance. Our approach is based on compiler analysis and we do not modify the application.

The following work [DCB02] describes mapping iterative stencil applications on to the Grid using GrADS framework. The work uses a linear programming strategy to find the best data distribution and locality to determine the mapping from processes to processors. It also uses equational performance models for the applications to guide the linear programming solver. Our work closely relates to this work, the difference being we have built a tool to build the mapper through compiler analysis of the application. Also our approach doesn't assume any intimate knowledge about the application.

Chapter 7

Conclusion and Future Work

We have built a tool that constructs mappers for HPF applications. Using the mapper generated we have been able to launch HPF applicationis onto the Grid. We have used the GrADSoft software as a framework for launching the application. The mapper generated by the tool has comparable performance with the GrADSoft generic mapper.

This work was developed as a proof of concept of a Mapper Builder tool. At present, this tool works only for HPF applications. We need to be able to map a wider range of applications written in a wider range of programming paradigms. If we are given an accurate performance model, we would be able to apply this tool to map any MPI application. We need to improve the mapping heuristics when we have a better performance model, thereby improving performance. We plan to model the resources in a hierarchical fashion instead of the flat fashion in which we are currently modeling since we think that the Grid will be hierarchical. Also we plan to apply our mapping techniques in context of another Grid based project, the MEAD (Modeling Environment and Atmospheric Discovery) project where there is a definite need for a mapper for scheduling a large number of WRF (Weather Research and Forecasting model) and ROMS (Regional Ocean Modeling System) simulations onto the Grid. This work is a good start toward the goal of having a component-based approach to construction of Grid Applications. We plan to drive this work to achieve that goal.

Bibliography

[AJMCY98] Vikram Adve, Guohua Jin, John Mellor-Crummey, and Qing Yi. High performance fortran compilation techniques for parallelizing scientific codes. In *Proceedings of SC98: High Performance Computing and Networking*, 1998.

[AORD99] V. Arnau, J M Orduna, A. Ruiz, and J Duato. On the characterization of interconnection networks with irregular topology. In *XI IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS'99)*, 1999.

[AS00] Vikram Adve and Rizos Sakellariou. Application representations for multiparadigm performance modeling of large-scale parallel scientific codes. *The International Journal of High Performance Computing Applications*, 14(4):304–316, Winter 2000.

[AS01] Vikram Adve and Rizos Sakellariou. Compiler synthesis of task graphs for parallel program performance prediction. *Lecture Notes in Computer Science*, 2017, 2001.

[BBS+] S. Barnard, R. Biswas, S. Saini, R. Van der Wijngaart, M. Yarrow, L. Zechtzer, I. Foster, and O. Larsson. Large-scale distributed computational fluid dynamics on the Information Power Grid using Globus. pages 60–67.

[CNNS94] Alok N. Choudhary, Bhagirath Narahari, David M. Nicol, and Rahul Simha. Optimal processor assignment for a class of pipelined computa-

tions. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):439–445, 1994.

[Dai02] Holly Dail. A modular framework for adaptive scheduling in grid application development environments. Technical Report CS2002-0698, Department of Computer Science, University of California at San Diego, 2002.

[DCB02] H. Dail, H. Casanova, and F. Berman. A modular scheduling approach for grid application development environments. Technical Report CS2002-0708, 2002.

[ea02] Ken Kennedy et al. Toward a framework for preparing and executing adaptive grid programs. In *International Parallel and Distributed Processing Symposium*, 2002.

[FK99] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kauffmann Publishers, Inc., 1999.

[KMG93] A. A. Khan, C. L. McCreary, and Y. Gong. A numerical comparative analysis of partitioning heuristics for scheduling tak graphs on multiprocessors. Technical Report CSE93-12, 21, 1993.

[KMP+] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and Dave Wonnacott. The omega calculator and library, version 1.00.

[KRC+99] Kathleen Knobe, James M. Rehg, Arun Chauhan, Rishiyur S. Nikhil, and Umakishore Ramachandran. Scheduling constrained dynamic applications on clusters. 1999.

[Kwo97] Y. Kwok. High-performance algorithms for compile-time scheduling of parallel - 84 - processors, 1997.

[MN99] K. Mehlhorn and S. Naher. *LEDA: A Platform of Combinatorial and Geometric Computing.* Cambridge University Press, Cambridge, England, 1999.

[OAD00] J M Orduna, V. Arnau, and J Duato. Characterization of communications between processes in message-passing applications. In *Proceedings of International Conference on Parallel Processing(ICCP'00)*, 2000.

[OD01] J M Orduna and J Duato. A new task mapping technique for communication-aware scheduling strategies. In *Proceedings of ICPP 2001 Workshop on Scheduling and Resource Management for Cluster Computing*, 2001.

[RKR+98] Jim Rehg, Kathleen Knobe, Umakishore Ramachandran, Rishiyur S. Nikhil, and Arun Chauhan. Integrated task and data parallel support for dynamic applications. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 167–180, 1998.

[Sar93] Vivek Sarkar. Parallel program graphs and their clasification. In *Languages and Compilers for Parallel Computing*, pages 633–655, 1993.

[TC00] Kenjiro Taura and Andrew A. Chien. A heuristic algorithm for mapping communicating tasks on heterogeneous resources. In *Heterogeneous Computing Workshop*, pages 102–115, 2000.

[Uni] Condor Team University. Condor version 6.1.12 manual.