1-1-2006

ATOP-grid for unified multidimensional adaptation of grid applications.

Garima Gupta
University of Windsor

ATOP-Grid for Unified Multidimensional Adaptation of Grid Applications

by

Garima Gupta

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada

2006

Your file Votre référence
ISBN: 978-0-494-35934-1
Our file Notre référence
ISBN: 978-0-494-35934-1

ABSTRACT

A computational grid is an ensemble of distributed, heterogeneous resources and provides various capabilities for efficient resource utilization. Grids are becoming prevalent platforms for high performance and resource intensive applications that require higher computing power or more resources than available on a single site. Grids are composed of heterogeneous resources, but not only are the resources heterogeneous, approaches for resource sharing are also heterogeneous. In addition some resources may become available or unavailable at some point of time. Thus, dynamic workload adaptation becomes a vital factor for an application's performance.

The approach presented here is an extension of the existing ATOP [27] approach for application internal dynamic workload adaptation. The goal is to maintain balanced progress across all nodes/sites executing an application. The presented approach provides:

- adaptation at different levels: globally across the sites, locally across the nodes of any local site and at node/CPU level for efficient resource utilization
- well-defined policies for local and global workload adaptation
- unification for scheduling with different resource sharing types and integration with the local job scheduler
- the option to trade between the time vs. the space dimension for flexible resource allocation. This is supported by introducing a new resource reservation type, *computational power*, in addition to the standard reservation type *number of nodes with or without reserved time share*

DEDICATION

To

My parents and Guruji, who guided me through the right path

And

My dear friend, Muqeeth for his endless support and encouragement

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to my advisor, Dr. A. C. Sodan, for giving me an opportunity to work in a very interesting area, and for her support, guidance and encouragement throughout my graduate studies.

I would also like to thank my committee members, Dr. Rankin, Dr. Wu and Dr. Tsin for their time and effort and their helpful comments and suggestions.

I am grateful to Mr. Mark Hahn from McMaster University for providing me the suitable technical environment to conduct my tests. I would also like to thank my colleagues, Lin Han, Yu Zou and Ahsanul Arefeen for helping me build the ATOP-Grid framework.

Finally, I thank several of my friends for helping with the preparation of this thesis and, my parents and my sister Deepika Gupta, for believing in me.

TABLE OF CONTENTS

ABSTRACT iii

DEDICATION iv

ACKNOWLEDGEMENTS v

LIST OF TABLES viii

LIST OF FIGURES ix

1. INTRODUCTION 1

2. RELATED WORK 3

3. A REVIEW OF THE STATE-OF-THE-ART 7

3.1 LOAD BALANCING PROBLEM 7
3.1.1 STATIC V/S DYNAMIC LOAD BALANCING 7
3.1.2 CENTRALIZED V/S DISTRIBUTED LOAD BALANCING 8
3.1.3 SYNCHRONOUS V/S ASYNCHRONOUS LOAD BALANCING 9
3.2 LOAD BALANCING ACROSS GRIDS 9
3.2.1 HIER – HIERARCHICAL PARTITIONING AND LOAD BALANCING 10

4. OUR APPROACH – ATOP-GRID 13

4.1 BASIC ATOP APPROACH 13
4.2 ZOLTAN LIBRARY 14
4.3 ATOP-GRID – EXTENSION OF ATOP FOR GRIDS 16

5. ADAPTATION FRAMEWORK 19

5.1 APPLICATION MODEL 19
5.2 RESERVATION 20
5.3 RESOURCE ALLOCATION 20
5.4 COMPONENTS OF ADAPTATION FRAMEWORK 22
5.5 METRICS 25
5.5.1 EXAMPLE CALCULATION 28
5.6 INTERACTION WITH LOCAL JOB SCHEDULER 32
5.7 POLICIES FOR LOCAL AND GLOBAL ADAPTATION 34
5.7.1 LOCAL ADAPTATION DECISION CRITERIA 34
5.7.2 GLOBAL ADAPTATION DECISION CRITERIA 37

6. IMPLEMENTATION 39

6.1 ADAPTATION CONTROLLER 40
6.2 DYNAMIC DIRECTORY 40
6.3 APPLICATION-INTERNAL GRID SCHEDULER 41
6.4 ADAPTATION LIBRARY 42
6.5 ADAPTATION COST MODEL 45
6.5.1 LOCAL ADAPTATION COST MODEL 45
6.5.2 GLOBAL ADAPTATION COST MODEL 48
6.5.3 ADAPTATION INTERVAL 49

7. TEST PLAN 50

7.1 TEST ENVIRONMENT 50
7.2 TEST APPLICATION 50

8. TEST CASES AND EXPERIMENTAL RESULTS 54

8.1 TEST CASES 54
8.2 EXPERIMENTAL RESULTS 57
8.2.1 TEST THE EFFICIENCY OF ATOP 57
8.2.2 FLEXIBLE ALLOCATION OF THREADS 62
8.2.3 LOCAL ADAPTATION – TIME VS. SPACE ADAPTATION 64
8.2.4 LOCAL ADAPTATION – BENEFITS OF TIME SHARING 68
8.2.5 LOCAL ADAPTATION – ADAPTATION TO DYNAMIC RESOURCE AVAILABILITY 69
8.2.6 GLOBAL ADAPTATION FOR FULLY MALLEABLE APPLICATION 72
8.2.7 GLOBAL ADAPTATION FOR CONSTRAINT MALLEABLE APPLICATION 73

9. CONCLUSIONS AND FUTURE WORK 75

REFERENCES 76

VITA AUCTORIS 81

LIST OF TABLES

TABLE 1 - ZOLTAN QUERY FUNCTIONS USED IN ATOP-GRID 15
TABLE 2 - ZOLTAN METHODS USED IN ATOP-GRID 16
TABLE 3 - METHODS PROVIDED BY ADAPTATION LIBRARY WITH THEIR DESCRIPTION [34]. 43
TABLE 4 - PROPERTIES OF GRAPHS 51
TABLE 5 - RUNTIMES FOR DIFFERENT THREAD ALLOCATION APPROACHES (CPU SHARING, NODE SHARING, AND SELF COSCHEDULING). AB-AB MEANS CPU HYPER SHARING, A-B NODE SHARING, AND AA-BB A COMBINATION OF NODE SHARING AND SELF COSCHEDULING. 63
TABLE 6 - RESOURCES USED (NUMBER OF NODES) UNDER TIME SHARING AND SPACE SHARING TO KEEP THE SAME PROGRESS/REMAINING RUNTIME. 67

LIST OF FIGURES

FIGURE 1 - POSSIBLE IMBALANCES AMONG SITES IN NORMALIZED WORKLOAD [4] 2
FIGURE 2 - ATOP- GRID – EXTENSION OF BASIC ATOP APPROACH [28] 17
FIGURE 3 - A) A TYPICAL SMP NODE WITH TWO HYPERTHREADED CPUS (CPU0 AND CPU1 REPRESENT PHYSICAL CPUS AND L0 AND L1 REPRESENTS LOGICAL CPUS B) CPU SHARING C) NODE SHARING D) SELF COSCHEDULING 22
FIGURE 4 - COMPONENTS OF ATOP FRAMEWORK AND INTERACTION BETWEEN THEM [4]. 22
FIGURE 5 - INTERACTION OF APPLICATION WITH JOB SCHEDULER THROUGH ADAPTATION CONTROLLER 33
FIGURE 6 - ADAPTATION CONTROLLER COMMUNICATES WITH APPLICATION INTERNAL GRID SCHEDULER AND LISTENER THREAD OF THE APPLICATION WHILE ACCESSING INFORMATION FROM DYNAMIC DIRECTORY [22] 42
FIGURE 7 - EXAMPLE OF AN APPLICATION USING THE ATOP ADAPTATION LIBRARY 45
FIGURE 8 - TEST APPLICATION COMMUNICATION PATTERN 51
FIGURE 9 - ABSTRACT ALGORITHM OF THE TEST APPLICATION 53
FIGURE 10 – INITIAL PARTITIONING TIME FOR PARTITIONING FROM SCRATCH (S) AND OVERPARTITIONING WITH 16, 64, 128 AND 256 PARTITIONS. 58
FIGURE 11 – ADAPTATION TIME FOR ADAPTATION STEP 64→32 (TOP) AND 32→40 (BOTTOM), SHOW FOR PARTITIONING FROM SCRATCH (S) AND OVERPARTITIONING (O) WITH 64, 128, AND 256 PARTITIONS 59
FIGURE 12 - ADAPTATION TIME FOR ADAPTATION STEP 40->16 SHOWN FOR PARTITIONING FROM SCRATCH (S) AND OVERPARTITIONING (O) WITH 64, 128, AND 256 PARTITIONS 60
FIGURE 13 – MAXIMUM EDGE CUTS PER NODE FOR ADAPTATION STEPS 64→32 (TOP) AND 32→40 (BOTTOM) FOR PARTITIONING FROM SCRATCH (S) AND OVERPARTITIONING WITH 64, 128 AND 256 PARTITIONS 61
FIGURE 14 – MAXIMUM EDGE CUTS PER NODE FOR ADAPTATION STEPS 40→16 FOR PARTITIONING FROM SCRATCH (S) AND OVERPARTITIONING WITH 64, 128 AND 256 PARTITIONS 62
FIGURE 15 - TEST FOR FLEXIBLE ALLOCATION OF THREADS 63

FIGURE 16 – RUNTIMES WITH AND WITHOUT ADAPTATION TO DYNAMIC CHANGE OF THREAD ALLOCATION APPROACH. 64
FIGURE 17 - RESOURCES USED (NUMBER OF NODES) UNDER SPACE SHARING AND TIME SHARING TO KEEP THE SAME PROGRESS/REMAINING RUNTIME. THE COSCHEDULED APPLICATION IS MENTIONED IN PARENTHESIS 65
FIGURE 18 - EXECUTION PROGRESS FOR WAVE, STARTING WITH DEDICATED ALLOCATION, COSCHEDULING FE_ROTOR FROM 50SEC TO 120SEC, AND COSCHEDULING FE_OCEAN AT 120SEC. IN THE LATTER CASE, RESULTS ARE SHOWN FOR BOTH CPU SHARING AND NODE SHARING [28] 66
FIGURE 19 – TEST CASE DEMONSTRATING THE BENEFITS OF TIME SHARING 68
FIGURE 20 - RUNTIMES WITH CPU/NODE SHARING VS. RUNTIMES FOR THE SAME APPLICATIONS UNDER DEDICATED RESOURCE ALLOCATION 69
FIGURE 21 – DYNAMIC RESOURCE AVAILABILITY IN TIME DIMENSION 70
FIGURE 22 - RUNTIMES WITH AND WITHOUT ADAPTING TO DYNAMIC RESOURCE AVAILABILITY IN THE TIME DIMENSION 70
FIGURE 23 – DYNAMIC RESOURCE AVAILABILITY IN SPACE DIMENSION 71
FIGURE 24 - RUNTIMES WITH AND WITHOUT ADAPTATION TO DYNAMIC RESOURCE AVAILABILITY IN THE SPACE DIMENSION: 32→50→32. 71
FIGURE 25 - RUNTIMES WITH AND WITHOUT GLOBAL APPLICATION ADAPTATION 73
FIGURE 26 – RUNTIMES OF BOTH SITES WITH AND WITHOUT RESOURCE ADAPTATION 74

1. Introduction

Grid computing is an emerging technology which allows sharing of computing resources on an unprecedented scale among several geographically distributed groups. Computational grids, [1] in particular, focus on sharing of resources for executing applications that require high computing power, thus permitting resource allocation to tasks that need more resources than available on a single site.

Applications executing in this environment can be structured as a group of individual tasks with coarse-grain communication, or parallel application with regular communication which need simultaneous reserved resource allocation on all participating sites. Resource reservation on each individual site is the responsibility of local job schedulers. Most importantly, cross-site jobs executing on distributed environments like grids have to deal with heterogeneity:

- Resources across the sites (and also within the sites) are heterogeneous with different number of nodes, relative CPU speed, memory, cache etc. and also network speed.
- Not only are the resources heterogeneous, but also according to [2], resource sharing on different sites can be heterogeneous. Resources can be shared in space or time dimension, or can employ Node/CPU sharing with/without time sharing (for details see Section 5.3).

Thus, heterogeneous hardware and network along with heterogeneous resource sharing types makes the correct workload distribution in this environment very difficult. In addition, resources may become available or unavailable during application execution. Hence, it is desirable that the application can adapt dynamically to ensure balanced

progress and to utilize the extra resources made available (potentially temporary). If workload adaptation is not dynamic, then the slowest node/site determines the overall runtime, degrading the performance of the whole application.

The approach presented here, ensures balanced computational progress of the application both globally across sites and locally across the nodes of a site, for optimal utilization of all available resources. This is explained in figure 1 which shows possible imbalances among sites in normalized workload. Dashed line shows balance target whereas solid lines show current progress per local compute node The Adaptive Time/Space sharing via Over Partitioning (ATOP) middleware is extended to work on grids as ATOP-Grid. This extension provides unified framework for scheduling and load balancing for different resource allocation types. ATOP-Grid integrates the local job scheduler with the application internal grid scheduler and also provides the option of trading between the time versus the space allocation for efficient resource utilization. This is achieved by supporting a new reservation type - *computational power*. Criteria or policies under which global or local redistribution will take place are clearly defined in the form of rules (conditions and consequences) using ideas from [3].

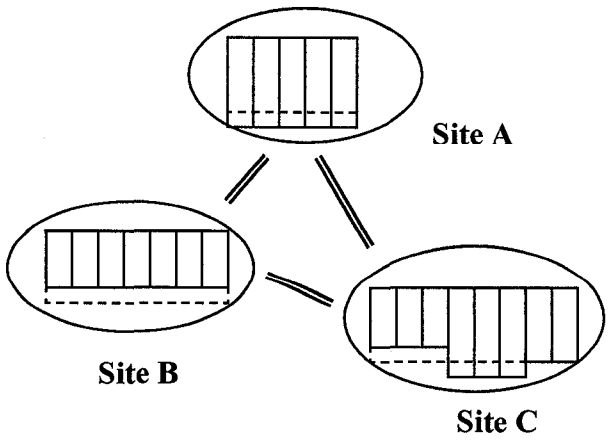


Figure 1 - Possible imbalances among sites in normalized workload [4]

2. Related work

Load balancing for parallel applications has been investigated over the last few years. While most of the load balancing algorithms focus on applications being executed locally (on a single site), there are a few approaches that explore dynamic heterogeneous environments like grids.

Zoltan [4] is a load balancing library that provides multiple load balancing approaches such as Recursive Coordinate Bisection (RCB), Recursive Inertial Bisection (RIB), Refinement tree based partitioning, ParMetis [6], Jostle and Octree partitioning The HIER [7] algorithm extends the Zoltan library to support hierarchical load balancing (for details see Section 3.2.1).

As mentioned in the previous section, different sites may employ different resource sharing approaches which include CPU/Node sharing. The CPU/Node sharing means that threads from different (or same) applications are scheduled on the same hyperthreaded CPU/Node, sharing the CPU/Node resources. Many local sites now use hyperthreaded CPUs. Hyperthreading [8] is Intel's definition for the concept of Simultaneous Multi Threading, in which a simultaneously multithreaded processor is split into two or more logical processors and threads are scheduled to execute on any of the logical processors.

The OPENMP scheduler described in [9] focuses on improving the performance of parallel applications executing on SMPs potentially using hyperthreaded CPUs. They propose a two level hierarchical scheduler that dynamically selects the appropriate scheduling strategy - within the nodes at the first level and the number of threads (one or

two threads per hyperthreaded node) at the second level. Scheduling strategy is selected for each parallel loop of the application. This scheduler proves to have marked benefit as compared to other runtime schedulers, but supports only parallel applications using OPENMP [10] and not MPI [11], which is a widely used parallel library. Secondly, this scheduler does not consider the effects of coscheduling (when application is scheduled to execute with another application on the same node/CPU). Similar work i.e. choosing the optimal number of threads to execute on the hyperthreaded CPU is done in [12], but this approach also does not take into account the effects of coscheduling.

Application internal scheduling was first introduced by AppLeS [13] (Application Level Scheduling). With this approach, each application executing on a grid is accompanied by a customized scheduling agent, tightly bound to the application. This agent gathers the information dynamically about the available resources and based on the user's performance criteria, schedules the application on these available resources. But since these agents are application customized, this limits reusability. AppLeS seek to address this problem by designing the reusable software modules targeting the Parameter Sweep Applications (PSAs) [14] and Master Worker model Applications (MWAs) [15].

Chen and Maheswaran [16] propose a dynamic scheduling algorithm for scheduling different applications over grids. In the proposed approach, scheduling is done in two phases – external scheduling (WAN wide) and internal scheduling (LAN wide). Proposed approach addresses issues like scalability, flexibility, and dynamic adaptability which are central to grid computing systems.

The SCOJO algorithm [17] proposes a cross-site scheduling approach, which takes the coscheduling effect into consideration, and can provide the start-time and share

reservations for applications executing on multiple sites. Effective shares use the slowdown factor to generate the actual share. The slowdown factor determines how much slower an application will execute when it is coscheduled with another application on the same set of nodes. This approach guarantees effective share reservation.

Taylor and Bryan [18] propose a hierarchical load balancing approach, which is divided into local and global load balancing phase, intended for the Structured Adaptive Mesh Refinement (SAMR) applications. This approach uses heuristic methods for calculating redistribution cost and associated performance gain and takes into consideration both the heterogeneity of processors as well as the networks into account including them in the cost estimation, though the focus is on the latter i.e. network heterogeneity.

Eager et al. [19] proposes an approach for adaptive data redistribution with respect to the dynamic resource availability of workstation clusters. The approach proposed is incremental data redistribution, similar to [20], and the results are compared to partitioning from scratch. The proposed scheme also mentions that allocation of data from the middle of data domain to the newly added node leads to a reduction in migration costs as compared to the data allocation from the edge.

Weinberg J et al. in [21] introduces a special form of space sharing, in which different parallel applications share the processors per node during their execution as compared to dedicated node allocation for a single application. This special form of space sharing, termed as the symbiotic space sharing is developed simultaneously and is similar to the node sharing discussed in this thesis work. Weinberg J. et al. outlines that since all jobs cannot share resources effectively, their approach proposes a scheduler that studies

different job characteristics and develops symbiotic schedules such that all the jobs scheduled are benefited with this special form of space sharing.

3. A Review of the State-of-the-Art

Most of the existing load balancing approaches focus on the traditional distributed systems and may not be directly applicable in the grid environments because of their inherent heterogeneity (machine, network and resource sharing). This chapter first reviews the traditional load balancing approaches and then we discuss the existing approaches to address the workload adaptation within computational grids.

3.1 Load Balancing Problem

The main goal of the load balancing is to distribute the workload evenly among all the processors (or all machines participating in application execution) to ensure optimal application performance and efficient resource utilization. Load balancing approaches can be classified into different categories, namely:

- Static v/s Dynamic load balancing
- Centralized v/s Distributed load balancing
- Synchronous v/s Asynchronous load balancing

We will discuss these categories in the following subsections

3.1.1 Static v/s Dynamic Load Balancing

Static load balancing algorithms distribute the workload between the processors based on the information regarding the task execution runtime and resource allocation at compile time. This means that the workload distribution occurs before the application starts executing and parallel jobs execute with the same distribution for the entire application runtime. This approach obviously is not suitable for the grid environments

where the resource availability is dynamic and [22] also mentions that this approach cannot be used for applications where the workload is generated dynamically at runtime.

Dynamic load balancing seeks to address these issues by reacting to the current system state while taking the load balancing decisions. This approach can redistribute the workload of application dynamically according to current available resources. But, the dynamic load balancing scheme needs to ensure that the overhead caused by dynamic redistribution should be significantly lesser than the performance gain achieved by the application. Kameda H. et al. in [23] compares the dynamic and static load balancing approaches and concludes that the dynamic approach performs better than the static ones in most of the cases. Dynamic load balancing can be centralized or distributed.

3.1.2 Centralized v/s Distributed Load Balancing

In the centralized load balancing approach, one process (called the master process) takes charge of checking the load imbalance by collecting the workload distribution information from all the processors, and redistributing the workload based on the cost benefit expectation. All processes synchronize during the load balancing phase and wait until the new workload is assigned to them.

In the distributed load balancing approach, each process has its local load balancing module, which at regular intervals will broadcast its current workload status to the other processors. In this way the work is transferred from a heavily loaded processor to a lightly loaded processor through work sharing or work stealing [22].

3.1.3 Synchronous v/s Asynchronous Load Balancing

The synchronous load balancing approach stops the execution of the application during the load balancing operation and resumes the application execution with the new workload (if redistribution during the load balancing step is invoked), whereas the asynchronous load balancing approach does not stop the application execution. Imbalanced (overloaded/underloaded) processors exchange information within themselves, or with the master processes (in case of centralized model) to redistribute their workload without affecting the execution of other processors.

The approach we propose here, ATOP-Grid, uses dynamic, centralized and synchronous load balancing scheme.

3.2 Load Balancing Across Grids

As mentioned earlier the traditional load balancing algorithms are not directly applicable to grids. Li and Lan [24] describes separate classification of the grid load balancing algorithms which are classified in three categories:

- Resource aware repartition based schemes
- Divisible load theory based schemes
- Prediction based schemes

Most scientific applications are represented by graphs where the vertices represent the computation and the edges represent the communication between the two vertices. Resource aware repartition based schemes balance the computational workload (number of vertices) across sites while trying to minimize the edge cut (potential communication volume). Algorithms with this scheme need to ensure that large volume of data should

not be migrated across sites, as the data movement cost across sites is very expensive due to high communication latency.

Divisible Load Theory (DLT) based schemes are suitable for the applications that communicate infrequently, and where the subtasks of the application do not depend on each other (wait for updated data from fellow processors). Thus the computation and the communication workload for this application can be divided arbitrarily.

Prediction based schemes capture the inherent dynamic nature of grids (in terms of resource availability). This scheme uses a performance evaluation model to accurately predict the future computation and communication cost of the application. This helps the load balancer to make more accurate partition decisions. Cactus [25] is a dynamic load balancing strategy that falls under this category. These prediction based schemes are generally accompanied with dynamic measurement feedback (of the current computational progress of each node/site), from the monitor for more accurate prediction of the future runtime. SCOPRO [26] is one such monitoring tool. This tool can extract the basic execution time of application (excluding the waiting times, where faster nodes wait for slower ones to finish) and can also provide information regarding the computation/communication ratio of the application.

There are certain load balancing strategies that fall into the combination of above mentioned categories. One such approach is the Hierarchical Load Balancing – HIER [7]. This approach is discussed in detail in the next subsection.

3.2.1 HIER – Hierarchical Partitioning and Load Balancing

This scheme extends the already existing load balancing library Zoltan by providing the hierarchical load balancing procedures. The different procedures are used at

the different levels of hierarchy in the computing environment. This is achieved by introducing an intermediate hierarchical balancing structure (IHBS) and providing their own set of callback functions. At each load balancing step (at each level of hierarchy), using these callback functions, partitioning is performed and IHBS is updated at each step of the partitioning. After partitioning is done, Zoltan migration arrays are created and returned to the application. With this approach only the lightweight structure - IHBS is migrated between various hierarchical levels, and not the whole application data, which potentially saves the data migration cost. This approach also exploits the fact that the different partitioning strategies, as provided by Zoltan, might be efficient at different levels of hierarchy and provides support for the same.

Teresco et al. [7] also describes Dynamic Resource Utilization model (DRUM). This model takes into consideration the heterogeneity of machines and the networks to assign the workload to different processors/machines taking part in application execution. DRUM represents the whole computing environment in the form of a tree, where the subenvironments are recursively divided. For example, in a single cluster, head node represents the whole cluster, which is further divided into SMPs (subenvironments) that are further divided into individual CPUs. Each leaf of the tree, which represents an individual computing entity, is attached with the load it is supposed to compute. DRUM assumes to have (initial) prior knowledge of the computing environment.

The HIER approach combined with the DRUM model might be useful for balancing load in grid environments. But, until now this approach has proven efficient only for local, heterogeneous SMP clusters. Load balancing steps involve repartitioning,

which might add up to significant partitioning cost across grids. Also the data structure - IHBS needs potential modification to work on grids.

4 Our Approach – ATOP-Grid

4.1 Basic ATOP Approach

ATOP (Adaptive Time/Space sharing via Over Partitioning) [27] provides two approaches for load balancing. They are:

- Overpartitioning
- Partitioning from scratch

Overpartitioning creates more data partitions than the processors that are allocated to the application. At the load balancing step, if any load imbalance is detected, partitions are migrated from heavily loaded processor to lightly loaded processor. For example if 128 data partitions are created i.e. 128 chunks of total computational workload and allocate them to 16 processors, then 8 partitions are allocated to each processor for computation (assuming all processors are homogenous). Later at the load balancing step, if some condition arises, like 8 processors becoming unavailable, then some partitions is migrated to each processor, such that now each processor has 16 partitions. This saves the repartitioning cost every time resource adaptation takes place. We later discuss that overpartitioning is a feasible approach for balancing workload across sites (see section 4.2). However, with this approach, there is a risk of increasing the edge cuts, which results in an increase of communication cost. Therefore, for cases where overpartitioning does not perform well ATOP proposes another approach.

Using partitioning from scratch approach, the number of partitions created is equal to the number of processors allocated for the execution of the application. At the time of resource adaptation, entire application data is repartitioned and these partitions are

migrated to the processors again. Thus the adaptation cost with this approach includes both the repartitioning and the migration cost.

These two approaches can perform resource adaptation in both the space and the time dimension. Both the approaches are implemented using the Zoltan library and use multilevel K-way partitioning approach provided by the well known ParMetis [6] library. In the next section a brief overview of the Zoltan library is provided and how certain features of this library are exploited to achieve the desired resource adaptation.

4.2 Zoltan Library

Zoltan is a collection of various tools that can be used by adaptive parallel and unstructured applications to improve their performance. Zoltan offers different load balancing and data partitioning algorithms, data migration tools, distributed data directories, and dynamic memory management tools, organized in a way that the application can choose from various utilities as needed. Apart from its own load balancing approaches, Zoltan also incorporates JOSTLE and ParMetis [6], a widely used parallel partitioning library. The greatest advantage of Zoltan is that it is a data structure neutral library. Thus, it allows the users to use their own data structures, by providing a set of callback functions also called as the query functions.

These call back functions are implemented by the user and query the application for the required information. These callback functions are registered in Zoltan by passing a pointer to the function and then Zoltan will call these functions, as and when any information from the application is required.

Query Functions	Explanation
ZOLTAN_NUM_OBJ_FN	Returns the number of objects that are currently assigned to the processor.
ZOLTAN_OBJ_LIST_FN	Returns object list currently assigned to the processor
ZOLTAN_PARTITION_FN	Returns list of partitions to which given objects are currently assigned.
ZOLTAN_NUM_EDGES_FN	Returns the number of edges in the communication graph of the application for each object in a list of objects.
ZOLTAN_EDGE_LIST_FN	Returns lists of global IDs, processor IDs, and optionally edge weights for objects sharing edges with objects specified in the *global_ids* input array.
ZOLTAN_OBJ_SIZE_FN	Returns the size of the buffer needed to pack a single object.
ZOLTAN_PACK_OBJ_FN	Information how to copy all needed data for a given object into a communication buffer.
ZOLTAN_UNPACK_OBJ_FN	Information how to copy all needed data for a given object from a communication buffer into the application's data structure.
ZOLTAN_PRE_MIGRATE_PP_FN	For performing any pre-processing desired by application.

Table 1 - Zoltan Query functions used in ATOP-Grid

(Source - http://www.cs.sandia.gov/Zoltan/Zoltan.html)

These callback functions are divided into two categories – the general query functions and the migration query functions. ATOP-Grid uses and implements query functions related to the partitioning and migration of data from Zoltan library. A brief description of the query functions used in ATOP approach is mentioned in Table 1.

Apart from these query functions, Zoltan also provides certain methods for initialization and finalization of the Zoltan interface, and for performing the actual partitioning and migration of data. Zoltan methods that are used in this thesis work are mentioned in Table 2 with their brief description

Zoltan Methods	Description
Zoltan_Initialize	Initializes MPI for Zoltan.
Zoltan_Create	Allocates memory for storage of information to be used by Zoltan and sets the default values for the information.
Zoltan_Set_Param	Modify the values of some parameters used in Zoltan. Parameters can be changed one at a time
Zoltan_Set_Fn	Registers an application-supplied query function in the Zoltan structure.
Zoltan_Invert_Lists	Computes inverse communication maps useful for migrating data, meaning if list of objects to be received by processor is known , then list of objects to be sent out is calculated and vice versa
Zoltan_LB_Free_Part	Frees the memory allocated by the Zoltan to return the results of Zoltan_LB_Partition or Zoltan_Invert_Lists.
Zoltan_Destroy	Frees the memory associated with a Zoltan structure
Zoltan_LB _Set_Part_Sizes	Specifies the desired relative partition sizes; equal by default.
Zoltan_LB_Partition	Invokes the real load-balancing routine that was specified using Zoltan_Set_Param function with the LB_METHOD parameter.
Zoltan_Migrate	Performs the migration for Zoltan; selects object lists to be sent to other processors, along with the destinations of these objects, and performs the operations necessary to send the data associated with those objects to their destinations.

Table 2 - Zoltan Methods used in ATOP-Grid

(Source - http://www.cs.sandia.gov/Zoltan/Zoltan.html)

4.3 ATOP-Grid – Extension of ATOP for Grids

The basic idea behind the presented approach is to ensure equal progress across all sites/nodes, participating in the application execution for efficient resource utilization.

Figure 2 illustrates how the basic ATOP approach is extended at the global level, for the workload adaptation among the participating sites, and at the Node/CPU level for the efficient resource utilization. Thus with the extended approach, ATOP-Grid, the workload is first distributed among all the participating sites, and workload on each site is subsequently allocated to each node of the site, based on the reservation negotiated with the local job scheduler. This extension is also applied at the Node and CPU level, where different threads bound to different nodes/CPUs share the computational workload, speeding up the overall execution of the application.

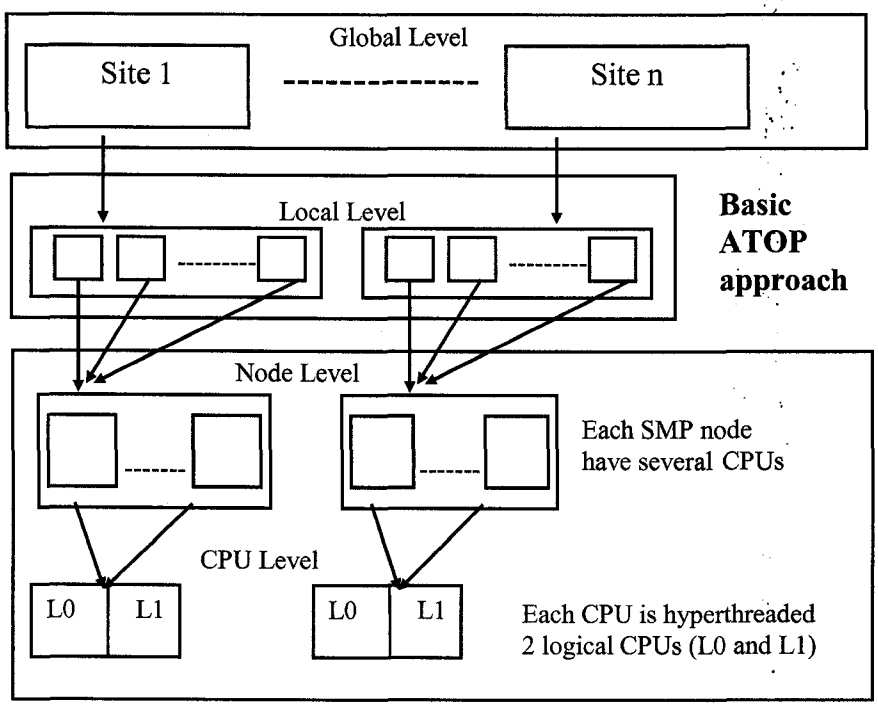


Figure 2 - ATOP- Grid – Extension of Basic ATOP approach [28]

In heterogeneous distributed environments like grids, the resource availability changes dynamically, which leads to workload imbalance both locally (within a site) and globally (across sites). The conditions that may lead to local or global imbalance is

clearly defined and well defined decision criteria (via rules) for balancing workload in both the cases, is provided..

Overpartitioning is chosen to perform global workload adaptation for various reasons. Firstly, Zoltan provides all parallel partitioning approaches, which might not be feasible to use due to the high communication latency across grids, and may result in a very high partitioning cost. Overpartitioning saves the repartitioning cost (as mentioned in section 4.1) and makes global redistribution relatively cheap. Secondly, Zoltan uses MPI for communication, which would need optimizations to work efficiently on grids.

5 Adaptation Framework

5.1 Application Model

Before describing the application model, the term malleability is introduced. The application is malleable if it can change the number of processes or adapt to various time shares during its execution. Application can be space malleable or time malleable. It is space malleable if it can change the number of processes, and time malleable if it can change to varying time shares on different processors during its runtime. The following assumptions about the application are made:

- Application is fully malleable (both in space and time dimension) for local adaptation which is supported by schedulers like [36].
- Globally, two different classes of the application malleability is introduced
    - The application is fully malleable (similar to local case)
    - The application is constraint malleable, if the global partitioning is restricted by module/cost hierarchies. Thus, this class of applications restricts the global redistribution to be done just once initially, and later any imbalance in the load has to be corrected locally. This assumption is applicable to most of the applications executing on grid environments.
- Data structures used by the application can be mapped on the graph structure where the vertices denote the computational workload and the edges represent the communication workload.
- The application is synchronous, permitting synchronous adaptation, and the global adaptation occurs less frequently than the local adaptation.

5.2 Reservation

It is assumed that the resources are reserved in advance [29] on each participating site by negotiating with the local job schedulers on each site. In the approach, ATOP-Grid, the following types of reservations are supported:

- *Number of nodes without time sharing* – this means dedicated resource allocation of particular number of nodes on the site for estimated runtime.
- *Number of nodes with effective time share* (*EShare*) – effective time share denotes the actual time share allocated to the application taking slowdown into account, which might occur if the application is coscheduled. Thus *EShare* = *Reserved time share/slowdown.*
- *Certain computational power for estimated runtime* – Computational power is expressed by certain estimated runtime on the dedicated set of nodes. This type of reservation gives us an option to trade between the time vs. space allocation. The time vs. space allocation means that we can change the resource allocation to less time shares by increasing the number of nodes and vice versa, maintaining the same runtime. This type of reservation provides flexibility to the local job scheduler for meeting reservations, but it also needs detailed cost formula or the rough complexity formula that can be used by the tools like SCOPRED [30] for the scalable cost estimation.

5.3 Resource allocation

Resource allocation can change dynamically in both the space and the time dimension. This dynamic allocation may be beneficial in the conditions when:

- More resources (with more processors or more time shares) become available, and the job scheduler advises that these resources can be used. This may arise in a case where the application of concern is coscheduled with another application, and that application terminates, freeing all the resources it was using earlier.
- One site is slower than the other and global adaptation cannot be performed (for details see section 5.1), then the faster site reduces the number of resources it was using to ensure equal progress on all the sites. Information about the released resources is transferred to the job scheduler, so that it can utilize these resources elsewhere.

In addition, locally the Node/CPU sharing on the hyperthreaded/SMP nodes is also supported. This approach is an extension of LOMARC [31] hyper coscheduling, under which we schedule (bind) threads of different applications on different CPUs of the same node (Node sharing), or schedule it on the same hyperthreaded CPU (CPU sharing). Optionally, an application can also coschedule two threads on the same hyperthreaded CPU (self coscheduling). The different thread scheduling methods are shown in Figure 3. A and B refers to two different applications in the figure.

Threads bind to the CPUs using the *set_schedaffinity()* function provided by Linux 2.6 API. Different sets of applications exhibit different coscheduling behavior in terms of slowdown as shown in [31]. This depends on how well they share all the shared resources like cache, execution units etc. Hence, the presented approach can dynamically switch between these thread scheduling approaches, until it finds an approach that is best for a particular set of applications.

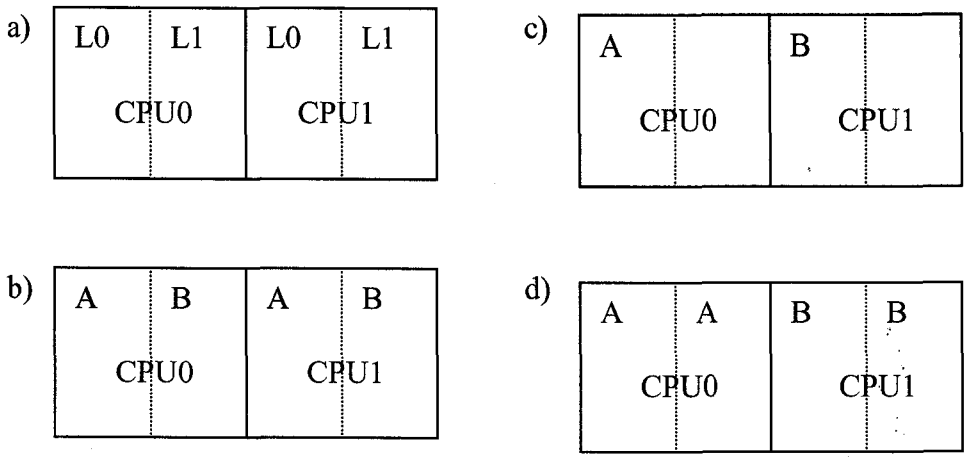


Figure 3 - a) A typical SMP node with two hyperthreaded CPUs (CPU0 and CPU1 represent physical CPUs and L0 and L1 represents logical CPUs b) CPU sharing c) Node sharing d) self coscheduling

5.4 Components of Adaptation Framework

Various components of ATOP-Grid and their interaction are shown in Figure 4. Here, one site is the master site that is responsible for taking global adaptation decisions, and also does the actual global workload redistribution. Figure 4 represents two sites where the box on the left represents the master site and box on the right represent another site participating in execution.

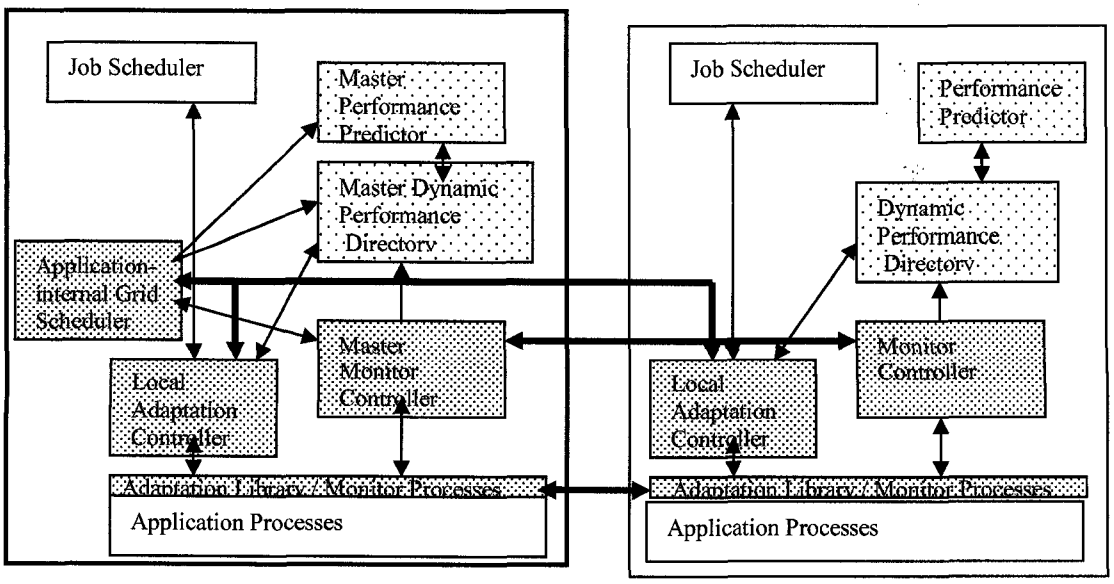


Figure 4 - Components of ATOP framework and interaction between them [4]

Locally, there are several components that interact with each other and also exchange information with the master site for potential global adaptation. In the figure, normal lines represent the local interaction whereas the heavy lines represent the cross-site communication. Now, the functionality of these components is discussed one by one:

- Monitor Controller – dynamically monitors application's progress on various nodes (locally), and detects if any imbalance exists. All the information collected by monitor is stored in the Dynamic Directory.
- Local Adaptation Controller – checks if any imbalance exists and decides whether adaptation is worthwhile or not (by taking the most recent information from the Dynamic Directory), and also reacts to the external adaptation requests given by the job scheduler.
- Dynamic Directory – stores information about all the running jobs on that site, by taking dynamic input from the monitor, and also provides this information to the performance predictor for correct runtime estimation. The idea of the Dynamic Directory is quite similar to the one described in [32].
- Performance Predictor – based on the detailed cost formula or a rough complexity formula [30] stored in the Dynamic Directory, it provides correct runtime estimation for the application. This information is passed to the Dynamic Directory, and used by the adaptation controller to take adaptation decisions.
- Adaptation Library – provides the functions to do the actual workload redistribution as advised by the adaptation controller. We use the same approach as described in the basic ATOP middleware (see section 4.1) for the workload redistribution.

- Master Monitor Controller – collects the current application progress information from all the local monitor controllers and detects if any global imbalance exists. It provides this information to the application-internal grid scheduler for global adaptation decisions.
- Application-internal Grid Scheduler – Each application executing on the grid has an Application-internal Grid Scheduler at the site of submission (master site). This component decides whether global redistribution should be done or not by interacting with the local adaptation controllers and the master monitor controller. The application-internal grid scheduler also calculates the global weights (relative workload) that should be assigned to each site.

For the initial workload distribution, the application-internal grid scheduler calculates the site weights and passes this information to the local adaptation controllers, and then they redistribute the workload locally by taking the resource allocation information from the local job scheduler. Local monitors asynchronously collect the progress information which can be used by the adaptation controllers to check for the imbalance. If an imbalance exists then adaptation controllers inform the application, but the application acts on this imbalance information only at the load balancing step, which occurs at regular intervals in the application. Checking for the global adaptation is less frequent as compared to checking for the local adaptation.

If the application notices that an imbalance exists, then it synchronizes all the processes for workload redistribution (global or local, depending on which check is performed), by taking the updated weight vectors from the local adaptation controllers.

Once the workload redistribution is complete, the application continues with the newly assigned workload.

5.5 Metrics

For the detection of imbalances and subsequent adaptation decisions, the adaptation framework uses several metrics. All these metrics are denoted in the form of vectors.

- Progress (*P*) - represents basic execution time of the application, includes the basic communication time but excludes the extra waiting times (times where faster sites wait for slower ones to complete their execution). This metric is a vector with one entry per node. Progress Vector (PVec) values can change during the execution of the application (and also determine potential imbalance).
- Application Relative Machine (*ARM*) factor - represents the reciprocal of execution speed of the application on a specific node, with reference to a base machine. This metric encapsulates the heterogeneity of machines, and also the application specific characteristics like instruction mixes or cache locality for the application data. Thus a larger factor represents slower execution, and smaller value means faster execution with reference to the base machine. ARM factor is also a vector (*ARMVec*) with one entry per node (of the local site).
- Slowdown (*SL*) factor - Slowdown comes into effect when the application is coscheduled and this captures how much slower the application will execute (due to resource sharing), than it will when it has dedicated resource allocation. For this metric we define a corresponding slowdown vector (*SLVec*) with one entry per node.

- Effective share (*EShare*) factor - As mentioned in section 5.2, this metric denotes how much slower the application will run beyond its reserved time share (*RShare*), if it is coscheduled with another application. This vector also has one entry per node and is calculated by $EShareVec_i = RShare_i/SLVec_i$ where $1 \leq i \leq N_{nodes}$

Each of the vectors mentioned till now have both estimated and measured values. Estimated values for each element of the progress vector are always equal, as the basic idea of this approach is to have equal progress. The estimated values for *ARMVec*, *SLVec* and *EShareVec* can be different as they capture heterogeneity, meaning machine factors or slowdowns on different nodes can be different.

Note that during the application execution, generally only one of the two vectors *SL* (slowdown) or *ARM* is relevant. This means that when the application is coscheduled, *SLVec* is relevant; when the application runs under dedicated resource allocation (or reserved time share), *ARMVec* is relevant. Though, both the vectors can exist at the same time if the application is coscheduled on a subset of nodes.

Using the metrics defined till now, the workload imbalance of the application is determined, both locally and globally. Some additional metrics for imbalance detection are described here.

- Progress Divergence Vector (*PDIVec*) - defined as the ratio of measured vs. estimated values in the progress vector.
- Site Progress Divergence Vector (*SPDIVec*) - defined as the average *PDIVec* value per site: $SPDIVec_{site,i} = \sum 1 \leq j \leq N_{sitei} PDIVec_{site,i,j} / N_{site,i}$ (with $N_{site,i}$ being the number of nodes at site$_i$). The average expresses the relative workload

imbalance between sites which cannot be corrected locally. The number of elements in this vector is equal to the number of participating sites.

- Local Imbalance (IMB_{local}): average of the values of the progress divergence vector vs. the largest value in the corresponding vector: $avgDIV_{local} = \sum i=0, N-1\ PDIVec_i / N$ and $IMB_{local} = max\{PDIVec_i \mid 0 \leq i < N\} / avgDIV_{local}$ with N being the number of local nodes. Thus, $IMB_{local} \geq 1$.
- Global Imbalance (IMB_{global}): average of the values of the site progress divergence vector vs. the largest value in the corresponding vector: $avgDIV_{global} = \sum i=0, Nsites-1\ PDIVec_i/N_{sites}$ and $IMB_{local} = max\ \{SPDIVec_i \mid 0 \leq i < N_{sites}\}/ avgDIV_{global}$ with N_{sites} being the number of local nodes. Thus, $IMB_{global} \geq 1$.

If larger time shares become available (condition where the job scheduler advises that more resources are available temporarily) and adaptation is not intended, then the progress vectors should be adjusted accordingly to ensure proper workload balance in the long run. Thus $PVec_{adjusted,i} = PVec_i * EshareDIVec_i$.

If a decision for the workload redistribution is taken (based on the decision criteria discussed in section 5.7), the workload to be allocated to each site and subsequently to each node needs to be recalculated. To represent this allocation ATOP weight vectors, both locally ($AWVec_{local}$) and globally ($AWVec_{global}$) are defined.

- Local ATOP weight vector ($AWVec_{local}$) - calculated by taking the ratio of measured Eshare vector to ARM vector. $AWVec_{local,i} = EShare_{mes,local,i} / ARM_{mes,i}$
- Global ATOP weight vector ($AWVec_{global}$) - calculated by summing up all the local ATOP weight vector values per site and multiplying it with appropriate communication fraction

$AWVec_{global,Sitei} = \sum_{j=0,Nsitei-1} AWVec_{local,I} * ((1\text{-}FracComm_{local,\ Sitei,\ Ni}) / (1\text{-}avgFracComm_{global}))$

Different machines induce different communication cost which is reflected in the ARM vector locally. But, globally, different sites may employ a different number of nodes (and there is a possibility that all nodes are not available for the application execution), and hence induce different communication cost, which should be included while calculating the global ATOP vectors. $AWVec_{local}$ has one entry per node and $AWVec_{global}$ has one entry per site.

Once these vectors are calculated the workload (W) is distributed accordingly, resulting in $W_{site,i} = W/N_{sites} * AWVec_{global,i} / avgAWVec_{global}$ work allocated to each site (for all N_{sites}) and $W_{node,j} = W_{site,i}/N_i * AWVec_{local,i} / avgAWVec_{local}$ work allocated to each node (for N nodes each local site). W is the total workload distributed among all sites and W_{site} is local workload distributed among the nodes within a local site.

Hence, $W = \sum W_{site,i}$ where $1 \leq i \leq N_{sites}$

And, $W_{site} = \sum W_i$ where $1 \leq i \leq N_{nodes}$.

5.5.1 Example Calculation

Here an example is presented to explain how the metrics and the vectors described in the previous section are used to correct the workload distribution.

The application is assumed to execute for 20000 iterations with total estimated runtime of 800 seconds. Local checks are performed after every 500 iterations whereas global checks are performed after every 2000 iterations. Moreover, the application executes on 2 sites, where each site has 8 nodes each reserved for application execution. Initially, machine vectors for both are assumed to be

ARMVecsite1 = {1, 1, 1, 1, 1, 1, 1, 1} and $ARMVec_{site2}$ = {1, 1, 1, 1, 1, 1, 1, 1}

and effective share vectors are assumed to be

$EShare_{site1}$ = {1, 1, 1, 1, 1, 1, 1, 1} and $Eshare_{site2}$ = {1, 1, 1, 1, 1, 1, 1, 1}

Hence for the initial step ATOP weight vectors are

$AWVec_{local, site1}$ = $EShare_{site1}$/ $ARMVec_{site1}$ = {1, 1, 1, 1, 1, 1, 1, 1}

$AWVec_{local, site2}$ = $EShare_{site2}$/ $ARMVec_{site2}$ = {1, 1, 1, 1, 1, 1, 1, 1}

And global ATOP weight vector is $AWVec_{global}$ = {8, 8}

As described in the previous section, calculation of global ATOP weight vector also includes communication fraction factor but this communication fraction factor is not applicable for the example calculation, since both the sites employ equal number of nodes. Moreover the test application used to test this framework employs nearest neighbor communication. If the application employs collective communication, then communication cost is known to increase at least logarithmically as the number of node increases, hence for those cases, the communication fraction described above should be taken into account when distributing the workload globally.

Initially, if we consider the total number of partitions to be 128, then it results in 128 * (8/16) = 64 partitions allocated to Site1 and 128 * (8/16) = 64 partitions allocated to Site2.

Similarly, using local ATOP weight vectors, partitions are allocated to all nodes on each site. Now at the first local check, application has completed the first 500 iterations (2.5% of its runtime), hence e.g. for Site1

$PVec_{est, site1}$ = {20, 20, 20, 20, 20, 20, 20, 20} (in seconds)

Let us assume that the feedback as received from the monitor is

$PVec_{mes,\ site1}$ = {21, 19, 19.7, 21.7, 22.2, 21, 30.9, 31.2}

Hence $PDIVec_{site1}$ = {1.05, .95, 1, 1.08, 1.1, 1.05, 1.54, 1.56}

This gives, $avgDIV_{local,\ site1}$ = 1.16 and $IMB_{local,\ site2}$ = 1.56 / 1.16 = 1.34 using the formulas described in the previous section.

Now assuming $\Delta_{app,local}$ (local imbalance tolerance range) = 0.1 and after checking cost effectiveness for the adaptation decision (for details see section 6.5.) local workload redistribution for Site1 is done.

Thus, for calculation of new ATOP weight vectors, we consider the measured values for ARMVec and EshareVec.

Hence, $ARMVec_{mes,\ site1}$ = {1, 1, 1, 1, 1, 1, 1.5, 1.5}

and $EShareVec_{mes,\ site1}$ = {1, 1, 1, 1, 1, 1, 1, 1}

Thus $AWVec_{local,\ site1}$ = $EShare_{mes,\ site1}$/ $ARMVec_{mes,\ site1}$ = {1, 1, 1, 1, 1, 1, 0.66, 0.66}

and the new partition allocation to various nodes on Site1 is {9, 9, 9, 9, 8, 8, 6, 6}

Similar calculations are performed on Site2. But on Site 2 we assume that the application is coscheduled with another application on first 4 nodes. Thus SLVec is relevant in this case for the imbalance detection and for calculating the new workload distribution. Now for Site 2

$PVec_{est,\ site2}$ = {20, 20, 20, 20, 20, 20, 20, 20} (in seconds)

It is assumed that the feedback as received from the monitor is

$PVec_{mes,\ site2}$ = {33.43, 35.40, 32.05, 34.02, 21, 19.04, 20.05, 22.24}

Hence $PDIVec_{site2}$ = {1.67, 1.77, 1.60, 1.70, 1.1, 0.95, 1.01, 1.11}

This gives, $avgDIV_{local,\ site2}$ = 1.36 and $IMB_{local,site2}$= 1.77 / 1.36 = 1.301

Now assuming $\Delta_{app,local}$ (local imbalance tolerance range) = 0.1 and after checking cost effectiveness for adaptation decision (for details see section 6.5) local workload redistribution for Site 2 is done.

Thus, for calculation of new ATOP weight vectors, we consider the measured values for ARMVec, EshareVec and SLVec.

Hence, $ARMVec_{mes,\ site2} = \{1, 1, 1, 1, 1, 1, 1, 1\}$,

$SLVec_{mes,site2} = \{1.6, 1.6, 1.6, 1.6, 1, 1, 1, 1\}$

And we calculate the measured effective share vector which we get by taking a ratio of the reserved time share on each node to the slowdown vector. Since we assume that all nodes are reserved with dedicated resource allocation, hence $EShareVec_{mes,\ site2} = \{0.625, 0.625, 0.625, 0.625, 1, 1, 1, 1\}$

Thus $AWVec_{local,\ site2} = EShare_{mes,\ site2}/\ ARMVec_{mes,\ site2} = \{0.625, 0.625, 0.625, 0.625, 1, 1, 1, 1\}$

and the new partition allocation to various nodes on Site 2 is {6, 6, 6, 6, 9, 9, 9, 9}

Now, for the global adaptation check, the average divergence vector value from each site is collected at the master site to form site progress divergence vector. Average divergence values are calculated locally.

We assume at the global checkpoint, $SPVec_{mes} = \{1.16, 1.57\}$, where 1.16 is $avgDIV_{local,site1}$ as described above and 1.125 is $avgDIV_{local,site2}$.

Hence avgSPVec = (1.16+1.57)/2 = 1.365 and $IMB_{global} = 1.151$

Assuming $\Delta_{app,global}$ (global imbalance tolerance range) = 0.1 and after checking cost effectiveness for global adaptation (for details see section 6.5) global redistribution is taken.

Now if at the global checkpoint value of local ATOP weight vectors are

$AWVec_{local,site1} = \{1, 1, 1, 1, 1, 1, 0.66, 0.66\}$ for Site 1

$AWVec_{local,site2} = \{0.625, 0.625, 0.625, 0.625, 1, 1, 1, 1\}$ for Site 2

then $AWVec_{global} = \{7.32, 6.5\}$, and subsequently workload allocated to each site in terms of the number of partitions is 68 partitions to site1 and 60 for site2. Thereafter these partitions are again allocated to all the nodes per site using the local ATOP weight vectors, $AWVec_{local,site1}$ and $AWVec_{local,site2}$.

5.6 Interaction with local job scheduler

The application interacts with the local job scheduler (see Figure 5) through the adaptation controller. Various conditions which will lead to this interaction are as follows:

- A reserved share guaranteed by the local job scheduler at the time of execution of the application is not met. In this case the application asks the job scheduler to rectify this situation by allocating more shares to the application, until reservation is met. The job scheduler might also preempt some local non-reserved, coscheduled jobs in order to meet the reservation limit for application of concern.
- In order to maintain balanced progress of the job, some faster sites may also voluntarily release the resources to adjust to the progress of the slowest site. If a site decides to do so, it will interact with the job scheduler and provide information of all the released resources so that these resources can be used elsewhere (e.g. to execute any local jobs).

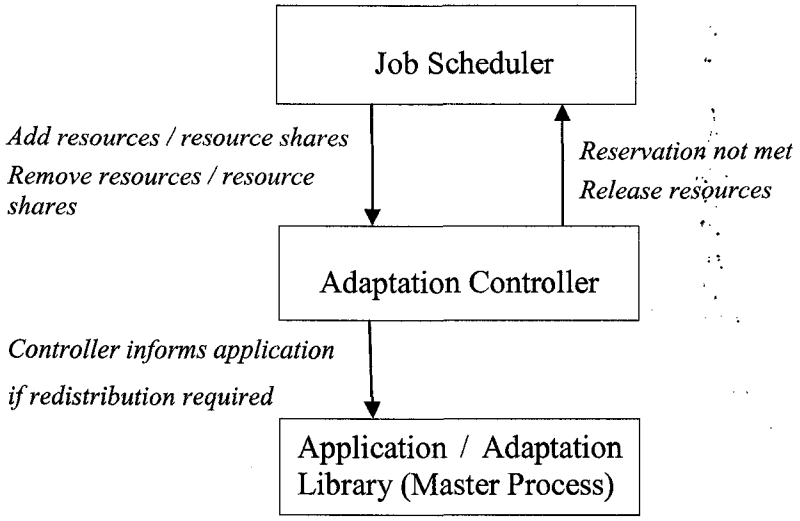


Figure 5 - Interaction of application with job scheduler through adaptation controller

- More resources (in terms of more time shares or more nodes) are available, and the job scheduler informs the application that it can utilize these resources temporarily. Based on the cost effectiveness, the application may or may not decide to adapt to extra available resources. Optionally, the job scheduler, or some agents attached to it can check if application can adapt (it is malleable), and calculate the minimum time duration for which resources should be made available to the application to make the adaptation worthwhile. This requires that the cost benefit formula used by the application (or something similar to that) is made available to the job scheduler (agent), via the adaptation controllers. Since the job scheduler in the presented framework is simulated, such functionality on the part of the job scheduler is not provided, but the application checks (the cost benefit) before taking an adaptation decision.

5.7 Policies for Local and Global Adaptation

There are certain conditions that lead to the local or global workload redistribution. Global redistribution is always accompanied by the local workload redistribution. In addition, if any one site performs local adaptation, other sites may have to wait, since our framework here considers synchronous applications. Thus, every local adaptation does have a global impact. In Section 7.5, we address this issue and derive the adaptation intervals to amortize the cost of these local adaptations.

In the following subsections the criterion for the local and global adaptation are discussed in detail.

5.7.1 Local Adaptation Decision Criteria

The following conditions lead to local adaptation:

- *Global Adaptation* → *Local Adaptation*

  As mentioned earlier, every time global adaptation occurs, the new workload is assigned to each site and this newly assigned workload has to be redistributed between the participating nodes of that site.

- $(IMB_{global} \geq \Delta_{App,global})$ && *NOT* $Cost\_Effective_{global})$ → *Local Adaptation*

  This means that if a global imbalance exists (global imbalance factor is more than delta tolerance range of application) and it is not worthwhile to do the global redistribution then we try to balance the workload locally to ensure overall equal progress. This can be done by allocating more resources to the slower sites (if potentially more resources are made available by the local job scheduler) or reducing the number of resources on the faster sites. Thus the local adaptation

controller needs this information, which is typically the maximum value of the Site Divergence Vector (*SPDIVec*). The Local Adaptation Controller retrieves this information from the Application-internal Grid Scheduler, and compares the local divergence to the globally worst progress. Based on this information it decides about the number of resources to be added or reduced.

Adapting in the time vs. the space dimension also depends upon the type of reservation that a site has. Section 5.2 describes the types of reservations the framework supports. Since for reservation type *number of nodes with/without time shares*, only the computing nodes as a whole can be added or removed, hence adaptation is done in space dimension for proper resource allocation. For reservation type *computation power*, there is an option to trade in the time vs. the space dimension depending upon cost benefit. The time vs. the space adaptation also provides flexibility for resource allocation to the local job scheduler. Thus this type of adaptation does not lead to any quantitative benefit, but the advantage is more qualitative in terms of more flexibility for resource allocation.

- *Local Adaptation && Resource reservation type = Number of nodes with/without reserved time shares → Space Adaptation*
- *Local Adaptation && Resource reservation type = Computational power → Space adaptation || Time adaptation*

Locally, the framework also supports multiple threads per node (see Section 5.3). Though the effects of changing the number of threads and thread allocation dynamically should be included in the cost benefit formula, still some general rules (for number of threads and thread allocation) are defined which will guide

the adaptation controller on how to adapt the number of threads and their allocation dynamically.

- *Application NOT coscheduled → number of threads = number of CPUs in SMP && thread_allocation = one thread per CPU within same SMP node*
- *Application is coscheduled && max* $\{SLVec_{mes,\,i} \mid 0 \leq i < N\} > threshold \rightarrow$ *number of threads = 1 && thread_allocation = one thread per CPU per application*

The above rules suggest that, if the application is not coscheduled with another application, and runs under dedicated resource allocation (or dedicated reserved time shares), the application can speed up its computation since each thread has dedicated CPU resources, while sharing only memory. Thus, by dividing workload in equal parts (equal to number of CPUs in an SMP node) and allocating equal share of work for computation to each thread, we expect a benefit in terms of the computational progress per node, apart from the synchronization overhead of having multiple threads.

On the other hand, if the applications (A and B) are coscheduled, then initially the application executes with thread allocation AB-AB (CPU sharing). At the time of checking for adaptation, if the adaptation controller realizes that applications do not coschedule well (by taking slowdown feedback from monitor), then the number of threads per application reduces to 1 and the thread allocation is changed to A-B (node sharing). The adaptation framework also provides the option of self coscheduling, i.e. AA-AA. But, in general, threads from the same applications may require similar type of resources, and this might lead to resource conflict, thus degrading the performance rather than benefit

achieved by distributing workload among multiple threads. Still this option is provided, since there are certain applications, e.g. integer intensive applications, which self coschedule well [31].

Locally, the framework offers an option to choose between two workload distribution approaches – overpartitioning and partitioning from scratch. To dynamically choose between the two approaches, we need a prediction model that can predict the future runtime and adaptation cost using the two approaches, taking adaptive resource allocation into account. In addition, adaptation frequency is also a major factor for selecting which approach to use between the two. According to [27] overpartitioning provides a benefit when adaptation is frequent and communication cost is low.

5.7.2 Global adaptation decision criteria

The following conditions lead to Global adaptation:

- *Communication (network) slowdown between sites exists → global adaptation*

  This means that the global imbalance is not due to computational imbalance among sites, but it might be due to decrease in communication speed (in terms of network speed) between some particular sites. The changes in network speed can be dynamically detected with tools like Network Weather Service (NWS) [33]. Then, the framework globally redistributes the workload such that less workload (and thus less communication overhead) is allocated to affected sites.

  This also includes a condition wherein communication to a site becomes very slow and expensive, to an extent that we will have to withdraw from that site, and globally redistribute the workload among the rest of the sites.

- *Global imbalance exists && application characteristics = fully malleable && $Cost\_Effective_{global}$ → global adaptation*

  The cost effectiveness calculation in the above mentioned rule also includes the remaining application runtime of the application. As mentioned in Section 5.7.1, global adaptation is not worthwhile, if the remaining application runtime is very low, which forms a case for potential local adaptation.

  In the case when global adaptation is not considered worthwhile by the application-internal grid scheduler, it sends the maximum value of site divergence vector (*SPDIVec*) to all the local adaptation controllers (including the one on the same site) for further adaptation decisions. A similar condition occurs when application is constraint malleable, because (as mentioned in Section 5.1) flexible reallocation is not possible in this case.

  If the application-internal grid scheduler decides to adapt globally, it calculates the new global ATOP weight vector ($AWVec_{global}$), and sends the updated vector values to the local adaptation controllers of all sites (including its own site). This also leads to local workload redistribution as mentioned earlier.

6. Implementation

This section describes the implementation details of our adaptation framework with specific details of different components.

As described earlier, our adaptation framework also supports adaptation at Node/CPU level. This is done by creating extra threads using the Pthreads library. These threads are bound to specific logical/physical CPUs using *setschedaffinity()* function supported by Linux API from kernel version 2.6.6 onwards. Since MPI is not a thread safe library, these extra threads created, are synchronized using a function *thread_barrier()*. This function block the threads until all the threads have completed their execution till the function calling point. If more than one thread per application is active during the application's execution, then the computational workload is divided among all these threads. For e.g. if 8 partitions are allocated to a node and, two threads are active, then partitions numbers 0, 2, 4 and 6 are allocated to the first thread and partitions numbers 1, 3, 5 and 7 are allocated to the second thread for computation. These threads remain active until the application terminates.

The next subsections provide implementation details of the specific components of our adaptation framework. Job scheduler, monitor controller and performance predictor in our framework are currently simulated. There exists detailed implementation of these tools [26] [30] but they are not yet integrated to work together due to platform dependability issues.

6.1 Adaptation Controller

In our implementation we have one adaptation controller per site. This adaptation controller communicates with the application through socket communication. The controller is connected to the application with a listener thread from the master process. The adaptation controller implements two important methods:

- *check_for_imbalance()* – This method takes the measured and estimated progress vectors (*PVec*) as input and checks if imbalance exists by subsequently calculating divergence vectors and imbalance using formulas described in Section 5.5. This function returns true if imbalance exists and false otherwise.
- *calculate_new_weights()* – If local imbalance exists and it is cost effective to carry on local redistribution, then the adaptation controller calls this method to calculate the new weight vectors for balanced workload redistribution. This method takes the measured *EShare* and *ARMVec* vectors as input and calculates the new local ATOP weight vectors ($AWVec_{local}$). Then the adaptation controller transfers this new weight information to the application by communicating with the listener thread of the application.

6.2 Dynamic Directory

The basic concept of dynamic directory is similar to [17]. Here, dynamic directory is implemented as a multithread socket server with one dynamic directory per site (implementation concept similar to A. Arefeen [22]). It keeps information about all the vectors corresponding to all active applications executing on that particular site. This information is stored in a two dimensional array with the main index representing the

application identification number (ID) of each application. For example *PVec_mes[2][4]*, means that *PVec_mes[0][4]* is the measured progress vector for the application with ID 0 and *PVec_mes[1][4]* is the measured progress vector for the application with ID 1. The application can communicate with the dynamic directory for any relevant information that is stored in the dynamic directory. All threads are synchronized using *pthread_mutex*. Each thread waits until it receives a request for information from the adaptation controller or receives an application's termination information.

6.3 Application-internal Grid Scheduler

The application internal grid scheduler is implemented as a multithreaded socket server where each thread is connected to the local adaptation controller on each site. This component receives the individual site progress divergence values from each site and forms a site progress divergence vector. This component also receives the local ATOP weight-vector information from each site and stores the sum of these vector values into a temporary ATOP global weight vector. This component also implements two important methods:

- *check_global_imbalance()* – Takes the site-progress divergence vector as input and calculates global imbalance using formulas described in Section 7.5. This method returns true "if global imbalance exists" or false otherwise.
- *calculate_global_weights()* – If global imbalance exists and it is cost effective to do global redistribution, then this method is used to calculate the new global ATOP weight vector which reflects the new workload distribution globally. This method takes the temporary ATOP global weight vector as input and calculates the actual ATOP weight vectors.

Figure 6 depicts how the various components interact with each other in the actual implementation.

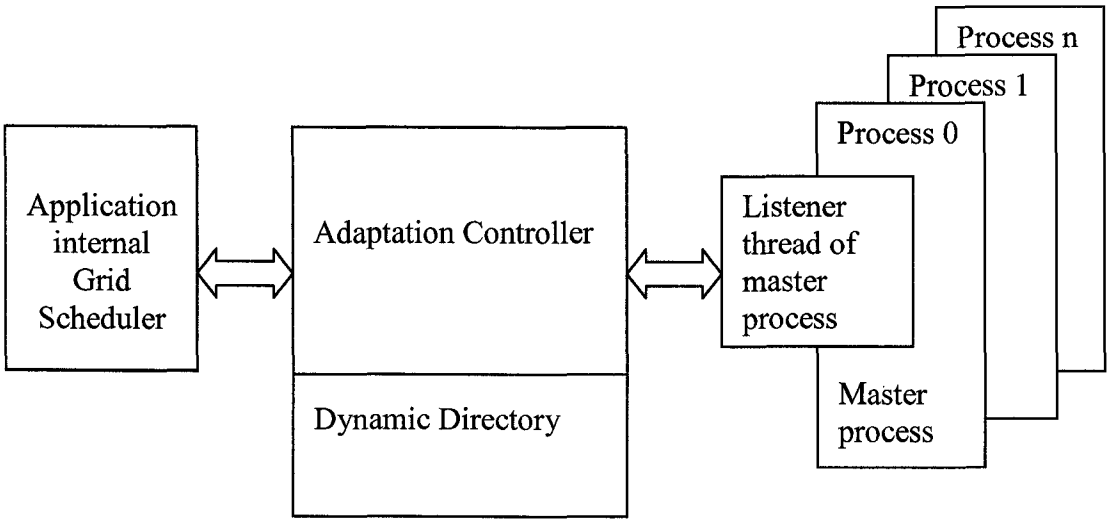


Figure 6 - Adaptation controller communicates with application internal grid scheduler and listener thread of the application while accessing information from dynamic directory [22].

6.4 Adaptation Library

The adaptation library provides various methods for local and global workload redistribution. The algorithm used for partitioning and migration of data is similar to the basic ATOP approach. All the methods described in Table 3 can be used by any parallel application which employs the MPI communication library.

Methods	Description
ATOP_ENVIRON_INIT	Initializes the MPI and ZOLTAN environment, and creates a listener thread at the master processor, which communicates with the adaptation controller.
ATOP_CREATE_MESH	This method initializes all the data structures used by the mesh (like nodes, partitions, border structure etc.) and maps them to the desired graph structure.
ATOP_ADAPT_INIT	This method initially distributes the workload among the sites and then among all local nodes by taking the initial ATOP weight vector from adaptation controller
ATOP_GET_NODE	Initializes node data on every node (every vertex of the graph)
ATOP_MESH_COMPUTE	This function maps the application to mesh data structures, and performs computation and communication according to workload distribution on various nodes.
ATOP_LOCAL_TRY_ADAPT	This function checks whether to adapt or not locally, based on feedback from adaptation controller and calls the actual redistribution routine (ATOP_ADAPT) if adaptation is advised.
ATOP_GLOBAL_TRY_ADAPT	This function checks for need of global adaptation according to feedback from application internal grid scheduler through local adaptation controllers.
ATOP_MESH_DESTROY	This method frees all the memory used by mesh data structures
ATOP_ENVIRON_DESTROY	This function shuts down the MPI and Zoltan environment and waits until listener thread (at the master process responsible for communicating with adaptation controller) terminates

Table 3 - Methods provided by adaptation library with their description [34].

Figure 7 shows an example of a parallel application using the ATOP adaptation library. The function ATOP_ENVIRON_INIT initializes the MPI and Zoltan environment and ATOP_CREATE_MESH creates all the mesh data structures needed by the application. Next we call the method ATOP_GET_NODE to initialize data on the vertices of the graph. Then the function ATOP_ADAPT_INIT is used to distribute the data initially to all sites and nodes participating in application execution. Then application starts executing and iterates over maximum number of iterations.

ATOP_MESH_COMPUTE does the actual computation and communication for the application and does the computation using the callback function defined by the user. In between the application's execution we check for the need of global and local adaptations. Checking for global adaptation is less frequent as compared to local adaptation, since global redistribution is more expensive than the local one due to high communication latency across sites. If the global or local adaptation is required all the processes of the application wait until they are allocated new workload and then resume the execution. Finally when the application execution is complete then method ATOP_MESH_DESTROY is called to free memory used by all data structures and finally the method ATOP_ENVIRON_DESTROY is called to shutdown the whole environment.

```
ATOP_ENVIRON_INIT (); //initializes the environment

ATOP_CREATE_MESH (); //create all mesh data structures used by application

ATOP_GET_NODE (); //initializes node data on every node

ATOP_ADAPT_INIT (); // Initial workload redistribution

For (i=0; i<max_num_of_iterations; i++)
 {
     ATOP_MESH_COMPUTE (); /* this method does the actual computation and
                              communication for the application, by using the
                              callback functions defined the user. */

      if (i mod larger_limit == 0 )      // global checks are done rather infrequently
      {
         GLOBAL_TRY_ADAPT ();
      }

      if (i mod smaller_limit == 0)     //local checks are more frequent than global ones
      {
         LOCAL_TRY_ADAPT ();
      }

ATOP_MESH_DESTROY ();

ATOP_ENVIRON_DESTROY ();

/* In the application larger limit and smaller limit are relative. For example, if application executes for ten
thousand iterations (max_num_of_iterations) then global check may be performed after every 2000
(larger_limit) iterations, while local check can be performed after every 500 (smaller_limit) iterations. */
```

Figure 7 - Example of an application using the ATOP adaptation library

6.5 Adaptation Cost Model

To check the cost effectiveness of local or global adaptation, we compare the cost of adaptation with the gain achieved by doing load redistribution in terms of reduction in application runtime. Thus *Cost_Effective* returns true if

$Gain * (1 - \Delta_{approx}) \geq Adaptation\_cost$

6.5.1 Local Adaptation Cost model

The adaptation cost $T_{adapt,local}$ can be broken down into

$T_{adapt,\ local} = T_{par,local} + T_{mig,local}$

where $T_{par,local}$ is the partitioning time *and* $T_{mig,local}$ is the migration time.

With overpartitioning, the partitioning cost ($T_{par,\ local}$) is negligible i.e. ~ 0. To derive the migration cost ($T_{mig,local}$), we study the overpartitioning algorithm. The algorithm follows the structure based order keeping partitions with successive numbers on the same or neighboring nodes. The main idea behind this partition-allocation strategy is that keeping neighboring partitions together results in lower edge cuts and hence higher distribution quality. In [27] Sodan A. et al. compared the structure-oriented order for partition allocation with the migration-oriented order and concluded that the migration-oriented order lowers the migration time, but with a significant increase of edge cuts, whereas the structure-oriented order results in lower edge cuts. Thus for the present work, we use the structure-oriented order to implement the overpartitioning algorithm. Hence, we can derive the maximum number of partitions $N_{part\_migr,\ max,local}$ to be migrated from current to target resources.

$$N_{part\_migr,\ max,\ local} = max\ (N_{par,\ curr} + N_{par,\ target})$$

$$= N_{par\_total,\ site} * max\ ((AWVec_{local,\ i,\ curr} / avgAWVec_{local,\ curr}) + (AWVec_{local,\ i,\ target} / avgAWVec_{local,\ target}))$$

with $N_{part\_total,site}$ being the total number of partitions on a local site, $N_{par,curr}$ is the number of partitions per node with the current workload distribution and $N_{par,target}$ is the number of partitions per node with the target workload distribution.

This gives a maximum communication volume

$C_{vol,\ max,\ local}=N_{part\_migr,\ max}*N_{vertices,\ part}*Size_{vertex}$ per node with the number of vertices per partition being $N_{vertices,\ part} = N_{vertices} / N_{part\_total,site} * (1 + IMB_{partition})$ where $Size_{vertex}$ is the

amount of data (in bytes) per vertex to be transferred and $IMB_{partition}$ the slight imbalance (due to the partitioner) in the number of vertices per partition.

Migration cost is determined by $T_{mig,\ local} = T_{transfer,\ local} * C_{vol,\ max,local}$ with $T_{transfer,local}$, the transfer time per byte on a local site. Hence $T_{adapt,local} = T_{mig,local} = T_{transfer,local} * C_{vol,max,local}$

Derivation of the adaptation cost mentioned above considers the worst case, meaning that migration cost is calculated with the assumption that all the partitions (maximum number of partitions) on a node will be migrated. Thus, the cost above is a general criterion for both time and space adaptation. This cost can be refined, if we are considering space adaptation for homogenous nodes. In this case, the maximum number of partitions migrated to change the allocation of number of nodes from $N_{nodes,curr}$ to $N_{nodes,target}$ is

$N_{part\_migr,\ max,\ local} = \lceil N_{part\_total,\ site} / max\ \{N_{nodes,\ curr}, N_{nodes,\ target}\} \rceil$

The maximum number of messages needed for receiving and sending data per node, then is given by $N_{msg,\ max,\ local} = \lceil max\ \{N_{nodes,curr}, N_{nodes,target}\} / min\{N_{nodes,curr},\ N_{nodes,target}\} \rceil + 1$. This gives a maximum communication volume

$C_{vol,max} = N_{msg,max,local} * N_{part\_migr,max,local} * N_{vertices,part} * Size_{vertex}$ per node with

$N_{vertices,part} = N_{vertices} / N_{part\_total,site} * (1 + IMB_{partition})$. Adaptation cost is then determined by $T_{adapt,local} = T_{mig,local} = T_{startup,local} * N_{msg,,max,local} + T_{transfer} * C_{vol,max,local}$

The gain is measured in terms of the benefit of reduced runtime which we expect after taking the adaptation decision, thus we measure the gain in terms of the difference between remaining runtime with and without adaptation. To calculate the gain we need the local imbalance factor, which is

$T_{gain,\ local} = T_{remaining,\ noadapt} - T_{remaining,\ adapt}$

$= IMB_{local} * (T_{total} - T_{exec,i} / IMB_{local} - T_{exec,prev}) — (T_{total} - T_{exec,i} / IMB_{local} - T_{exec,prev})$

$= (IMB_{local} - 1) * (T_{total} - T_{exec,\ i} / IMB_{local} - T_{exec,\ prev})$

The function $Cost\_Effective_{local}$ (using an error factor Δ_{approx}) delivers true if $T_{gain} * (1 - \Delta_{approx}) \geq T_{adapt,\ local}$

T_{total} being the total estimated runtime, $T_{exec,i}$ the application's runtime since the last adaptation, $T_{exec,prev}$ the runtime consumed before that last adaptation and IMB_{local} is the local imbalance factor as described in Section 5.5.

6.5.2 Global Adaptation Cost Model

For global adaptation, we also use the overpartitioning approach. Rationale for the same is explained in Section 4.3. Thus

$T_{adapt,global} = T_{par,global} + T_{mig,global}$

With overpartitioning $T_{par,\ global} \sim 0$, hence $T_{adapt,global} = T_{mig,global}$

The global adaptation cost model is quite similar to the local one; except that we use the global communication parameters for startup and data transfer time and calculate the number of partitions to be migrated with respect to total number of sites.

Thus, to calculate the global migration cost we calculate the maximum number of partitions migrated per site, which is,

$N_{part\_migr,\ max,\ global} = \lceil N_{part\_total} / N_{sites} \rceil$

where N_{part_total} is the total number of partitions and N_{sites} is the total number of sites.

The maximum number of messages needed for receiving and sending data per site, then is $N_{msg,\ max,\ global} = N_{sites}$. This gives a maximum communication volume globally

$C_{vol,max,global} = N_{msg,,max,global} * N_{part\_migr,max,global} * N_{vertices,part} * Size_{vertex}$ per site with

$N_{vertices,part} = N_{vertices}/N_{part\_total} * (1 + IMB_{partition})$ and $IMB_{partition}$ as described above. The global adaptation cost is determined by

$$T_{adapt,\ global} = T_{startup,\ global} * N_{msg,\ max,\ global} + T_{transfer,\ global} * C_{vol,\ max,\ global}$$

where $T_{startup,\ global}$ is the startup cost and $T_{transfer,global}$ is the time to transfer data per byte across sites.

The gain is calculated similarly as described in local adaptation cost model except that local imbalance factor IMB_{local} is replaced by global imbalance factor IMB_{global}.

$$T_{gain} = T_{remaining,\ noadapt} - T_{remaining,\ adapt}$$

$$= (IMB_{global} * (T_{total} - ((T_{exec} - T_{exec,\ prev})/IMB_{global}))) - (T_{total} - ((T_{exec} - T_{exec,\ prev})/IMB_{global}))$$

$$= (IMB_{global} - 1) * (T_{total} - ((T_{exec} - T_{exec,\ prev})/IMB_{global}))$$

Then, the function $Cost\_Effective_{global}$ (using an error factor Δ_{approx}) delivers true if

$$T_{gain} * (1 - \Delta_{approx}) \geq T_{adapt,\ global}$$

6.5.3 Adaptation Interval

Every local adaptation causes a delay for all the sites, since in a synchronous application all the other sites wait until the site undergoing adaptation resumes its execution. Hence, we find the minimum adaptation interval (I_{adapt}) to amortize the adaptation cost via

$$(T_{gain}/T_{remaining,\ adapt}) * (1 - \Delta_{approx}) * I_{adapt} \geq T_{adapt.}$$

7 Test Plan

7.1 Test Environment

We performed all our tests on our Horus cluster with 16 nodes (enode1 – enode16) each with dual Intel Xeon processors, 512 Mbyte memory, and 512 Kbyte L2 cache and Myrinet interconnect. The Horus cluster runs Debian Linux with kernel version 2.6.6. The first 14 nodes (enode1 – enode14) have a CPU speed of 2 GHz, and the last two nodes (enode15 - enode16) have a CPU speed of 2.4 GHz. Thus we get local heterogeneous environment for our tests. The fronted node (emaster) has four Intel Pentium III Xeon processors with 700 MHz speed and 1 MB L2 cache. In addition some of our tests are performed on a cluster at McMaster University with 64 dual Opterons, i.e. 128 CPUs, with Myrinet interconnect. Both clusters run MPICH-GM 1.2.5.12bs, Zoltan Version 1.52, and ParMetis Version 3.1

For global adaptation, we simulate a grid environment by defining the two subsets of the cluster as two sites. Also, space adaptation is currently simulated (redistributing the work but keeping the maximum number of processes), since we are lacking an MPI which can expand and shrink efficiently on Myrinet/GM device.

7.2 Test Application

As mentioned earlier, we assume that the application using our adaptation framework can be mapped onto a graph structure, where vertices represent the computational workload and edges between them represent the communication workload. To represent our application structure, sample graphs are taken from the University of Greenwich Graph Partitioning Archive [35].

Graph	Vertices	Edges
wing	62,032	121,544
brack2	62,631	366,559
finan512	74,752	261,120
fe_tooth	78,136	452,591
fe_rotor	99,617	662,431
fe_ocean	143,437	409,593
598a	110,971	741,934
wave	156,317	1,059,331
m14b	214,765	1,679,018

Table 4 - Properties of graphs

These graphs represent the basic skeleton of our application and define the computation and communication pattern. Graphs of different sizes and different properties (different ratios of edges vs. vertices) are selected, see Table 4. These graphs describe the structure only, i.e. do not have any weights attached to vertices or edges. In our experiments, we set all weights uniformly.

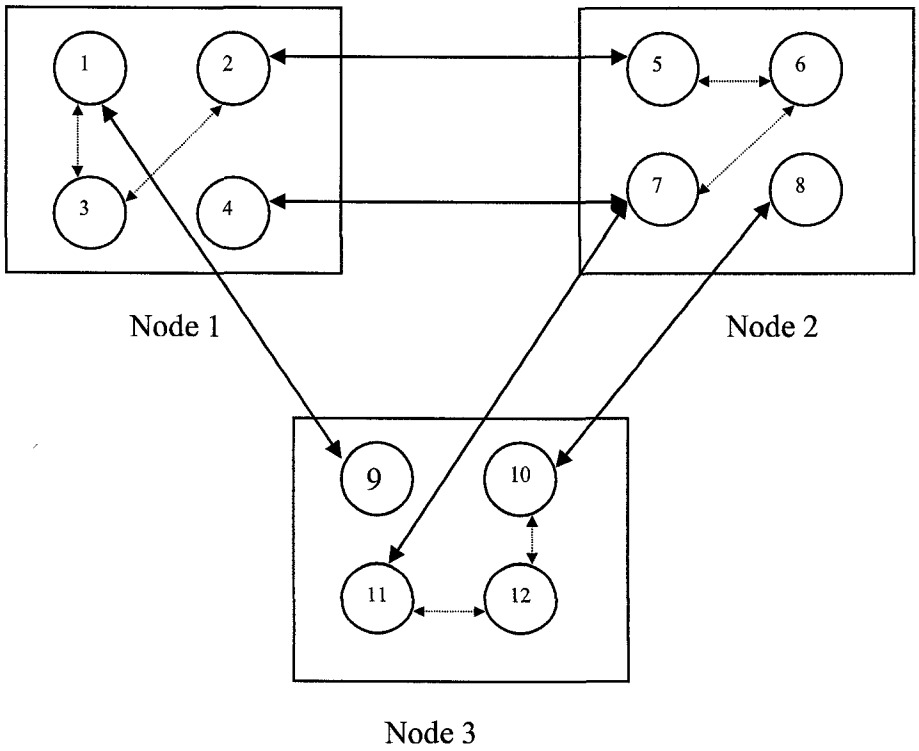


Figure 8 - Test application communication pattern

For the adaptation tests, we have implemented a simple Jacobi finite-difference-method simulation to work on the graphs. This application is mostly synchronous, employing nearest neighbor communication. In each iteration step, after computation step, every partition communicates with all border partitions that do not belong to same node. This is shown in Figure 6. Rectangles denote the nodes on a local site that participate in application execution. Small circles denote the partitions (with their numbers) allocated to each node. Arrows denote the adjacency relationship between various partitions. Solid lines denote the actual communication taking place. For example Partition 7 has adjacent nodes on 4, 6 and 11, but actual communication (data exchange) is done with only partitions 4 and 11, as they do not belong to the same node. The computation step in our application is relatively simple: each vertex in a partition calculates the average of the data stored in all neighboring vertices.

The abstract algorithm of the application is described in Figure 9. Since our framework can use multiple threads for the application execution dynamically, this requires some kind of thread synchronization to ensure that the application executes with correct and updated data in each succeeding iteration. Thread synchronization in our application is achieved through the function *thread_barrier()* as described in Section 6. The communication required by the application (exchanging updated information with neighboring partitions on other nodes) is done by the main thread, whereas the computation workload is divided between all the threads uniformly. For example, if there are two threads active within the application at that time, and each node has 8 partitions, then each thread picks the odd and even number of partitions, thus dividing the computational workload among them equally.

```
thread_barrier()                    //so that all threads start executing the application together
While (iteration < limit)
{
  if (thread_id == main thread)
    {
      Request receiving of data from all the neighboring partitions that are located on different
      processor                              /* receiving data from all neighboring partitions */
    }
        /* Computation starts */
  For all local partitions on each node
   {
      if (number of threads >1)
       each thread picks one partition for computation   //dividing the computational workload
      else
       the main thread picks the partitions one by one and proceeds for computation.
       for (each vertex of the selected partition)
           {
            Calculator () (Computation function)
           }
       end for
  }
  end for (for all local partitions)
  /* Computation ends */
     thread_barrier()          // to make sure each partition has completed their computation and have
                               //updated data to send to neighboring processors
if (thread_id == main thread)
{
     Post send to send out data to all neighboring partitions          /* sending updated data */
     Wait till all requests (sends and receives) are finished
}
thread_barrier()                          //to make sure that all threads have completed one round of
                                          //computation and communication
} //end of while
```

Figure 9 - Abstract algorithm of the test application

8. Test Cases and Experimental Results

8.1 Test Cases

We have several test cases to test the efficiency of our framework, and we designed separate test cases to test for local and global adaptation

- Test cases for proving the efficiency of ATOP

  This test case is designed to show that overpartitioning performs better than partitioning from scratch with regards to adaptation time and is feasible with respect to the edge cuts (potential communication cost). In the case of overpartitioning, we perform these tests with relatively larger number of partitions (up to 256 partitions). Creating many more partitions than the number of processors might provide a benefit in case of overpartitioning since we get more flexibility for data allocation.

- Test cases for the flexible allocation of threads

  With this test case we want to show that different sets of applications provide better runtimes under different thread allocation methods. So we test different thread allocation strategies on various combinations of application pairs to investigate which thread allocation strategy works better for a particular combination of applications. Next we check if the dynamic change in the number of threads and flexible reallocation provides performance benefit in terms of total time of the application. The total runtime of the application is compared with and without this dynamic change of thread allocation and number of threads, showing

the percentage of performance benefit our scheme offers when choosing the best thread allocation strategy.

- Test cases for Local Adaptation
    - time vs. space adaptation

      This test case proves that we can maintain the same progress of the application (in terms of remaining application runtime) by trading in the time vs. space dimension. First we start executing the application of concern with dedicated resource allocation (space sharing) and measure the application runtime with this allocation. The application is executed again with the same resource allocation as used previously (dedicated allocation on same number of nodes) and after some time, the application of concern (A) is coscheduled with another application (B). Now to maintain the same progress, the number of nodes is increased for the application A. We measure the runtime for application A, when switching to time sharing by coscheduling with the application B and compare it with the runtime under a merely dedicated resource allocation scheme (space sharing).

    - Benefits of time sharing

      Through this test case, we want to show that, CPU sharing and node sharing provide better resource utilization than dedication resource allocation, in many cases. First we schedule both the applications on each half of the reserved resources, such that they have dedicated resources

allocated on those nodes. We measure the runtimes of both the applications in this case. Next we coschedule both the applications on all available nodes and we compare the runtime of scheduled applications with and without coscheduling.

- Dynamic resource availability

  In this test case, the resource availability changes dynamically (the job scheduler advises the application that more resources, in terms of more nodes or larger time shares, are available), and we compare the application runtime with and without taking the adaptation decision to utilize the available resources.

- Test cases for global adaptation

  To test the global adaptation we denote different sites by discrete node groups. Moreover, heterogeneity between various sites is simulated by coscheduling the application of concern with another application on one of the nodes groups. Thus, the coscheduled site is slower than the site with dedicated resource allocation exhibiting heterogeneity, central to grid environment. This simulation is required as we lack the real heterogeneous environment on our local cluster.

  - Adaptation for fully malleable application

    In the first test case, we assume that the application is fully malleable, and we divide the workload among all sites equally without taking heterogeneity of various sites (which is reflected in the ARM vector) into account. At the global adaptation step, we adapt and adjust the workload

by taking the proper ARM vector per site into consideration. We compare the total runtime of the application with and without taking the global adaptation decision. Through this test case, we test the efficiency of our adaptation framework for global adaptation.

- Adaptation for constraint malleable application

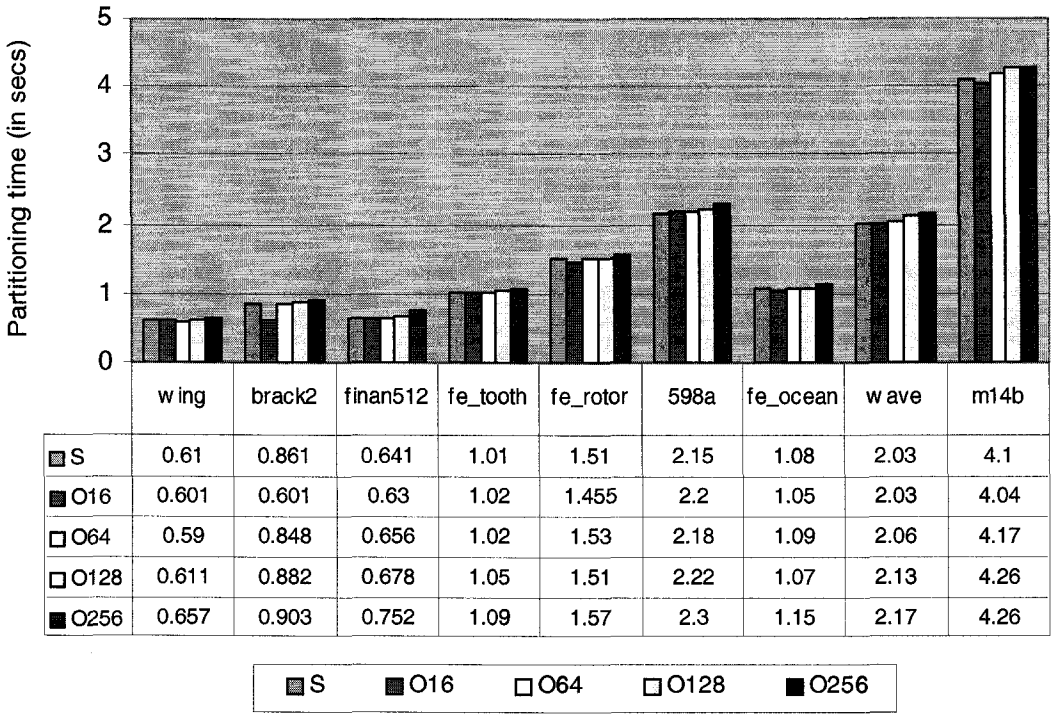
  This class of application does not permit flexible resource allocation; hence, at the global adaptation step, we reduce the number of resources at the faster site to adjust to the worst site progress. We compare the runtime on the faster site with and without resource adaptation to check if after adaptation the runtime on both sites is same or not, using fewer resources on faster site.

8.2 Experimental Results

8.2.1 Test the Efficiency of ATOP

First we show the results for adaptation time when the adaptation is performed in space dimension. The number of nodes are shrunk from 64 to 32, then expanded from 32 to 40, and reduced again from 40 to 16.

The adaptation cost is the sum of the partitioning and the migration cost, and both the costs are analyzed separately. Figure 10 shows the results for initial partitioning time. The results suggest that partitioning time is independent of the number of partitions. This partitioning cost constitutes the initial setup and distribution cost. This initial partitioning test was done on 16 nodes (using 16 nodes on Horus cluster) using partitioning from scratch and overpartitioning with 16, 64, 128 and 256 partitions.

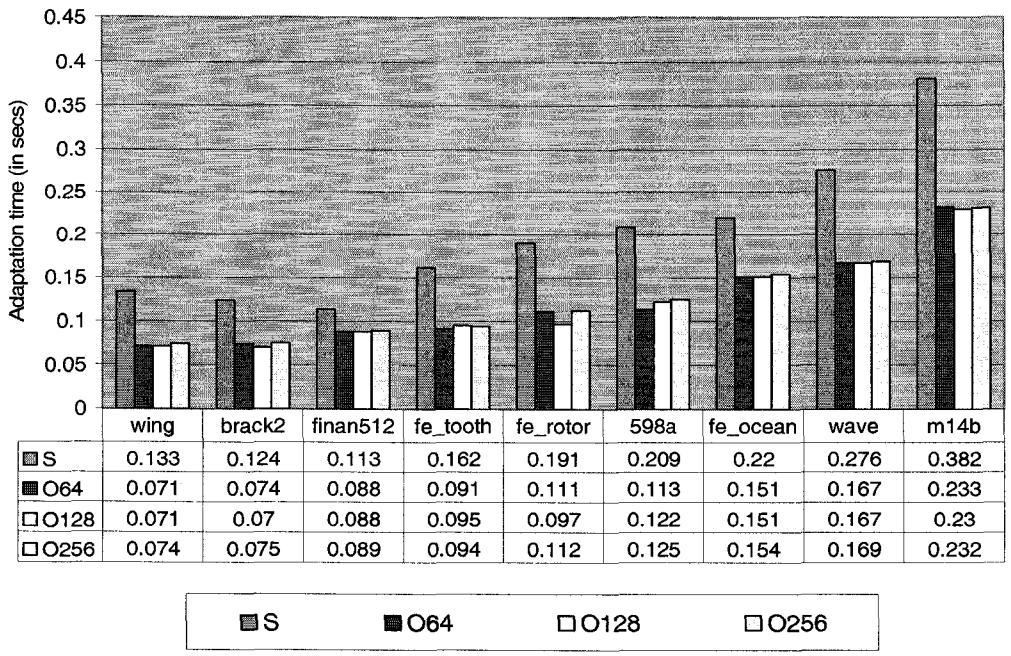


	wing	brack2	finan512	fe_tooth	fe_rotor	598a	fe_ocean	wave	m14b
S	0.61	0.861	0.641	1.01	1.51	2.15	1.08	2.03	4.1
O16	0.601	0.601	0.63	1.02	1.455	2.2	1.05	2.03	4.04
O64	0.59	0.848	0.656	1.02	1.53	2.18	1.09	2.06	4.17
O128	0.611	0.882	0.678	1.05	1.51	2.22	1.07	2.13	4.26
O256	0.657	0.903	0.752	1.09	1.57	2.3	1.15	2.17	4.26

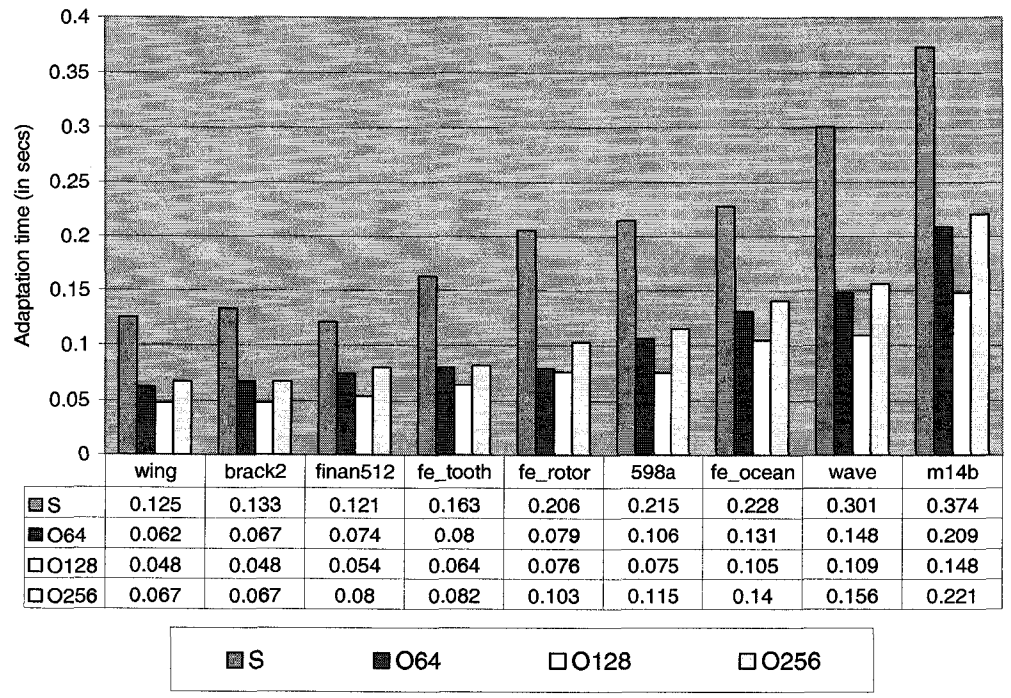
Figure 10 – Initial partitioning time for partitioning from scratch (S) and overpartitioning with 16, 64, 128 and 256 partitions.

Figure 11 and Figure 12 show the adaptation time results for adaptation steps 64→32, 32→49 and 40→16. Results suggest that overpartitioning performs better than partitioning from scratch yielding lower adaptation cost in all the cases. The improvements as compared to partitioning from scratch lie between 46 % and 60 %, where the best is for graph 598a for adaptation step 32→40 and 128 partitions. We also study the variation in adaptation times, when we increase the number of partitions from 64 to 128 to as large as 256 partitions. Results show that 64 and 128 partitions perform better, with 128 partitions providing the best times for all graphs over all adaptation steps.

Thus generally speaking, overpartitioning performs better than partitioning from scratch and best for 128 partitions with respect to adaptation time.

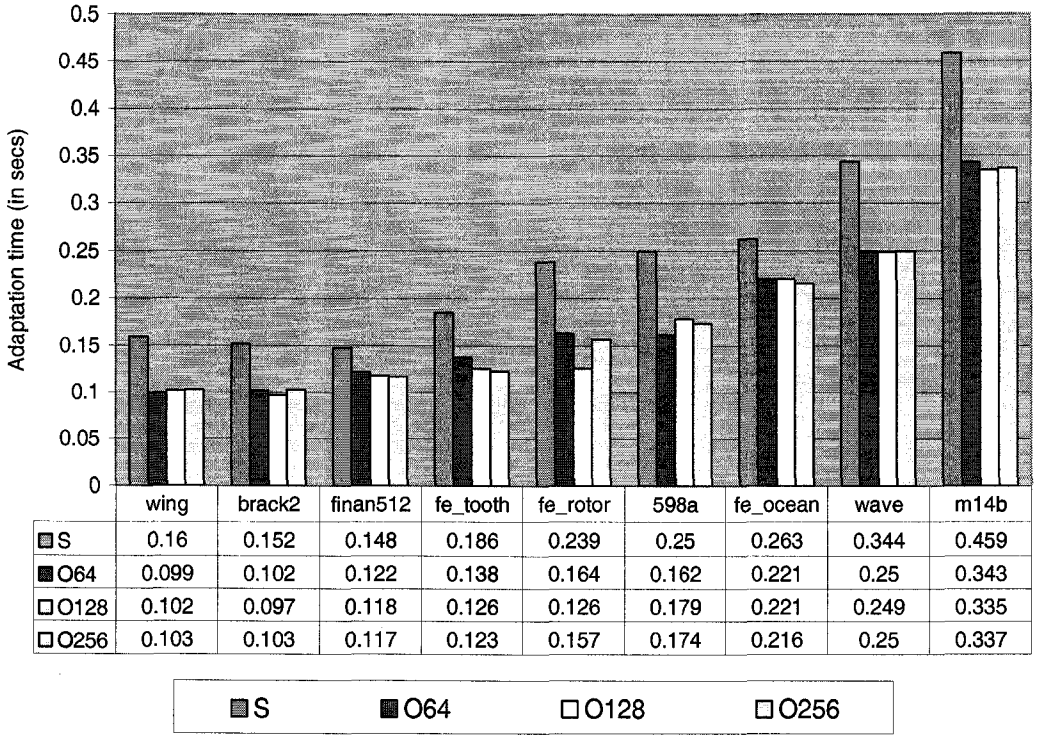


	wing	brack2	finan512	fe_tooth	fe_rotor	598a	fe_ocean	wave	m14b
S	0.133	0.124	0.113	0.162	0.191	0.209	0.22	0.276	0.382
O64	0.071	0.074	0.088	0.091	0.111	0.113	0.151	0.167	0.233
O128	0.071	0.07	0.088	0.095	0.097	0.122	0.151	0.167	0.23
O256	0.074	0.075	0.089	0.094	0.112	0.125	0.154	0.169	0.232



	wing	brack2	finan512	fe_tooth	fe_rotor	598a	fe_ocean	wave	m14b
S	0.125	0.133	0.121	0.163	0.206	0.215	0.228	0.301	0.374
O64	0.062	0.067	0.074	0.08	0.079	0.106	0.131	0.148	0.209
O128	0.048	0.048	0.054	0.064	0.076	0.075	0.105	0.109	0.148
O256	0.067	0.067	0.08	0.082	0.103	0.115	0.14	0.156	0.221

Figure 11 – Adaptation time for adaptation step 64→32 (top) and 32→40 (bottom), show for partitioning from scratch (S) and overpartitioning (O) with 64, 128, and 256 partitions



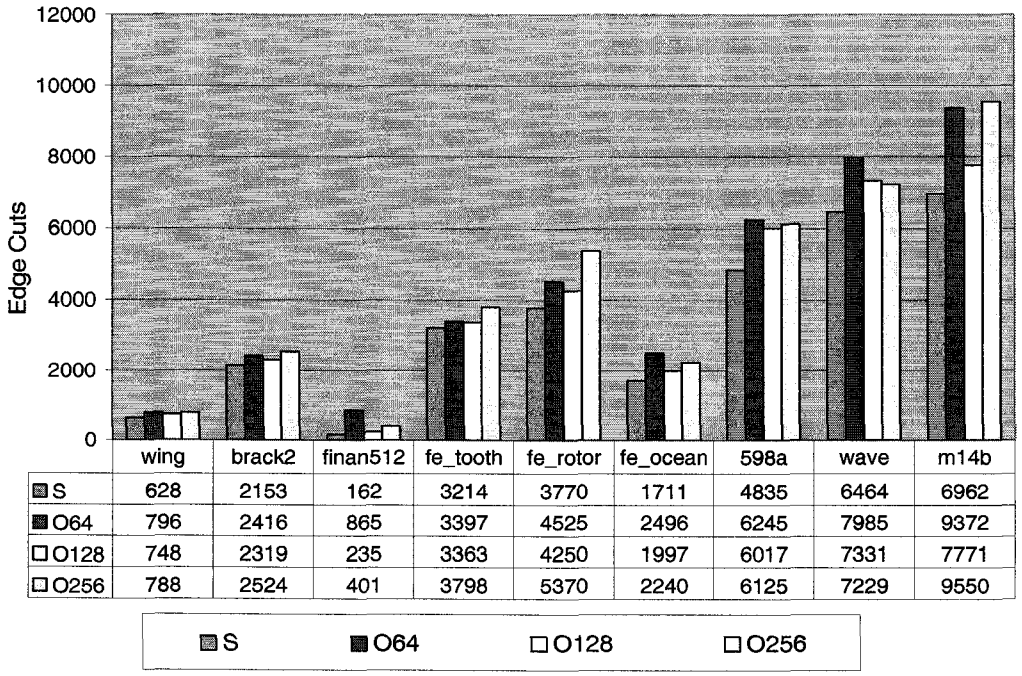
	wing	brack2	finan512	fe_tooth	fe_rotor	598a	fe_ocean	wave	m14b
S	0.16	0.152	0.148	0.186	0.239	0.25	0.263	0.344	0.459
O64	0.099	0.102	0.122	0.138	0.164	0.162	0.221	0.25	0.343
O128	0.102	0.097	0.118	0.126	0.126	0.179	0.221	0.249	0.335
O256	0.103	0.103	0.117	0.123	0.157	0.174	0.216	0.25	0.337

Figure 12 - Adaptation time for adaptation step 40->16 shown for partitioning from scratch (S) and overpartitioning (O) with 64, 128, and 256 partitions

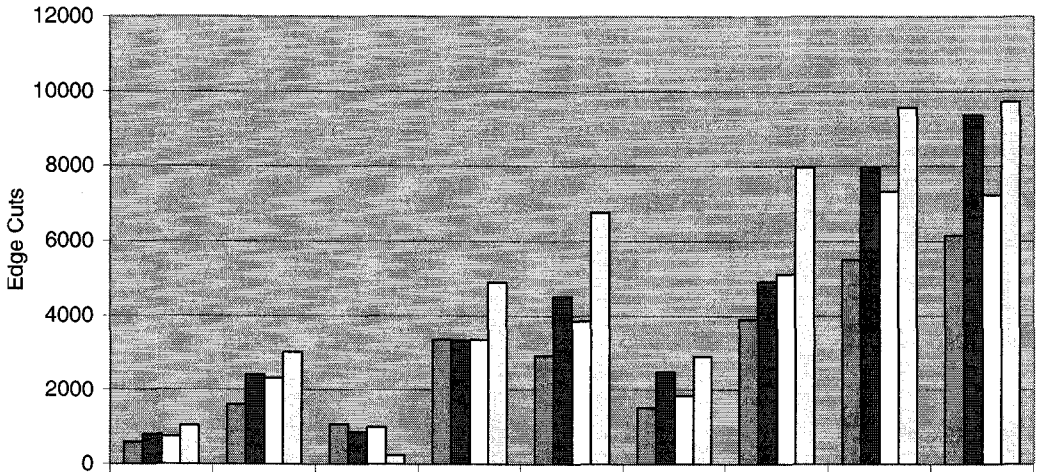
Next we look at the edge cuts which define the distribution quality and the induced communication cost of the application. Edge cuts are the number of edges connecting the vertices that belong to different nodes. Maximum edge cuts are considered since the adaptation framework ATOP-Grid focuses on synchronous applications where maximum communication decides the overall communication cost. This maximum communication cost is derived from the maximum edge cuts per node.

The results for edge cuts, in Figure 13 and Figure 14 show that the edge cuts are best for partitioning from scratch, which takes the exact number of nodes used into account. Increase in edge cuts with overpartitioning for 128 partitions as compared to partitioning from scratch were in the range of 1.2 to 1.5 times, worst being 1.53 times.

Results for 64 and 256 partitions were worse as compared to 128 partitions, the worst being 1.98 times for graph m14b for adaptation step 40→16 with 64 partitions.



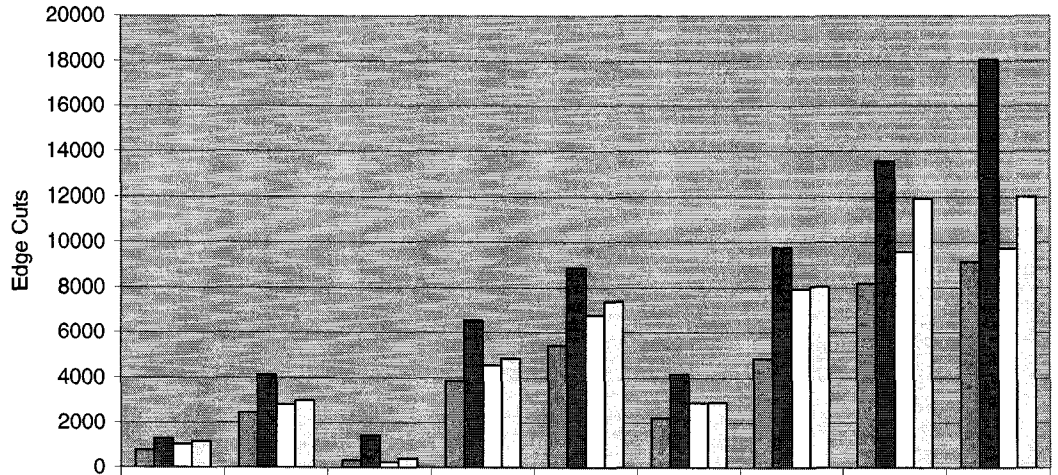
	wing	brack2	finan512	fe_tooth	fe_rotor	fe_ocean	598a	wave	m14b
S	628	2153	162	3214	3770	1711	4835	6464	6962
O64	796	2416	865	3397	4525	2496	6245	7985	9372
O128	748	2319	235	3363	4250	1997	6017	7331	7771
O256	788	2524	401	3798	5370	2240	6125	7229	9550



	wing	brack2	finan512	fe_tooth	fe_rotor	fe_ocean	598a	wave	m14b
S	588	1610	1078	3375	2929	1516	3914	5515	6160
O64	796	2416	865	3346	4525	2496	4918	7985	9372
O128	748	2319	1005	3363	3863	1855	5126	7331	7238
O256	1049	3027	246	4894	6765	2908	7967	9578	9739

Figure 13 – Maximum edge cuts per node for adaptation steps 64→32 (top) and 32→40 (bottom) for partitioning from scratch (S) and overpartitioning with 64, 128 and 256 partitions

Hence, for larger numbers of partitions the probabilistic joining of partitions can partially balance the loss in quality of calculating adequate partitions (like for 128 partitions as compared to 64), whereas, for too many partitions, the chances of degrading the quality of the partitioning is stronger than the probabilistic gain (for 256 partitions).



	wing	brack2	finan512	fe_tooth	fe_rotor	fe_ocean	598a	wave	m14b
S	774	2454	324	3874	5440	2231	4890	8185	9122
O64	1318	4127	1444	6541	8867	4178	9776	13567	18063
O128	1049	2832	246	4575	6765	2908	7967	9578	9739
O256	1171	3027	417	4894	7387	2919	8069	11960	12032

Figure 14 – Maximum edge cuts per node for adaptation steps 40→16 for partitioning from scratch (S) and overpartitioning with 64, 128 and 256 partitions

The overall results for adaptation time and edge cuts suggest that overpartitioning is a feasible approach for data redistribution. Moreover, the results suggest that 128 partitions provide the most stable results. Hence, for rest of the experiments, we have used overpartitioning with 128 partitions as the adaptation approach.

8.2.2 Flexible Allocation of Threads

This test case demonstrates the different thread allocation approaches which lead to different time sharing approaches: CPU sharing, Node sharing and self coscheduling.

Table 5 summarizes the results. As the results show, different combinations of applications give best results under different thread scheduling approach. Application brack2 and wing provides best runtimes for AA-BB, brack2 and fe_ocean for AB-AB and 598a and wave for the A-B approach. Hence, choosing the best approach is important, for better application performance in terms of reduced runtime and for efficient resource utilization.

Applications coscheduled (A and B)	**AA-BB**		**AB-AB**		**A-B**	
	A (secs)	B (secs)	A (secs)	B (secs)	A (secs)	B (secs)
Brack2 (A), wing (B)	***33.60***	***30.64***	36.51	34.24	40.87	37.57
Brack2 (A), fe_ocean (B)	36.32	28.12	***30.04***	***25.57***	32.92	36.84
598a (A), wave (B)	36.23	32.46	37.07	33.26	***32.87***	***31.13***

Table 5 - Runtimes for different thread allocation approaches (CPU sharing, node sharing, and self coscheduling). AB-AB means CPU hyper sharing, A-B node sharing, and AA-BB a combination of node sharing and self coscheduling.

Next we show that, adapting to the correct thread allocation strategy provides a benefit. Figure 15 explains this test case. At first, we schedule both the applications with 1 thread per SMP node (thread allocation type A-B or node sharing), and later, at then next adaptation step, depending on which thread allocation type works best for

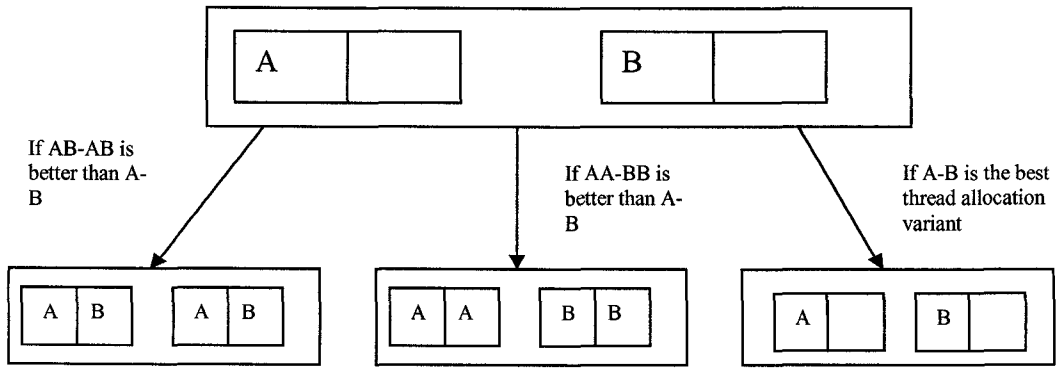


Figure 15 - Test for flexible allocation of threads

coscheduled set of applications; the allocation type is changed to AB-AB (CPU sharing) or AA-BB (node/self coscheduling) or continue with the same allocation.

Results are depicted in Figure 16. Two sets of applications, brack2 & fe_ocean and brack2 & wing are coscheduled, using the A-B thread scheduling approach (node sharing). After 10 % of the application runtime, the thread scheduling approach is changed to AB-AB (CPU sharing) for brack2 & fe_ocean pair and changed to AA-BB (node/self coscheduling) for brack2 & wing pair. Changing the thread allocation leads to a benefit of 8 % for brack2 and 27 % for fe_ocean (brack2 & fe_ocean pair) and of 14% for brack2 and 13 % for wing (brack2 & wing pair).

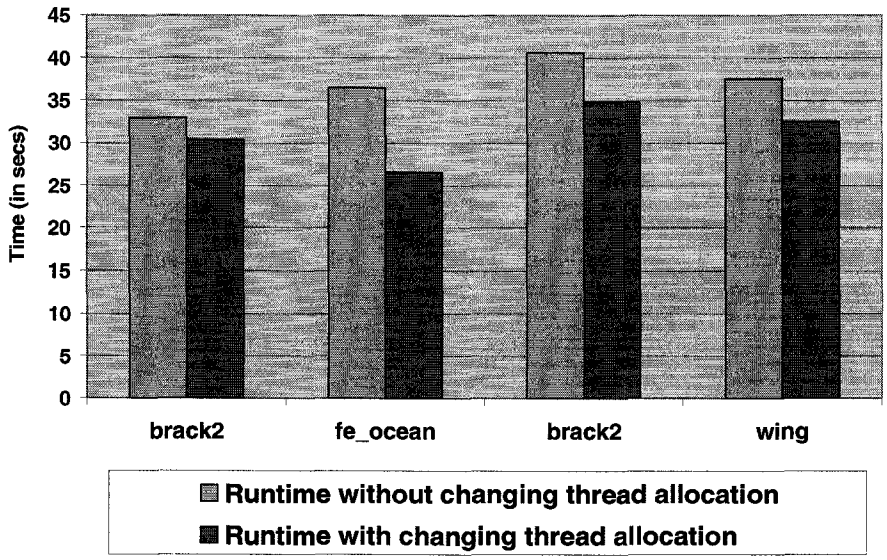


Figure 16 – Runtimes with and without adaptation to dynamic change of thread allocation approach.

8.2.3 Local Adaptation – Time vs. Space Adaptation

Through this test case, we want to demonstrate that the same progress can be maintained in terms of the remaining application runtime if changing to more nodes but less time shares per node, i.e. when switching from the space to the time dimension.

Figure 17 shows the results for this case. The application of concern (fe_rotor, wing and wave) is started under space sharing, and after 10% of its runtime another application is coscheduled. The results demonstrate how the number of resources changes to maintain the same progress (resource sharing variant is CPU sharing for fe_rotor and wing and is node sharing for wave). It is important that the coscheduled application should not suffer from resource sharing beyond the normal time-sharing effect. We take this into account in our case and the slowdown caused due to coscheduling is always less than 2 for all coscheduled applications. Also the adaptation cost is always low, i.e. in the range of 0.25 sec.

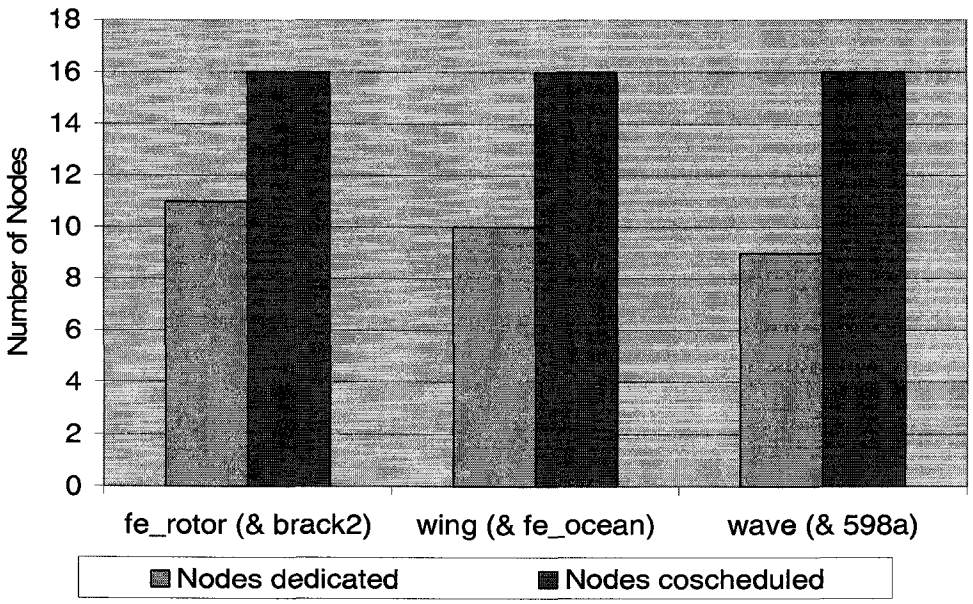


Figure 17 - Resources used (number of nodes) under space sharing and time sharing to keep the same progress/remaining runtime. The coscheduled application is mentioned in parenthesis

The next test shows that how resources are changed (to maintain the same progress), when the coscheduled application changes during the application's runtime. In the following test, the applications coscheduled with wave, with wave being the application of concern, change during the runtime. The application wave (see Figure 18)

executes under dedicated resource allocation on 10 nodes for the first 50 seconds, then it is coscheduled with fe_rotor, and we have to increase the number of nodes from 10 to 14 to maintain the same execution progress. At 120 seconds, fe_rotor terminates and wave is coscheduled with fe_ocean. Thus, we have to change the number of nodes from 14 to 16 (or 15) to maintain the same progress. This test also demonstrates that using the proper thread allocation variant results in better utilization of resources. As shown in Figure 18, with the thread allocation type AC-AC (CPU sharing), wave requires 16 nodes, whereas with A-C (node sharing) approach it requires only 15 nodes to maintain the same progress.

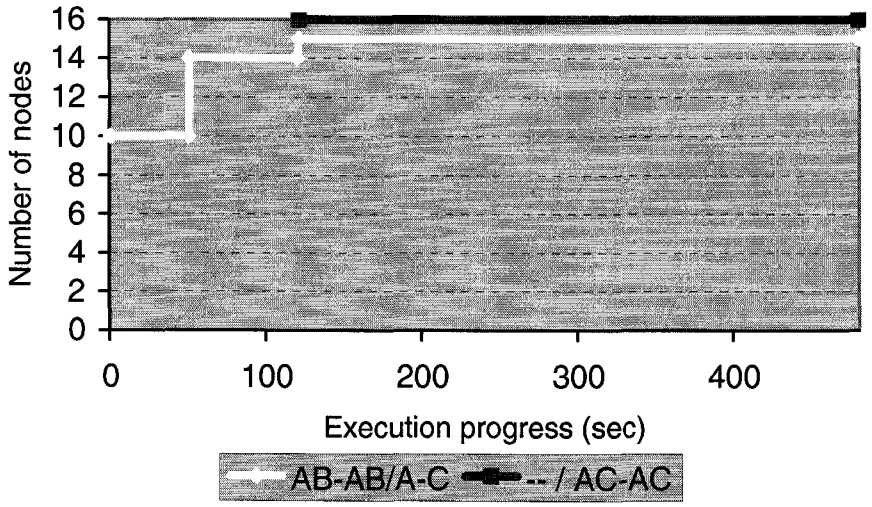


Figure 18 - Execution progress for wave, starting with dedicated allocation, coscheduling fe_rotor from 50sec to 120sec, and coscheduling fe_ocean at 120sec. In the latter case, results are shown for both CPU sharing and node sharing [28]

Similarly the same progress can be maintained when switching from the time to the space dimension i.e. smaller number of nodes but more time shares (or dedicated resource allocation on reduced number of nodes). As shown in Table 6, brack2 and fe_ocean are coscheduled under CPU sharing to execute on 16 nodes. After 10% of the

runtime, one of the applications terminates, and the remaining application gets dedicated allocation on all 16 nodes. Hence, we reduce the number of resources from 16 to 10 in case of brack2 and from 16 to 11 nodes in case of fe_ocean, to maintain the same execution progress for both the applications.

These results also show the benefit of time sharing, brack2 and fe_ocean need 16 nodes under time sharing whereas they need 10+11 = 21 nodes under space sharing to maintain the same progress. We show more results for the benefits of time sharing in the next section.

Application	Sequential runtime	Parallel runtime under CPU sharing on 16 nodes (in secs)	Adaptation Time (in secs)	Number of processors under space sharing to obtain same runtime
brack2	312.45	31.26	0.283	10
fe_ocean	301.87	26.02	0.251	11

Table 6 - Resources used (number of nodes) under time sharing and space sharing to keep the same progress/remaining runtime.

Table 6 also includes a column for sequential time (runtime in a single node), which shows that the parallel version yields an application speedup of almost 10 times for brack2 and 12 times for fe_ocean. Ideal speedup of the application should be 16 times (execution on 16 nodes vs. single node), but communication and synchronization cost, typical for every parallel application, slows down the application execution. Thus, less communicating parallel applications is expected to benefit more, since the communication time is reduced. Moreover, resources are utilized better since the application spends more time in computation rather than exchanging information.

8.2.4 Local Adaptation – Benefits of time sharing

This test case demonstrates that time sharing i.e. CPU sharing or node sharing improves application performance and leads to better resource utilization. This test case is explained in Figure 19. Here, application A and application B are scheduled to execute on half of all available nodes with dedicated resource allocation, i.e. if 16 nodes are available then the application A executes on nodes 1-8 and application 2 is executes on nodes 9-16. Next, both the applications are coscheduled on all available nodes i.e. application A and B executing on all 16 nodes.

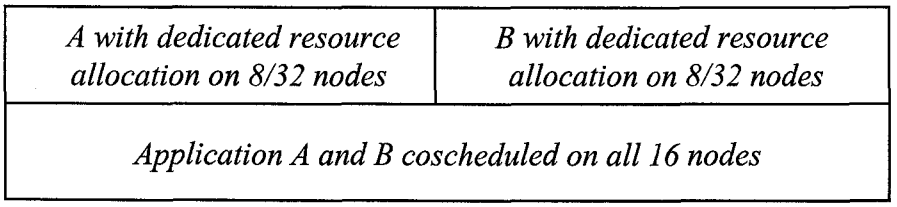


Figure 19 – Test case demonstrating the benefits of time sharing

Figure 20 shows that runtime is reduced when using time sharing (CPU or node sharing) on all available nodes as compared to dedicated resource allocation on half of the nodes. The application sets brack2 & fe_ocean and brack2 & wing are coscheduled using CPU sharing on 16 nodes, whereas wave & 598a and fe_rotor & wave are coscheduled using node sharing on 64 nodes. The dedicated runtime is taken on 8 nodes when using 16 nodes for coscheduling and on 32 nodes when using 64 nodes for coscheduling.

The improvement lies between 13% and 21% for CPU sharing and between 8% and 13% for node sharing. For node sharing, even better gains are expected if the application communicates more [28].

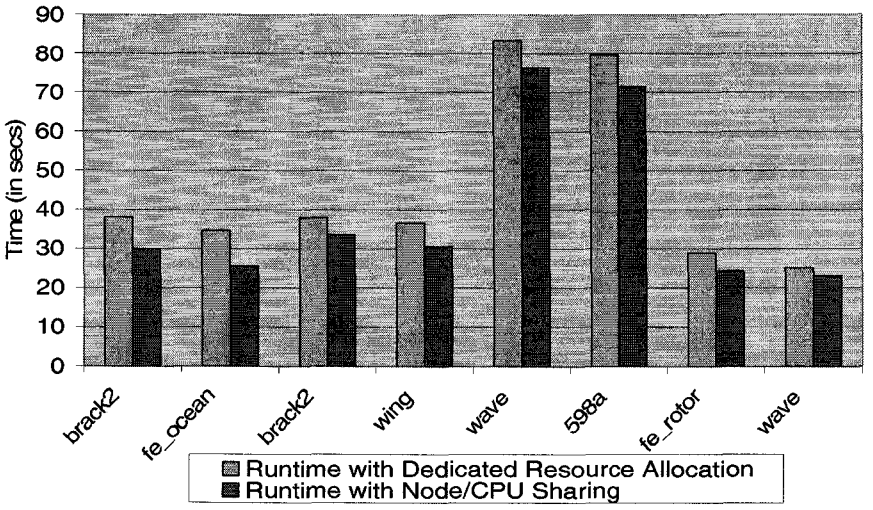


Figure 20 - Runtimes with CPU/Node sharing vs. runtimes for the same applications under dedicated resource allocation

8.2.5 Local Adaptation – Adaptation to Dynamic Resource Availability

In the following test, we demonstrate that adapting to more resources, which may become available dynamically, provides a benefit. We demonstrate the adaptation to resource availability in both the space and the time dimension. For the time dimension we coschedule the application of interest with two other applications on disjoint sets of nodes. Figure 21 explains this test. Here initially at time T1, application A is coscheduled with application B and application C on separate 8 nodes. After some time, the application C terminates, leaving more time shares available for the application A.

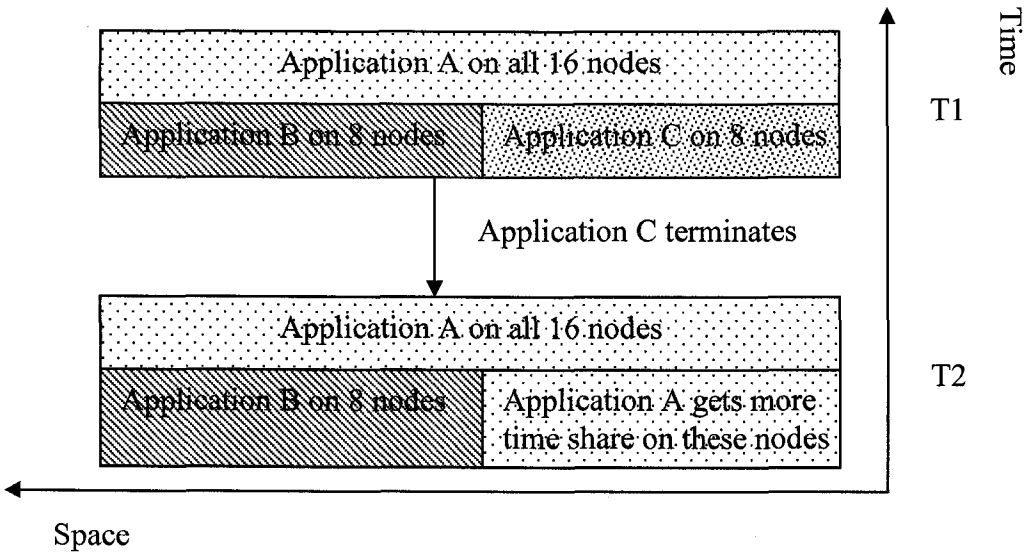


Figure 21 – Dynamic resource availability in time dimension

We have tested 2 cases with brack2 and 598a as the application of interest. These applications are coscheduled with two other applications. After 10% of the runtime, one of the coscheduled applications terminated as shown in Figure 21, i.e. *EShare* became 1 on these 8 nodes. On the 8 nodes where still another application was coscheduled, *EShare*=0.55 (test case brack2) and *EShare*=0.6 (test case 598a). The results in Figure 22 show that adaptation provided a benefit of 20.4 % for brack2 and 17.5 % for 598a.

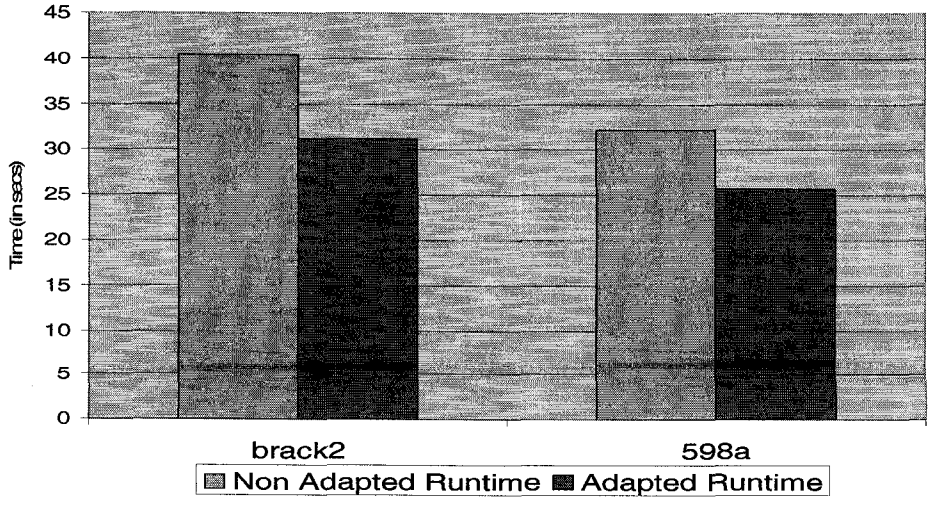


Figure 22 - Runtimes with and without adapting to dynamic resource availability in the time dimension

Next we show the benefit of adapting to the dynamic resource availability in the space dimension. This test is explained in Figure 23. The application starts executing at time T1 on a reserved set of resources. At time T2, extra nodes become available, and the job scheduler informs the application of this resource availability. The application decides whether to adapt or not to use these resources. Resource availability is temporary and, hence at time T3, the job scheduler takes away these extra nodes from the application and the application adapts to continue its execution on the reserved nodes.

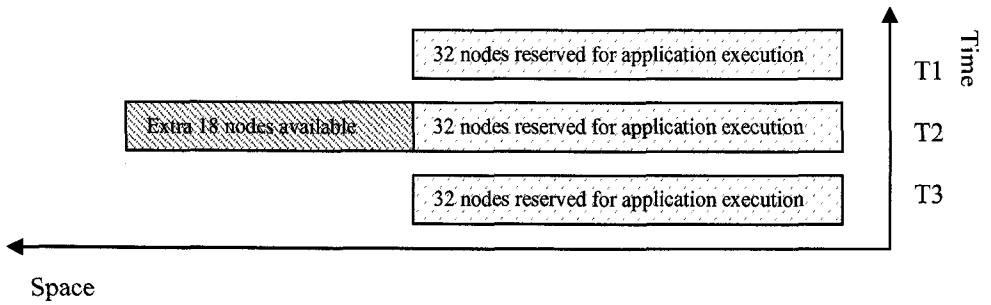


Figure 23 – Dynamic resource availability in space dimension

Figure 24 shows the results. In the following test, extra nodes (from 32 to 50) are available for 60% of the application runtime. Using these extra nodes provided a benefit

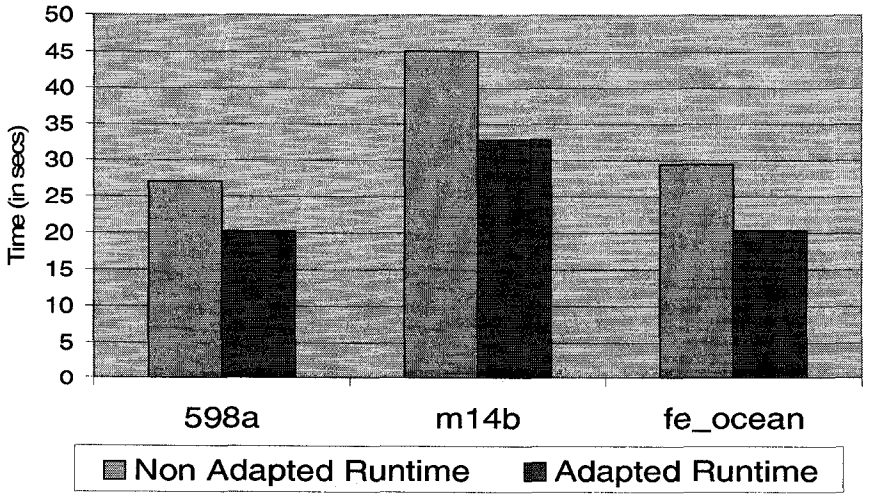


Figure 24 - Runtimes with and without adaptation to dynamic resource availability in the space dimension: 32→50→32.

of 25% for 598a, 27% for m14b, and 29% for fe_ocean.

8.2.6 Global Adaptation for Fully Malleable Application

Here we present results to test the efficiency of our framework with regards to global adaptation for fully malleable applications. As described in Section 8.1, we simulate a grid environment with two node groups as two sites. For a 16 node cluster, the two simulated sites have 8 nodes each or 4 nodes for one site and 12 nodes for the other, and one site is simulated to be slower than the other site by coscheduling another application on one of the sites. For the 64 node cluster, one site has 16 nodes while the other site has 48 nodes, and we assume that the site with 16 nodes is twice as fast as the site with 48 nodes. In both cases the initial workload distribution is done assuming not to know the correct heterogeneity of the sites. This means that for the 16 node cluster, we equally distribute the workload to both the sites. But actually, one site is simulated to be slower than the other. At the global adaptation step, we adapt to the actual heterogeneity (which is a factor of 1.6 on the 8 or 12 nodes vs. the other 8 or 4 nodes). Similarly on the 64 node cluster, double workload is allocated to the site with 16 nodes as compared to the site with 48 nodes, since we assume the site with 16 nodes to be twice as fast as site with 48 nodes. At the global adaptation step, we adapt to the actual heterogeneity which is an equal machine factor (ARM) for all nodes on both sites.

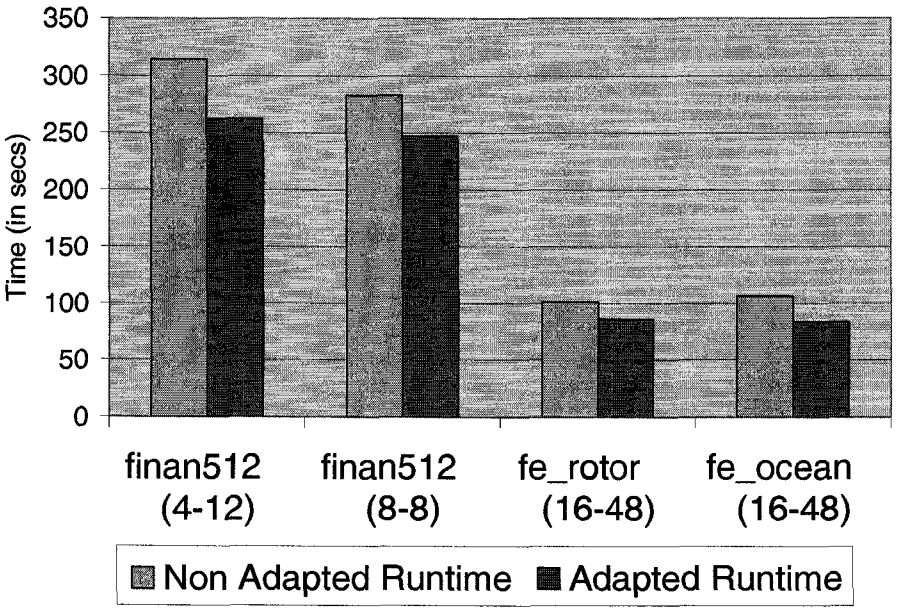


Figure 25 - Runtimes with and without global application adaptation

These results (see Figure 25) show that by adapting to the correct heterogeneity (ARM values), the runtimes are improved by 16.6% (4-12), 12.7% (8-8), 15% (fe_rotor, 16-48), and 21.6% (fe_ocean, 16-48).

8.2.7 Global Adaptation for Constraint Malleable Application

Here, we provide test results for constraint malleable applications. As mentioned earlier, global adaptation for this class of applications is not possible. Hence, we reduce the number of resources on the faster site for efficient resource utilization.

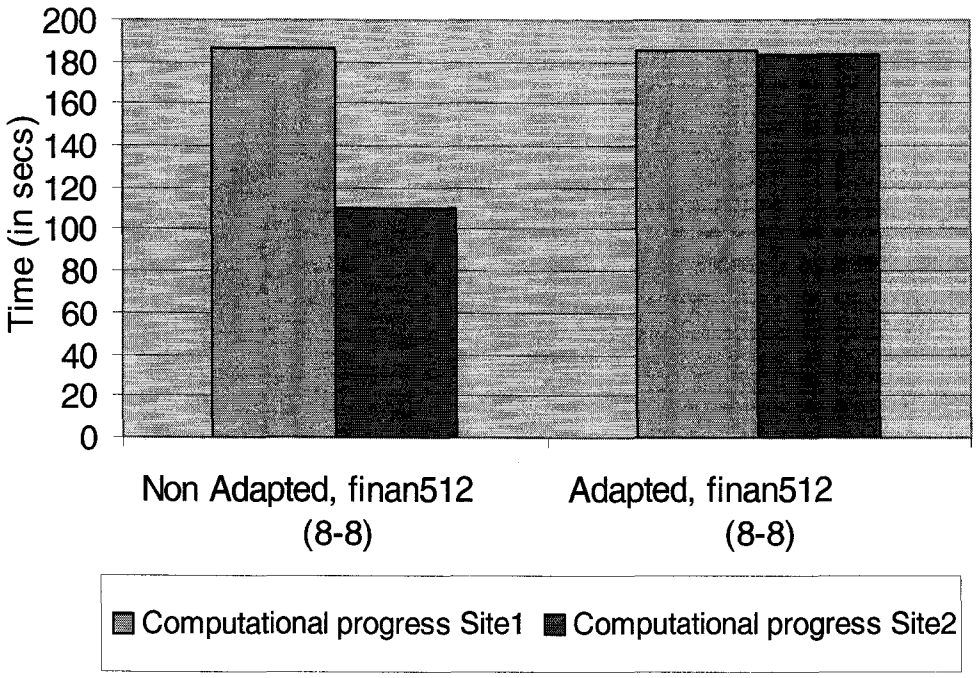


Figure 26 – Runtimes of both sites with and without resource adaptation

These sites are simulated with the same heterogeneity factor (a factor of 1.6 on the 8 nodes vs. the other 8 nodes on a 16 node cluster) as described in the previous test case.

Now since workload is imbalanced on both the sites but global redistribution is not possible, after 10 % runtime (or global adaptation step), resource allocation on the site with the faster nodes was reduced from 8 to 5. As the results show (see Figure 26), the same progress was achieved with fewer resources. Though the runtime could not be improved (due to the application constraints), the utilization of resources was improved. The released resources can be used by the job scheduler e.g. to schedule some local non reserved jobs.

9. Conclusions and Future work

In our approach, we have presented the ATOP-Grid adaptation framework for dynamic workload adaptation in grid environments. The focus of our framework is to achieve balanced progress across all nodes/sites. This goal is achieved by a hierarchical approach which performs the adaptation at different levels, globally across the sites and locally across the nodes of an individual site and additionally at the node/CPU level.

The ATOP-Grid integrates different resource sharing approaches and is integrated with the local job scheduler for efficient resource utilization. In addition, we introduce a new reservation type, computational power, for time vs. space allocation. This provides more flexibility to local job schedulers for meeting the reservation on each local site.

Locally, we support the overpartitioning and partitioning from scratch for workload adaptation, in both space and time dimension. We can dynamically switch between these approaches but our adaptation framework currently uses overpartitioning. Using partitioning from scratch would require to use the hierarchical Zoltan extension [7], such that partitioning and migration can be done at different levels of the hierarchy independently. This extension would be promising for intensely communicating applications, since partitioning from scratch provides a higher distribution quality and, thus, lower communication cost for parallel applications as compared to overpartitioning.

Job scheduler, monitor controller and performance predictor of our framework are simulated but their prototypes are already implemented separately [26] [30]. Integration of all these tools to work with the presented adaptation framework will provide real time monitoring and performance prediction, improving the efficiency of the overall framework.

References

[1] I. Foster, C. Kesselman The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann Publishers, Inc., 1999, ch. 2: Computational Grids p. 15-52

[2] A. C. Sodan, "Loosely Coordinated Coscheduling in the Context of Other Dynamic Approaches for Job Scheduling—A Survey", Concurrency & Computation: Practice & Experience, 17(15), Dec. 2005, pp. 1725-1781.

[3] A. C. Sodan and V. Torra, "Hierarchical Fuzzy Configuration of Implementation Strategies", *Proc. ACM Symposium on Applied Computing (SAC)*, San Antonio / Texas, Feb. 1999, pp. 250- 259.

[4] A. C. Sodan and G. Gupta, "ATOP-Grid for Unified Multidimensional Adaptation of Grid Applications", Accepted for IASTED PDCS, Dallas/Texas, November 2006.

[5] K. Devine, B. Hendrickson, E. Boman, M. St.John and C. Vaughan, "Design of Dynamic Load-Balancing Tools for Parallel Applications", *Proc. ICS*, Santa Fe, NM, 2000

[6] G. Karypis, K. Schloegel, V. Kumar, "ParMetis—Parallel Graph Partitioning and Sparse Matrix Ordering Library, Version 3.1", August 2003

[7] J. D. Teresco, J. Faik, and J. E. Flaherty, "Hierarchical Partitioning and Dynamic Load Balancing for Scientific Computation", *Proc. Workshop on State-Of-The-Art in Scientific Computing: 7th International Conference, PARA 2004*, Lyngby, Denmark, June 2004.

[8] Marr D., Binns F., Hill D. L., Hinton G., Koufaty D. A., JMiller J. A., Upton M., "Hyper Threading Technology Architecture and Microarchitecture", Intel Technology Journal-Vol. 6, February 2002.

[9] Y. Zhang, M. Burcea, V. Cheng, R. Ho and M. Voss, "An Adaptive OpenMP Loop Scheduler for Hyperthreaded SMPs", *Proc. PDCS*, San Francisco, CA, September 2004

[10] OPENMP home page http://www.openmp.org/

[11] MPI home page http://www-unix.mcs.anl.gov/mpi/mpich2/

[12] C. Jung, D. Lim, J. Lee, and S. Han, "Adaptive Execution Techniques for SMT Multiprocessor Architectures", *ACM SIGPLAN Symp. on Principles and Practice of Par. Programming (PPoPP)*, Chicago, Illinois, June 2005.

[13] F. Berman, R. Wolski, H. Casanova, W. Cirne , H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, D. Zagorodnov, "Adaptive Computing on the Grid Using AppLeS", *IEEE Transactions on Parallel and Distributed Systems*, V.14 No.4, pp.369-382, April 2003.

[14] H. Casanova , G. Obertelli , F. Berman , R. Wolski, The AppLeS parameter sweep template: user-level middleware for the grid, Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM), p.60-es, November 04-10, 2000, Dallas, Texas, United States

[15] AMWAT Project homepage http://grail.sdsc.edu/projects/amwat/

[16] H. Chen, M. Maheswaran, "Distributed Dynamic Scheduling of Composite Tasks on Grid Computing Systems," International Parallel and Distributed Processing Symposium: IPDPS 2002 Workshops, 2002.

[17] A.C. Sodan and X. Huang, "SCOJO—Share-Based Job Scheduling with Integrated Dynamic resource directory in Support of Grid Scheduling", *Proc. HPCS*, Sherbrooke, 2003.

[18] Taylor, V. E., and Bryan, G., "Dynamic load balancing of SAMR applications on distributed systems, *Proc. ACM/IEEE Conference on Supercomputing,* Denver, Colorado, Nov. 2001.

[19] A. Maharti and D.L. Eager, "Adaptive Data Parallel Computing on Workstation Clusters", Journal of Parallel and Distributed Computing, Vol. 64, No. 11, 2004, pp. 1241-1255.

[20] K. Schloegel, G. Karypis, V. Kumar. "Wavefront Diffusion and LMSR: Algorithms for Dynamic Repartitioning of Adaptive Meshes", IEEE Trans. Parallel Distributed Systems, Vol. 12, No. 5, 2001, pp. 451-466

[21] J. Weinberg and A. Snavel, "Symbiotic Space-Sharing on SDSC's DataStar System", 12th Workshop on Job Scheduling Strategies for Parallel Processing, Saint-Malo, France June 26, 2006

[22] A. Arefeen, "Time Adaptation for Parallel Applications in Unbalanced Time Sharing Environments", Master's Thesis, Supervisor A. Sodan, University of Windsor, May 2005.

[23] H. Kameda, S. F. El-Zoghdy, I. Ryu and J. Li, "A performance comparison of dynamic vs. static load balancing policies in a mainframe – personal computer network model," Proc. IEEE Conference on Decision and Control (IEEE CDC 2000), Sydney, Australia, pp. 1415-1420, Dec. 2000.

[24] Y. Li and, Z. Lan , "A Survey of Load Balancing in Grid Computing." Proc. CIS (Comutational and Information Science, 2004, pp. 280-285

[25] M. Ripeanu, A. Iamnitchi and I. Foster, "Cactus Application: Performance Predictions in Grid Environments", Euro-Par 2001: Parallel Processing, Proceedings of

7th International Euro-Par Conference Manchester, UK August 28-31, 2001, Springer, 807-816.

[26] A.C. Sodan and L. Liu, "Dynamic Multi-Resource Monitoring for Predictive Job Scheduling with ScoPro", Tech. Report No. 05-002, University of Windsor, Computer Science, Feb. 2005.

[27] A.C. Sodan and L. Han, "ATOP—Time and Space Adaptation for Parallel and Grid Applications via Flexible Data Partitioning", *Proc. ACM Workshop on Adaptive and Reflective Middleware*, Toronto, September 2004.

[28] A.C. Sodan and Garima Gupta, "Time vs. Space Adaptation with ATOP-Grid", Submitted to ACM Workshop on Adaptive and Reflective Middleware, Melbourne, Australia 2006.

[29] W. Smith, I. Foster, and V. Taylor, "Scheduling with Advanced Reservations," International Parallel and Distributed Processing Symposium (IPDPS '00), May 2000

[30] B. Lafreni and A.C. Sodan, "ScoPred—Scalable User-Directed Performance Prediction Using Complexity Modeling and Historical Data", *Proc. JSSPP*, Cambridge, 2005, Springer.

[31] A.C. Sodan and L. Lan, "LOMARC—Lookahead Matchmaking in Multi-Resource Coscheduling", *Proc. JSSPP*, New York / USA, June 2004, Springer.

[32] A.C. Sodan and X. Huang, "Adaptive Time/Space Sharing with SCOJO", Proc. *HPCS*, Winnipeg, Canada, May 2004

[33] R. Wolski. Dynamically Forecasting Network Performance Using the Network. Weather Service, *Journal of Cluster Computing*, 1(1), 1998, pp. 119-132.

[34] Y. Zou. Masters Thesis work in progress, University of Windsor.

[35] The Graph Partitioning Archive http://staffweb.cms.gre.ac.uk/~c.walshaw/partition

[36] Moab Workload Manager http://old.clusterresources.com/moabdocs/22.4dynamicjobs.shtml

Vita Auctoris

Name: Garima Gupta

Place of Birth: Indore, India

Education:

2004-2006 M.Sc. Computer Science
University of Windsor
Windsor, Ontario, Canada

1999-2003 Bachelors of Computer Applications (Hons.)
Devi Ahilya University
Indore, India