2005

Time adaptation for parallel applications in unbalanced time sharing environment

Ahsanul Arefeen
University of Windsor

Time Adaptation for Parallel Applications in Unbalanced Time Sharing Environment

By

Ahsanul Arefeen

A Thesis

Submitted to the Faculty of Graduate Studies and Research

through the School of Computer Science

in Partial Fulfillment of the Requirements for

the Degree of Master of Science at the

University of Windsor

Windsor, Ontario, Canada

2005

Abstract

Time adaptation is very significant for parallel jobs running on a parallel centralized or distributed multiprocessor machine. The turnaround time of an individual job depends on the turnaround time of each of its processes. Dynamic load balancing for unbalanced time sharing environment helps to equally distribute the work load among the available resources, so that all processes of a single job end almost at the same time, thus minimizing the turnaround time and maximizing the resource utilization.

In this thesis we propose and implement an approach that helps parallel applications to use our library so that it can adapt in time dimension (if running in a time sharing environment) without changing the space allocation. This approach provides an interface between application, monitoring information, the job scheduler and a cost model that considers application, system and load-balancing information. This interface allows binding of different adaptation approaches for synchronous adaptation and semi-static remapping. We also determined job types for what this approach is suitable and at the end we present results from our test run on a 16-node cluster with synthetic MPI programs and a time adaptation approach, demonstrating the gain from our approach. In this work, we make extension of existing ATOP [11] work. We directly use their over partitioning strategy. But unlike ATOP, applications can use our adaptation library and adapt dynamically. We also adopted the dynamic directory concept used in SCOJO [8].

Dedication

To my parents who raised me and guided me through the right path.

Acknowledgement

There are a number of people without whom this thesis would not have been completed. I acknowledge my gratefulness to my thesis supervisor Dr. Angela C. Sodan, who always helped me with her guidance, support, motivation and enthusiasm. I am also grateful to my supervisor for providing me the test environment and technical support from the formative stage to the final draft. I thank Han Lin for his work related to Zoltan and over-partition, providing the base of my work.

I thank my thesis committee members: Dr. Karen Y. Fung, Dr. Ziad Kobti and Dr. Yung H. Tsin for their precious time and comments.

I extend my deepest appreciation to my mother and my brother for their constant financial and moral support.

Table of Content

List of Tables

List of Figures

1. Introduction

In parallel applications, usually there are multiple interacting processes running on different Central Processing Units (CPUs). This helps to overcome the physical limitations of processing capability of a non-parallel computer system. Depending on the interaction between these parallel processes, parallel jobs can be classified into three different types. They are

- Jobs with tight coupling,
- Jobs with balanced processes and loose interaction,
- Jobs structured as work-pile of independent tasks.

The first type of jobs usually consists of a certain number of processes and they are communication intensive. The processes of the second type do not interact with other processes very frequently, but the turnaround time depends on the finishing time of the slowest process. The third type of jobs is basically worker processes and they are very flexible. They can change the number of processes during runtime (malleable) and are very suitable for Network of Workstations (NOW) environment.

Load balancing is a critical issue for achieving good performance in any parallel system [1]. A great deal of research has been done on improving load balance of particular algorithm or application, but the general purpose load balancing research deals with process migration in operating system and more recently in application framework [6]. Applications in the areas like very large-scale integration (VLSI), computational fluid dynamics (CFD), meteorological simulations, structural dynamics, magnetic and thermal dynamics use a load balancer to perform the initial load balancing, eventually several application show dynamic behavior (in communication structure) during runtime. That's why it requires employing a dynamic load balancing strategy. For achieving scalable performance, it is important to evenly distribute the workload among the processing nodes [3]. The variation in system load and application requirements during execution is imminent in a real environment. The distributed and global availability of runtime load information and its maintenance require dynamic exchanges of information between the workstations [5]. This dynamicity of system and application load and the limitation of a priori knowledge of parallel application behavior imply the requirement of dynamic load balancing.

Our approach is to make sure that all parallel processes of a parallel application proceed in the computation approximately to the same extent and finish at about the same time which is similar to the load balancing. In an ideal homogeneous environment, all the similar processors are allocated with equal amount of workload, so that all the processes can run to completion at the same time. But this is not feasible under different environments like 1) heterogeneous environments with heterogeneous CPUs and/or different size of memory running at different speed, 2) NOW environment with different background load at different processors, 3) time/space sharing environment where loosely coordinated processes are coscheduled on different processors/nodes and again different processors has different number of processes scheduled/coscheduled on them from different set of parallel applications. In such time/space sharing environment, multiple applications can run per processor determined by a certain multiprogramming level [9, 8]. Similar situation can occur for cross-site jobs in computational grids if different time share is allocated on different sites. In such cases of imbalances, if not adapting the workload, the slowest processor or highest multiprogramming level would determine the performance of the whole application. We present a framework to address load balancing in such situations of imbalance along the time axis with the following main goals of supporting balancing with imbalanced workload assignment, including certain coscheduling effects especially dynamic resource availability changes along the time direction and cases where the above multiprogramming occurs on subsets of processors. We confine our approach to rigid jobs that do not change the number of parallel processes of an application during execution period and these jobs are non-preemptive. But we allowed jobs to be time malleable so that a job can dynamically adapt in dynamic time sharing environment. Our approach provides the following solutions for parallel applications to adapt dynamically:

- An interface between application and system providing an integration of application level and system level.
- A software framework in the form of adaptation library, enabling to bind different load balancing strategies.
- Support of job scheduler initiated adaptation.

We present results from our experiment of our 16 dual Xeon node cluster. In order to run our adaptation performance test, we developed a simulated scheduler and synthetic applications. We also explain what type of job is suitable for our time adaptation approach.

2. Review of the State of the Art

There are four different basic load balancing strategies along two axes [2], either local or global in one axis or centralized or distributed in another axis. In this chapter we will review these strategies and run-time systems and then we will introduce the time malleability and space malleability problems. Finally, we will explain how graph partitioning helps to deal with these problems.

In dynamic load balancing, a monitoring system keeps information about the workload of each processor during execution time and invokes the balancing operation between the heavily loaded processors to the lightly loaded processor when imbalance is found by the monitoring system beyond a certain level of imbalance. Balancing operation can also be invoked when the monitoring system finds a significant amount of change in resource availability. This invocation can be performed in centralized or distributed manner.

2.1 Centralized load balancing model

Depending on the location of the load balancer, the load balancing strategy can be categorized as centralized or distributed. When the load balancer is located at a master node (processor) that has the global knowledge of other processor's load information and the master node initiates the workload balancing, the model can be characterized as centralized load balancing model. Here all the processors take part in the synchronization and send their load information to the central load balancer. The central load balancer, after receiving load information, calculates the new load distribution and related work movement and redistribution profit. If migrating workload is profitable, the balancer sends instruction to the worker processor to do so mentioning the recipient information. The receiving processor waits until it receives the instructed amount of work.

2.2 Distributed load balancing model

In distributed load balancing, the load balancer is placed on every processor and instead of sending a load profile to the master node, it can be broadcasted to all other nodes or only to the neighbors depending on different model. This helps to circumvent the communication bottleneck problem in the centralized model and eliminates the need to instruct other nodes as well.

The two popular ways of distributed load balancing are work sharing and work stealing, even though they are not exclusively for distributed load balancing. In work sharing when some new work load is generated, the generating processor attempts to migrate some of its load to the other processor expecting them to be underutilized [7]. This is also called sender initiated load balancing. On the other hand, in work stealing, underutilized processors request work load from overloaded processors. In either cases the request may be denied when the destination processor is overloaded (load sharing) or the sender processor does not have enough workload (work stealing). Both of these strategies are suitable for fine grain parallel applications. Global system knowledge can be acquired by agents running on each node and they exchange the load profile in a collaborative manner [4].

2.3 Comparison between centralized and distributed model

The centralized load balancing model can not perform well enough when the number of working machines increases. This has the limitation of scalability and the performance degrades with the increased number of communication overhead. But this can help to have simple global load knowledge and is suitable with a small number of nodes. Distributed load balancing model can offer better scalability as this does not require communicating with a single master node containing the load balancer [2]. But again the distributed load balancing model suffers from having a better load profile of the entire system. One of the attractive aspects of distributed model is that it increases locality. In the centralized load balancing model, it requires sending all-to-one profile send and followed by a one-to-all instruction. But in distributed load balancing model, it requires all-to-all or one-to-neighbor broadcasting. Considering the scalability problem, the distributed model outperforms the centralized model but for a small number of nodes, hence a centralized model can be more suitable.

2.4 Work sharing Vs true load balancing

In work sharing, when a processor creates new work, it makes an endeavor to migrate some of its work to other processors hoping that they are not heavily loaded as this processor. This is particularly suitable for fine grained (multithreaded) application. When they create new threads, they try to migrate some of newly created threads to other processors. In load sharing, two important components are: allocator and scheduler. The allocator is responsible for deciding where the job will be executing and the scheduler is responsible for deciding when a job will be getting its share of the CPU. There is more migration of processes or threads in work sharing compared to work stealing [7]. If all the processors are heavily loaded, there is always some migration by the work sharing scheduler.

In order to compare the two forms of load distribution, load balancing makes sure that each processor has almost the same amount of work load in order to increase the system utilization. Most of the time, load balancing is dependent on the accuracy of load profiling. Inaccuracy of load information may lead to worse performance. In preemptive migration schemes [56] of load balancing, the overhead related to the preemptive migration is considered for the profitability of performance gain as it reduce the mean delay (queuing and migration) by 35-50%, compared to non-preemptive migration. Load sharing on the other hand is weaker than load balancing and implemented with non preemptive migration of processes. As in load sharing policy, there is lag of global load knowledge. Load sharing does not ensure equal distribution of load but it is easier to implement and approaches the heterogeneity in a more convenient way.

2.5 Comparison between Synchronous and Asynchronous load balancing

Depending on the load-redistribution, dynamic load balancing can again be classified into synchronous and asynchronous model. In synchronous dynamic load balancing, the application needs to stop so that it can redistribute the workload among its processes and thereby reducing the imbalances; then the application can continue to execute at the end of synchronization. The total process of synchronization is performed in two steps. First, repartition of required data for each process and then migrate the newly repartitioned data to the destination processes. This approach is used by our strategy of load balancing adaptation.

In asynchronous load balancing, instead of stopping and synchronizing, processes continue to execute, and depending on work-sharing or work-stealing method, the lightly loaded processes communicate with heavily loaded processes for additional work. If both parties agree, they migrate the workload in an asynchronous manner. Asynchronous load balancing provides the opportunity of latency hiding by overlapping communication and computation.

2.6 Related Work

There exist a few load balancing libraries that provide multiple approaches. Zoltan [12] library includes recursive coordinate bisection, recursive inertial bisection, refinement tree based partitioning, ParMETIS [18], Jostle and octree partitioning. It provides a generalized interface and data structure that the applications do not have to depend on but use them in the call back functions. ParMETIS [18], which is basically an extended version of METIS [19], provides an MPI (Message Passing Interface) based parallel library that implements a variety of algorithms for partitioning and repartitioning of unstructured graphs and meshes. A measurement based automatic load balancing framework is presented in [20]. Parallel applications are projected to this framework as collection of computing objects which communicate with each other. There is a load balancer database, which is responsible for coordinating load balancing activity and helps to form an object communication graph. Each processor collects a partial object-communication graph consisting of local objects. The load balancer strategy decides which object is to migrate for better performance and pass this information to the framework. In [21] introduces an approach of load balancing in distributed environment by means of thread migration. They worked on top of Chant, which is a distributed lightweight thread package for point-to-point communication between threads. They also proposed a layered load balancing approach where the bottom layer contains the load balancing routines. The middle layer contains the load balancing commands and the topmost layer does the actual load balancing function.

Flexible co-scheduling (FCS) [9] address the existing problems of gang scheduling and implicit co-scheduling. They address the fragmentation, load imbalance and the heterogeneity problem in particular. They come up with another parallel scheduling algorithm similar to the gang scheduling algorithm. In FCS, they classify the processes depending on their demand and behavior. Processes requiring gang scheduling are gang scheduled and the rest are used to fill out the fragmentation. The load imbalance and heterogeneity problem is solved with classification on per-process basis. The classification process is done after monitoring communication behavior and detection of possible load imbalance. [22] describes an approaches which is close to our approach. Their proposed and implemented system that uses a compile time analysis in order to capture the access pattern and make instrumentation to the code with calls to the runtime library. The runtime system uses these compile time information to facilitate partitioning of work depending on locality of data access and resource availability. The locality problem is equally important as load imbalance problem. Here the runtime library works as a bridge between the operating system and application, and monitor process activity in order to facilitate cooperative scheduling flexibility. They also perform runtime measurement and are able to correct workload allocation dynamically if required. But their approach is not very specific about the application characteristics and co-scheduling affects.

EARTH [3] describes the design of nine dynamic load balancing algorithms focusing on the complexity that arises due to the fine granularity of multi-threaded execution environment. They also implement these algorithms on multithreaded multiprocessor test-bed and evaluate the performance. They cover a wide range of load balancing strategies. They also design a suite of stress tests for the analysis of the strengths and weakness of load balancers and they find that dynamic load balancer utilizing history information and employing both the work stealing and work sharing performs well in various kinds of applications. Performance varies significantly with the change of grain size. They also find other effects like polling interval, number of nodes, and communication topology on the performance of load balancer. Recent load balancing encompasses the resources beyond the typical computational resources. They include memory, network and I/O. The opportunity cost framework [23] optimizes CPU load and reduces the maximal utilization of CPU for those jobs that perform I/O and inter process communication. A job is assigned to a machine minimizing the sum of cost of resources, where each resource has a cost considering CPU load, memory available etc. In [24], memory-I/O-based policy is recommended which minimizes the page fault within the co-scheduled jobs. In [25], they profiled an application (both communication and computational memory access) and machine, and predicted the performance after convolution of their profile. This application profile can be incorporated with adaptation approach to get the better performance.

2.7 Malleable Jobs

Definition *Space-Malleable*: An application is space malleable if it is able to change the number of processes dynamically during its execution [11].

Definition *Time-Malleable*: An application is time malleable if it is able to adapt dynamically with varying time shares on different processors.

Definition *Time-Moldable*: An application is time moldable if it is able to run with different work load on different processors. The work load on each node is determined at the time of startup and remains constant during execution.

In our study we considered time malleable jobs while keeping the total number of processes of an application static. Putting more work load on a node than on others works when processor speed is different and workload is adjusted to keep the computation in synchrony. Due to different processor share, some processes can take more workload than other processes of the same application and computations that have loose or little dependencies. Besides, putting more computation on a node does not overly increase the communication with other nodes, but increase only marginally. For latency hiding, the application model needed is a coarse grain work pile that does not communicate frequently and does large communication at a time and can have relaxed dependencies.

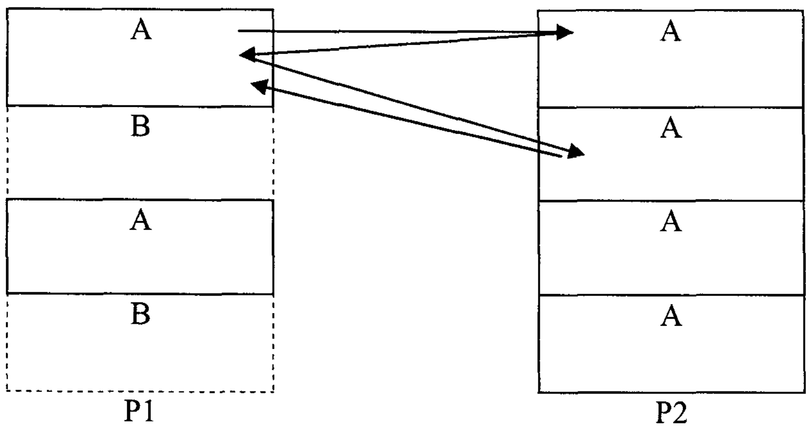

Figure 1. Two application A & B on nodes P1 & P2. A is a parallel application with mutual communication dependencies [14]

In figure 1, it is depicted that different time share can cause delay in computation due to the process dependencies as they don't run in synchrony. Assuming that we have loose coordination with spin-blocking support, parallel application A on P1 releases CPU when waiting for a message from A on P2 and spin-block time outs. This means switching cost includes the cache locality, which is infeasible for frequent short communication. Approaches like AMPI [15] or fine-grain multithreading, can solve this problem. In our case, we assume that the application supports load balancing at the application level, which is suitable for dynamic applications that needs load balancing anyway.

Definition *Work Unit*: A work unit is a migratable description of a piece of work that is not yet in execution. Such work requires a functional code and data description. The functional description is a function or procedure, or a loop-slice. The data may be simple parameters or complex data structures. In the latter case, the description needs to include inter-node descriptions of the mapping. The definition of the work units permits a pre-partitioning into work chunks as well as basic data structures.

We support the following application types:

- Independent work units which do not communicate with each other and can easily be moved [10]. Load sharing is sufficient for such applications.
- Work units with restricted dependencies with direction from potentially moved unit (like tree structures). Load sharing is sufficient for such application as well.
- Computations with work and communication being separated and the dependencies being described in graph structures. Load balancing is required for such applications.

In the later case, computations can still be kept in synchrony for potential frequent communication. Though the mapping of the graph structures need to be updated if moving work units are done in all graph-based load balancing.

3. Zoltan

Real world applications can be represented as a graph, where the vertices of a graph represent the computation of application and the edge between two vertices represents the communication between them. In parallel applications, we distribute their processes among different processor so that they can compute in parallel. This kind of parallel application is also represented by partitioned graph, where each partition vertices represent the total computation of a process and the edge cuts or edges between two neighbor partitions represent the communication between two processes. While representing the application with graph partitions, we always like to keep the edge cuts minimized so that the communication between two processes remains minimized. During repartitioning we want to minimize the edge cuts for the same reason, and we also like to keep the new partition more likely to the old ones, so that less number of vertices would require migrating minimizing the migration cost. ATOP [11] used Zoltan for their partition and migration work. Our work is implemented as an extension of ATOP and we used Zoltan for the similar reason.

3.1 Zoltan Overview

Zoltan is a dynamic load balancing library with object oriented interface that allows user to use it with parallel application and call the various load balancing algorithm that comes with it [12]. It provides flexible data management services to parallel applications. Unstructured and adaptive parallel applications can use the following utilities:

- Dynamic load balancing and parallel partitioning tool that helps to distribute data over processors.
- Data migration tools.
- Distributed data directories.
- Unstructured communication package.
- Dynamic memory management package.

Zoltan has object based callback function design. Application can provide the required callback function that access the application data structure. Callback functions are registered in zoltan by passing a pointer to the function. The most interesting feature that comes with zoltan is that application don't have to be zoltan data structure dependent and it can be used almost every kind of operating system. Following are important zoltan query functions:

QUERY FUNCTIONS	RETURNED INFORMATION
ZOLTAN_NUM_OBJ_FN	Query function returns the number of objects that are currently assigned to the processor.
ZOLTAN_OBJ_LIST_FN	Objects list currently assigned to the processor
ZOLTAN_FIRST_OBJ_FN & ZOLTAN_NEXT_OBJ_FN	First object returns the global and local IDs of the first object on the processor and next returns the next object assigned to the processor.
ZOLTAN_PARTITION_MULTI_FN or ZOLTAN_PARTITION_FN	Returns a list of partitions to which given objects are currently assigned.
ZOLTAN_NUM_EDGES_MULTI_FN or 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_MULTI_FN or 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	To tell Zoltan how to copy all needed data for a given object into a communication buffer.
ZOLTAN_UNPACK_OBJ_FN	To tell Zoltan 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	To perform any pre-processing desired by application.
ZOLTAN_POST_MIGRATE_PP_FN	To perform any post-processing desired by application.

Table 1. Important query functions of Zoltan (Source: http://www.cs.sandia.gov/Zoltan/Zoltan.html*)*

ZOLTAN'S OPERATIONS	SEMANTICS OF OPERATION
Zoltan_Initialize	This function initializes MPI for Zoltan.
Zoltan_Create	This function allocates memory for storage of information to be used by Zoltan and sets the default values for the information.
Zoltan_Set_Param Zoltan_Set_Param_Vec	Modifies the values of any parameter used in Zoltan. Only one parameter can be changed in each time.
Zoltan_Set_Fn Zoltan_Set_<zoltan_fn_type>_Fn	It registers an application-supplied query function in the Zoltan structure.
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 and sets the structure to NULL in C.
Zoltan_LB _Set_Part_Sizes	specifies the desired relative partition sizes; equal by default; for some ParMetis algorithms, the partition size cannot be set as empty.
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 real 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. Basic Zoltan operations (Source: http://www.cs.sandia.gov/Zoltan/Zoltan.html*)*

3.2 Over-partitioning for Time Adaptation

In our approach we used over-partitioning which is described in [11]. In traditional partitioning strategy we create data partition equal to the number of processor allocated for an application or equal to the total process number of a job. But in over-partitioning we create more data partition than the number of processes so that processes are allowed to have more than one partition, so that during resource adaptation we can migrate required partition to other processes instead of migrating individual vertices reducing the repartition cost. For example, we can create 128 partitions for 8 processes. We directly used the existing over partitioning strategy [11] in our load balancing adaptation framework library. Partitions are delivered from Zoltan sequentially maintaining the neighbor relation of partitions so that edge cuts can be reduced after adaptation thereby minimizing the after adaptation inter-process communication. In our test cases, we always created 128 number of partitions for over-partition.

3.3 Partitioning from Scratch

Zoltan provides a variety of graph partitioning algorithm. We used one of the popular graph partitioning algorithm K-way graph partitioning algorithm [17] from Zoltan. In this algorithm, a graph is partitioned in three consecutive steps. (1) Graph coarsening phase (2) Initial partitioning and (3) Graph un-coarsening phase. In the graph coarsening phase, they coarsen the initial graph multiple times in order to get the possible coarsest graph, so that it is much easier and less expensive to partition the coarsest graph instead of partitioning the original graph. In the initial partitioning phase, the coarsest graph is fed into the Kernighan-Lin partitioning algorithm to get the initial partition. Once the partition is done, they un-coarsen the graph, which is the reverse process of the first phase. This is how they get back the original graph, but partitioned at the end. The following figure depict the three phases of k-way partition algorithm. During the coarsening phase, the initial graph is successively decreased and in the initial partitioning phase a 6-way partition is done here. At the end the graph is successively refined and projected back to the larger graph.

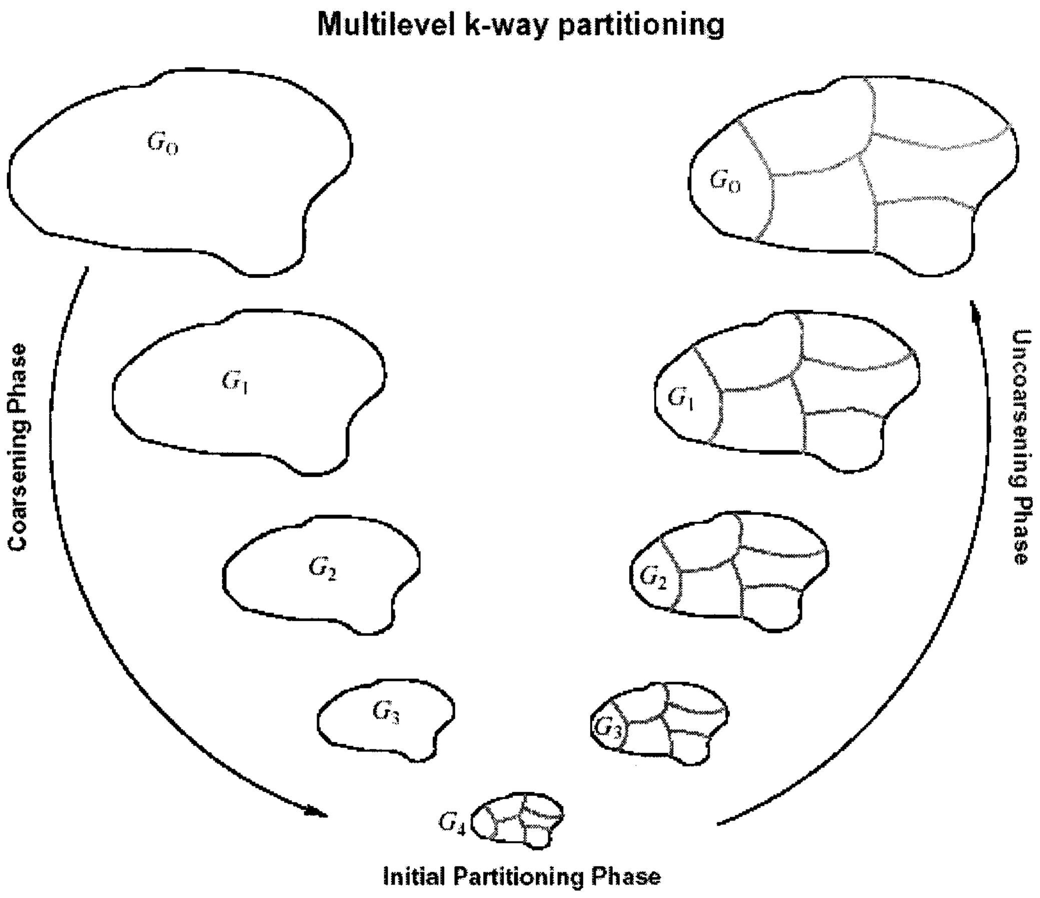

Figure 2. Phases of multilevel k-way partitioning algorithm. Source [16]

4. Our Approach

Load balancing adaptation can be initiated by following conditions:

- Job scheduler notices job completion or departure among co-scheduled jobs.
- Job scheduler notices new job start or initiation among co-scheduled jobs.
- Unknown resource usage due to the dynamism in program behavior.
- Inaccurate prediction noticed due to heterogeneous resources or slowdown affect.

Note that this load balancing initiation can be triggered by job scheduler (first two conditions) and workload monitoring information system (last two conditions). The adaptation always applies to groups of processes running on groups of processors which are groups of hardware nodes.

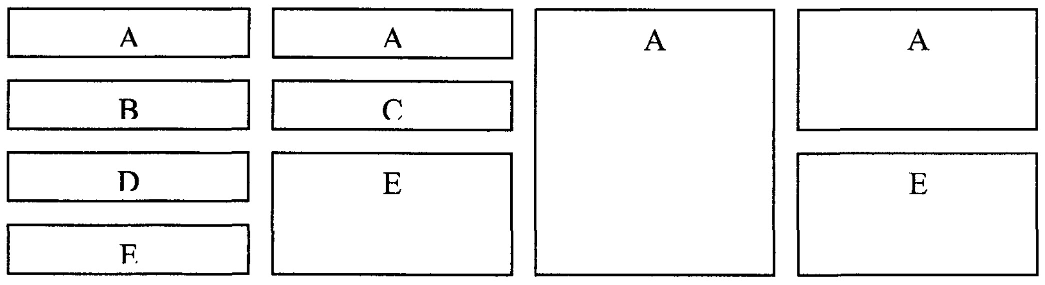

Figure 3. Different multiprogramming levels and different co-scheduled application for the application A and E on different subsets of nodes on a cluster [14].

4.1 The Dynamic Directory

The dynamic directory keeps information about all the scheduled and running jobs in the system. This version of dynamic directory is described in [8]. It stores the following updated informations:

- Owner or user
- Remote request yes/no, single site/ cross-site request
- Requested share and runtime estimate
- Communication pattern
- Communication frequency
- Memory, I/O and other requirements

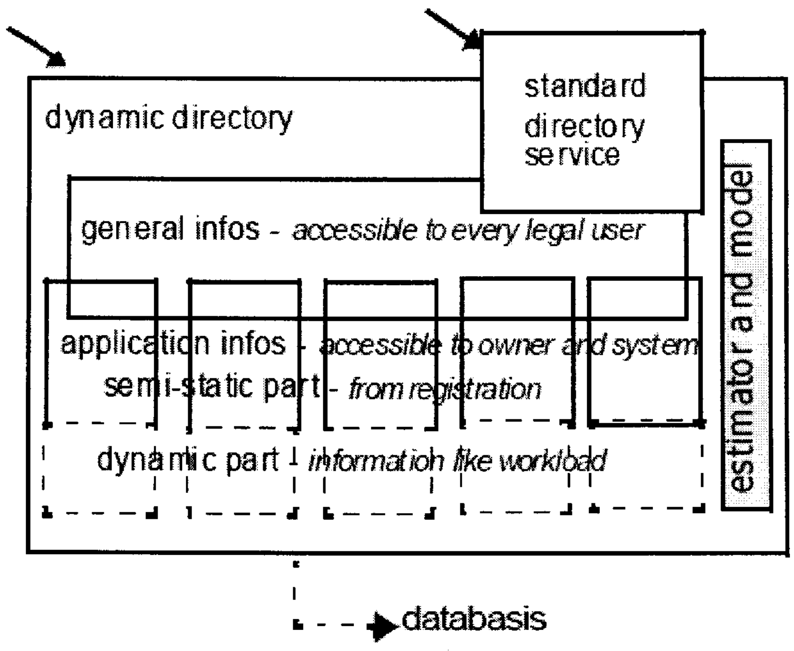
dynamic directory
standard directory service
general infos - accessible to every legal user
application infos - accessible to owner and system
semi-static part - from registration
dynamic part - information like workload
estimator and model
databasis

Figure 4. Dynamic directory (Source: [8])

This dynamic directory can keep information about running application and their workload on each process. We also assume that this dynamic directory maintains information per user (permitted resource usage, left over usage, maximum runtime, and performance information from previous runs). In our implementation concept, this dynamic directory links between the operating system scheduler and adaptation controller so that it is possible to combine the system and application information is the adaptation method. However for simplicity, instead of linking with operating system scheduler, we implemented our own simulated scheduler with a script provided that we know the sequences of job and their arrival interval. And our dynamic directory gets the information about scheduling of jobs through the adaptation library, not from the operating system library (though that is the original concept).

4.2 General overview of our approach

We assume that we have N nodes, that each node is uni-processor, and that the set of nodes that were assigned for a certain job is S_{new}. Work type can be ARRIVAL or COMPLETION. Assuming that we have M applications running on the system, we have M node sets S_j for application $j \in M$.

Algorithm to find out the affected jobs due to ARRIVAL or COMPLETION:

```
$for(j = 1; j \leq M; j++)$
$BEGIN$
$if(\neg((S_j \cap S_{new} = \phi) \vee (S_j \subseteq S_{new}) or (Adapt\_time \geq exec\_time\_left + \delta)))$
$Send\_to\_j(work\_type, S_j \cap S_{new})$
$END$
```

We assume that the required information for the possible adaptation is sent to all processes of an application. This information is sent through the socket communication to the master process and then the master process broadcast this information to the rest of processes of that application through MPI communicator, so that it is possible to make the adaptation cheaper. Usually job scheduler are independent of any communication system used and do not require a job to be malleable. In our case we have a load-adaptation controller per job that communicates and initiates any possible load-adaptation. But this load-adaptation controller is not directly any part of the job scheduler so that job scheduler remains independent of the adaptation work. But the job scheduler puts all the scheduling decision and terminating information to the dynamic directory. The load-adaptation controller can access updated information from the dynamic directory for possible adaptation. In our approach, we decoupled the job scheduler and dynamic directory, so that the job scheduler remains independent of application adaptation. Dynamic directory ensures the consistency of the system information by storing job information, machine information and updated resource allocation among scheduled jobs.

For resource based reasons for adaptation, we assume that we can measure progress of an application with a monitor. It is important to measure the progress of an application relating the processing of workload. We can do this either by relating to absolute time estimates and determining how much percentage the time estimation was wrong. Another way is to measure the relative progress, which is more feasible as this does not require exact estimate. Another important thing is to find out about how much percentage longer an application runs on one computer than another. The two possible ways are:

- Use a fixed time interval and determine the progress in workload processing
- Use a fixed amount of work and determine after which time this amount of workload is processed.

The latter can be done with simple time stamps and is easier to implement where as the former would require expensive timer interrupt. We assume that information is collected at the load-adaptation controller and scalability can be ensured by either collective communication or by using representative process from each co-scheduled group reporting to the load-adaptation controller.

This approach applies to all possible types of applications. We can set a certain number of work units as the fixed amount of work and then determine the time after which they are processed. In iterative processing, this could be one or multiple passes over all the local units.

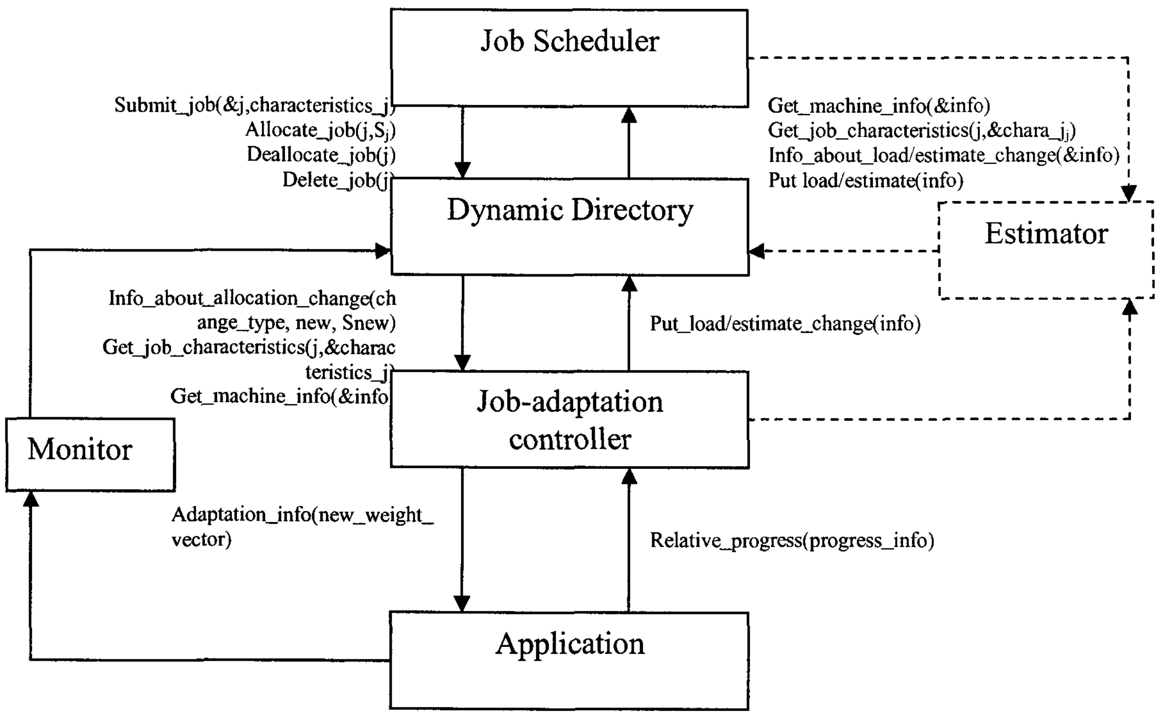
Job Scheduler
Submit_job(&j,characteristics_j)
Allocate_job(j,S_j)
Deallocate_job(j)
Delete_job(j)
Get_machine_info(&info)
Get_job_characteristics(j,&chara_j_j)
Info_about_load/estimate_change(&info)
Put load/estimate(info)
Dynamic Directory
Estimator
Info_about_allocation_change(change_type, new, Snew)
Get_job_characteristics(j,&characteristics_j)
Get_machine_info(&info)
Put_load/estimate_change(info)
Monitor
Job-adaptation controller
Adaptation_info(new_weight_vector)
Relative_progress(progress_info)
Application

Figure 5. Architecture of Our Approach

After each allocation of a new job or de-allocation of a completed job, changes are made to the dynamic directory and inform the job-adaptation controller. Then the job adaptation controller sends information to the representative processes of each affected job about the possible load adaptation through the communication socket. The representative process of a job broadcasts this information to the rest of the processes of that job, so that they can initiate load balancing after evaluating the feasibility and necessary migration calculation. Load information is updated to the job scheduler and dynamic directory by job adaptation controller after each adaptation.

Job runtime is estimated based on machine information and job characteristics at the beginning of job execution and this estimation is corrected by putting progress checkpoint on application process (after a fixed amount of work) and required time to execute that far. This progress is reported back to the each adaptation controller by its respective job. Depending on this progress report, adaptation controller may change the estimation (by estimator) and suggest for possible load balancing to the representative process of that job if (a) unequal relative progress (b) error in estimation is found.

Definition *Balance*: Nodes are balanced if they are allocated a relative workload that will be processed within the same amount of time. The relative workload is determined by the application's computational tasks, the machine's share and the machine's processing power.

Definition *Share*: Machine share s_i is the resources of a machine that are being used by an application. If an application takes 50% of resources of a machine, then the machine share of that application is 0.5.

Definition *Partition weight*: Partition weight (w_i) is the amount of task of an application that has been assigned to a processor. If we represent the application in a graph, then partition weight is the total weight of vertices of one partition.

Definition *Relative Progress*: Relative progress ($w_{procesed}/w_i$) can be defined as the proportion of total number of vertices that has been computed or processed ($w_{processed}$) in one partition of an application with the total partition weight (w_i).

Definition *Machine Weight Factor*: Machine weight factor (f_i) expresses the relative speed of one machine compared to the base machine (probably the slowest machine of the system).

In the case of adaptation, we calculate the new weight in the following way:

Calculation of new Partition weight: Let us assume that we have a weight vector $<w_1, w_2, w_3...w_m>$ and a share vector $<s_1, s_2, s_3...s_m>$ for an application running on m processors where w_i is the partition weight of ith partition and s_i is the machine share of the same application for that partition. If the total application weight is W and the total machine share is S, then we have

$$w_1 + w_2 + w_3 + \ldots + w_m = W$$

and

$$s_1 * f_1 + s_2 * f_2 + s_3 * f_3 + \ldots + s_m * f_m = S$$

Where f_i is the machine weight factor of machine i.

After adaptation the new share vector become < $s_1^{'}, s_2^{'}, s_3^{'} \ldots s_m^{'}$ > and the new total share becomes

$$s_1^{'} * f_1^{'} + s_2^{'} * f_2^{'} + s_3^{'} * f_3^{'} + \ldots + s_m^{'} * f_m^{'} = S^{'}$$

Then, the new partition weight becomes

$$w_i^{'} = w_i * (S / S^{'}) * (s_i^{'} / s_i) * (f_i^{'} / f_i)$$

and

$$w_1^{'} + w_2^{'} + w_3^{'} + \ldots + w_m^{'} = W$$

The new weight that we found is good for non Hyperthreaded (HT) processor. But in HT processor, two threads can logically execute concurrently, virtually doubling the processing power. Since we did our test in HT Xeon processors, we need to find the new HT weight of a processors. We used a factor ($1/f_{HT}$) to convert the new weight vector to the new HT weight vector.

t_A=dedicated execution time of an application A

n=number of processes running on a processor

t'_A=execution time of application A on a processor in time shared manner

$t_A'=n*t_A$ (in a not HT processor)

$t_A'=n*t_A*f_{HT}(A,B)$ (f_{HT} is the HT factor)

Then we have:

HT factored new weight $w_{HT}'=(1/f_{HT})*w'$

4.3 Implementation

In this section we are going to explain how we implemented different components like simulated job scheduler, controller and adaptation library in our load balancing adaptation framework.

4.3.1 Job Scheduler

For our experiment we needed to establish the coordination between the job scheduler and the dynamic directory. Instead of playing around with the job scheduler of our cluster, we created our simulated job scheduler with a shell script file where we put our preplanned sequence of jobs and their arrival delays. Following is an example of such simulated job scheduler.

```
/home/arefeen/test12/DD&
/home/arefeen/test12/controller0&
mpirun -np 4 -machinefile machine0 /home/arefeen/test12/application0&
sleep 150
/home/arefeen/test12/controller1&
mpirun -np 4 -machinefile machine1 /home/arefeen/test12/application1&
```

Figure 6. Example of a simulated job scheduler script

In this example, the first line schedules the dynamic directory and at the same time schedules controller0 and application0. Controller0 had direct socket communication channel with the dynamic directory and application0 has indirect communication with the scheduler through its controller (controller0). Then we wait 150 seconds and schedule application1 and controller for that. But while we execute our test with such simulated job scheduler script, we need to make sure that no other user is logged in and running their application on the cluster. Otherwise our test results might be incorrect.

4.3.2 Dynamic Directory

The basic Dynamic Directory idea is adopted from [8] and it is explained earlier in section 4.1. In our implementation we created our dynamic directory with a multi-threaded socket server. Each thread from different application through controller can operate on the dynamic directory data. Threads are synchronized with pthread_mutex. Pthread condition variable was used to signal the waiting thread after arrival or departure of an application (after creation or termination of a thread). Dynamic directory is connected to every adaptation controller through socket from different threads as long as the application continues to execute.

4.3.3 Controller

In our implementation of adaptation controller, we created adaptation controller per application instead of creating single adaptation controller for all applications. So that it is possible to keep our controller simpler. Each adaptation controller has two socket communications. One with the dynamic directory and other with the application. The controller is connected to the application with a listener thread from the representative process (usually process 0).

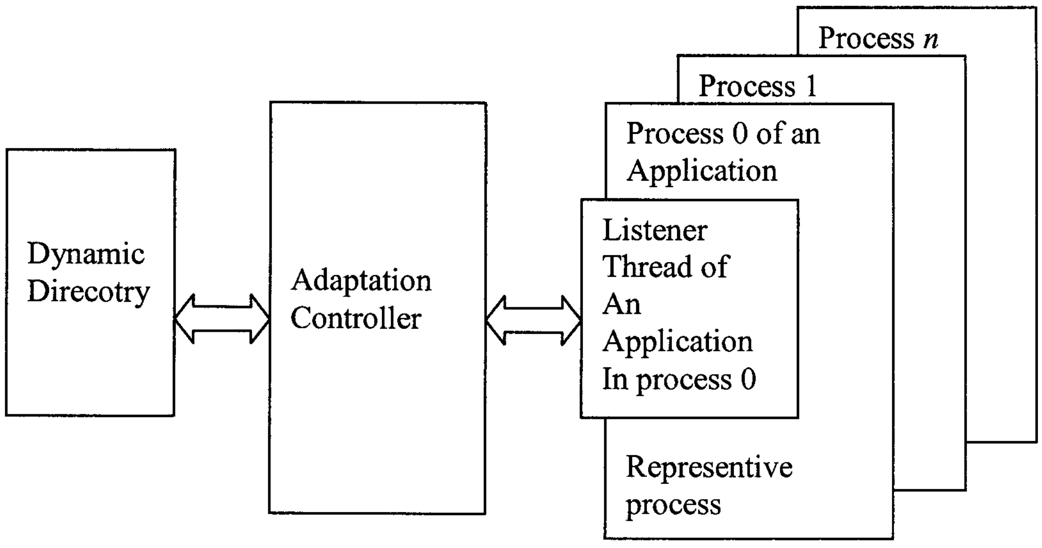

Figure 7. Adaptation controller communication with dynamic directory and listener thread of an application.

4.3.4 Adaptation Library

In our adaptation library we used the over-partitioning algorithm and migration from ATOP [11]. We did not use the space adaptation though it is equally important, we only focused on time adaptation. Unlike [11], we used dynamic weight vector for our application processes. We provided the following methods in our adaptation library to be used by MPI application.

Method	**Description**
set_policy( )	This method defines whether to use over partition or partition from scratch.
OP_init( )	This is to initialize the over partition at the beginning.
OP_adapt( )	Adaptation using overpartitioning and then migrate, depending on the new weight vector.
My_weight( )	This method returns the related updated weight for each process.
my_flag( )	Thie method returns the Boolean value whether to adapt or not, depending on whether or not the listener thread received any new weight vector for the application processes.
ZP_adapt( )	This method does the adaptation using the zoltan partition (partitioning from scratch) and then migrate.
set_communicator( )	This method copy the application communicator to the library and also creates the listener thread at the beginning of application.
Register_Environment( )	This method register the query functions (both partition and migration) for Zoltan and sets the parameters
sys_finalize( )	This method finalize the zoltan and terminates the communication with controller and thereby dynamic directory.

Table 3. Methods of our Adaptation library

Following is a general example of a parallel application using our adaptation library

```
Set_communicator()
Register_Environment()
While (iteration<limit)//iteration for the application
{
        //Computation for parallel application
        compute()
        ...
        //Communication for parallel application
        ...
        send()
        receive()
        ...
        if ((iteration % syn_num)==0) //syn_number if the frequency to check for adaptation
        {
                try_adapt()
        }
}
sys_finalize()
```

Figure 8. Example of application using our adaptation library

4.4 Adaptation Cost Model

The cost of adaptation can be split into two parts. One is adaptation cost of application C_{app} which includes the partition and migration cost and the other one is system overhead C_{sys} that includes the monitoring cost and load information acquiring cost.

Our framework support both zoltan partition and over partition. Application programmer can decide which method to use. For zoltan partition:

C_{init}=Initiate the adaptation by the representative process including the broadcast to all processes, communication with the adaptation-controller

C_{zoltan_mig}= Migration cost for zoltan

C_{zoltan_par}= Partition/Repartition cost for zoltan

C_{over_mig}= Migration cost with overpartition

C_{over_par}= Partition/Repartition cost with overpartition

$C_{adapt\_app\_zoltan}$=Adaptation cost for the application using zoltan partition and migration

$C_{adapt\_app\_over}$=Adaptation cost for the application using over partition and migration

$C_{adapt\_app\_zoltan} = C_{init} + C_{zoltan\_par} + C_{zoltan\_mig}$

$C_{adapt\_app\_over} = C_{init} + C_{over\_par} + C_{over\_mig}$

The cost model in [11] explains that it is suitable to use over partitioning when time saved due to the overpatition is greater than the application communication cost caused by more edge cuts after adaptation. In our framework the application programmer has the freedom to choose from $C_{adapt\_app\_zoltan}$ and $C_{adapt\_app\_over}$ from the library, so that the application can be adapted more efficiently.

$C_{app} = MIN(C_{adapt\_app\_zoltan}, C_{adapt\_app\_over})$

The system cost C_{sys} will include the communication between the scheduler and dynamic directory and the communication between dynamic directory and adaptation controller per job. So the total adaptation cost will be:

$C_{total} = C_{sys} + C_{app}$; which will be dominated by the application C_{app} part.

X=Speed up due to adaptation

T=time to complete execution without adaptation

$T_{comp} = MAX(T_{comp,i}: i <= m)$, remaining computation time without adaptation that takes to process all the vertices of a partition. Note that this T_{comp} will be determined by the longest computation time (of *ith* partition) of *m* processes where *m* is the total number of processes of a job.

$T_{comm} = \sum E_{cuts}$, communication time without adaptation, which is the total edge cuts with neighboring partitions.

Thus we have,

$T = T_{comp} + T_{comm}$

T'=time to complete execution with adaptation

$T'_{comp} = MAX(T'_{comp,i}\text{: } i <= m)$, remaining computation time after adaptation that takes to process all the vertices of a partition. T'_{comp} will also be determined by the longest computation time (of *ith* partition) of *m* processes where *m* is the total number of processes of a job.

$T'_{comm} = \sum E'_{cuts}$,communication time after adaptation, which is the total edge cuts with neighboring partitions. E'_{cuts} is the new edge cuts after adaptation (partition and migration)

Then, $T' = T'_{comp} + T'_{comm}$ and the speed up due to the adaptation will be $X = T - T'$.

The adaptation will be meaningful if the X,speed up due to adaptation becomes greater than C_{total}, the total adaptation cost. We found the speed up time in second for X, and we can also present this speed up in percentage like below:

Percentage of speed up= $(T-T')/T*100\%$

4.5 Test Plan

4.5.1 Test Environment

We are going to execute our tests on our AlphaMeta lab's Hourus IBM cluster. This cluster has 16 nodes (enode1-enode16), each containing dual Intel Xeon processor with 512 Mbytes of memory. The first 14 nodes (enode1-enode14) have CPU speed of 2.0 GHz and the last two nodes have CPU with 2.4 GHz. This provides us somewhat heterogeneous test platform. The frontend node (emaster) has 4 Intel Pentium III Xeon processor with 700 MHz speed with 1Mbyte L2 cache. All these nodes are connected through Myrinet interconnect. Operating system running on all nodes is Debian Linux with kernel version 2.6.6 and we used MPICH-GM 1.2.5.12 (an implementation of MPI over GM) over GM (low level message passing system for Myrinet network). Our framework used MPICH-GM 1.2.5.12, Zoltan 1.52 and ParMETIS 3.1.

4.5.2 Test Application

Real word applications are represented with graph while the vertices represent the computation of an application and the edges represents the communication between two vertices. In our case, we represented an application with benchmark graphs from the University of Greenwich Graph Partitioning Archive [13]. This graph archive was used in [11]. We used the Chaco file input format, where the first line contains the integer value of total vertices or nodes *N* and total number of edges *E*. Then the following N lines represent the neighborhood relation of corresponding vertex. An example is given below:

```
6 9
2 6
1 3 4
2 4 6
1 2 3 5
4 6
1 5
```

Figure 9. Chaco graph input file format

In this file there are 6 vertices and 9 edges, where vertex 1 is adjacent to 2 and 6, vertex 2 is adjacent to 1, 3 and 4 and so on. We represented an application in a reverse direction. We first selected the graph and created our application based on the graph pattern. We used the following graph for our applications.

Graph	**Total number of Vertices in graph**	**Total number of Edges in graph**	**Description**
t60k	60005	89440	Not available
wing	62032	121544	3D finite element mesh
brack2	62631	366559	3D finite element mesh

Table 4. Different Graph for our test applications

We selected these 3 graphs due to their differences in their edge number. All of the graphs have almost same number of vertices but they have different number of edges. This graph constitutes the application skeleton. This represents the applications with different computation and communication pattern.

For our test purpose we used these graphs as the basic skeleton of our application that defines the computation and communication pattern. Each process of our parallel application has a boundary array (updated_value[]) that contains the data of the adjacent nodes that are located in the neighbor processes. For example in the following figure, process 1 (partition 1) has a boundary array that keeps the data of the adjacent nodes that are in partition 2, 3, 4 and 5. In every iteration it does the computation, then sends the computed updated value of that boundary array to the adjacent processes and then receive the updated values from the adjacent processes (partitions). This sequence of computation and communication iterates until the maximum iteration limit is reached. At the end of certain number of iterations, the application synchronizes and checks if the controller advised for adaptation. The computation in our application is pretty simple; each node calculates the average value of all the adjacent nodes including its own value. At the end of this computation, processes exchange the values in the boundary array with the neighbor partitions that are other processes. Here we used the word process and partition in the sense that a process represents one or more partitions. We also presented our jobs with certain percentage of computation and the remaining percentage of communication. Application iteration number is 90000 except the test case where we expand and shrink the overlapped processes. So we can get the computation time of each iteration.

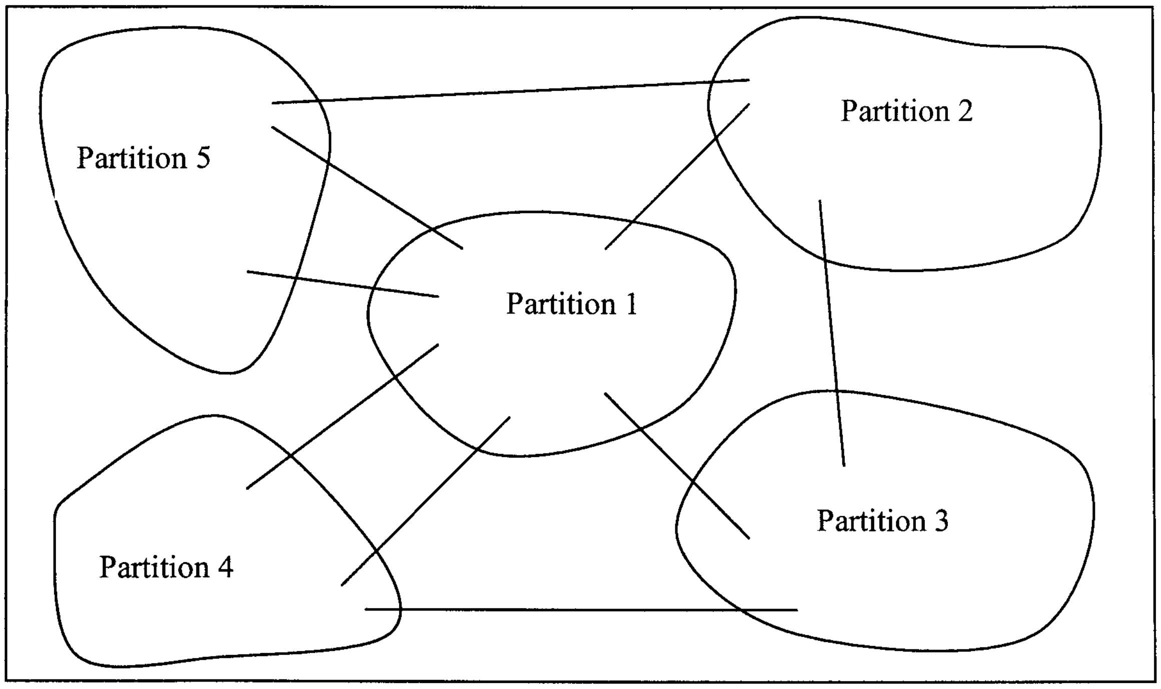

Figure 10. Example of application structure.

Our test application does not represent all kinds of parallel application. We wanted to test our time adaptation framework with a test application, which is built on graphs taken from real applications. In our test application, we assumed that application can or able to adapt after completion of an iteration. But in reality, this might not be possible for complex structured application, where iteration is not just a sequence of computation and communication. One important limitation of our test application is that it takes certain amount of time for start-up. Since our application is built on graphs, we need to partition the graph and distribute over the processes of an application. Once the graph or partition of graph is distributed over processes, application can start working. In our cases, this start-up time takes 200 (on dedicated processors) to 400 (on time shared processor) seconds. In our application we also used some dummy block of computation that computes on double data type so that we can change the granularity of our application.

4.5.3 Test Cases

- For all sets of application we will take the application runtime in the dedicated set of processors (while no other application is running on these processors), application runtime in timeshared environment without adaptation while the processes of an application is overlapped with another applications processes, and application runtime in timeshared environment with adaptation using our load balancing adaptation framework (using both over-partition and partition from scratch) providing the similar kind of timesharing overlap. Finally we compare all these three different kind of runtimes. This test is to evaluate the performance of our adaptation framework, and to get the slowdown factor for this particular pair of application.

- Compare the speed up of our adaptation framework while imposing different overlap of application. The following figure explains this test. We have application1 scheduled at time t2 on set of processors 1 to 8. Application2 gets scheduled at time t3 on set of processors 4 to 11. In this test, we will compare the different speed up of an application using our adaptation framework, changing the overlapped set of processors (in this example processor 5-8) while the application space does not change over time.

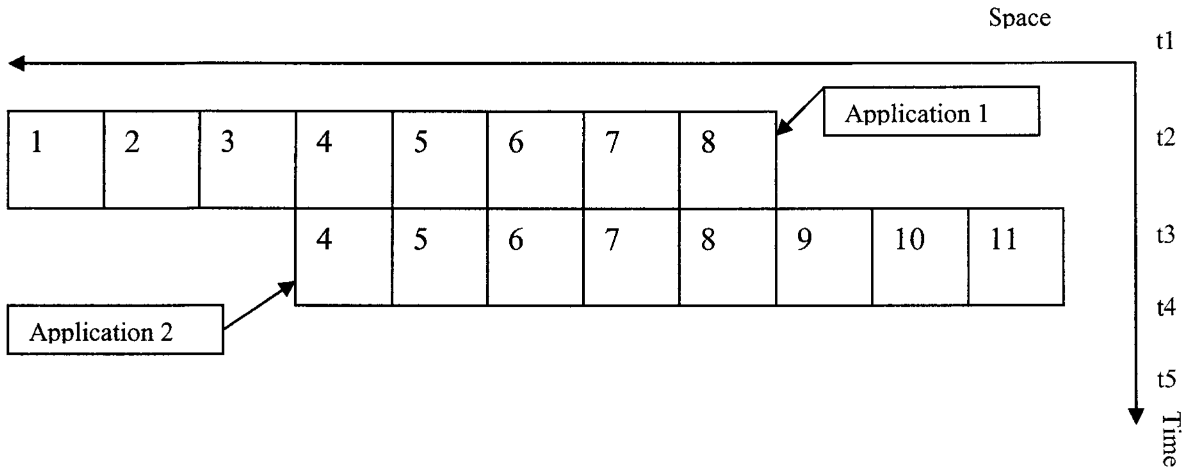

Figure 11. Test case 2, adaptation in different number of processor

In this test, we will change the number of processes overlapped with another application while keeping the application processes constant. By changing the overlapped set of processor we can evaluate the performance of our adaptation framework in different sets of under-loaded and over-loaded of processes.

- Unlike the previous test, in this test, we will change the number of processes (0, 4, 8, 16) of one application in different test run (not during the application run). This test is to evaluate the scalability of our adaptation framework. Here we will change the application process numbers and their overlapped processes number as well. Finally we will compare the different speed up with different process number of an application.

4.6 Test Results

In order to examine the performance of our adaptation frame work, we present the dedicated execution time of application of different granularity and we present the adapted execution time (both using over-partition and partition from scratch). While calculating the HT factor for weight vector as mentioned at the end of section 4.2, we collected the dedicated execution time of certain number of iteration and then shared execution time of same number of iteration. But in some cases we found that HT factor is little different for each of the application of application pair. In those cases we simply used the average of the two HT factor value for that application pair. The following table presents this execution time for application wing with 65 % computation and t60k with 85% computation. All these times are taken by using MPI_Wtime(), which return time in seconds. Both applications are run on 8 processors and adaptations are done while imposing 4 processes overlapped. Here wing adapts at the arrival of t60k and again at the termination of t60k. t60k starts execution with adapted weight vector, but does not adapt during runtime. All execution or run times are calculated after the startup time of application, i.e. excluding the start-up time.

Application	Dedicated Runtime (sec)	Runtime Without Adaptation (sec)	Adapted Runtime with Over-partition (sec)	Speed up (OP)	Adapted Runtime with partition from scratch (sec)	Speed up (ZP)
Wing, 8P, 65% computation	822.6	1067.0	982.8	7.89% 84.2 sec	1005.9	5.73% 61.1 sec
T60k, 8P, 85% computation	385.4	635.0	593.0	6.61% 42.0 sec	598.4	5.76% 36.5 sec
f_{HT}=1.17			OP_Adapt cost1=1.004		ZP_Adapt cost1=13.194	

Table 5. Adapted runtime Vs non-Adapted runtime, for wing and t60k, both with 8 processes and 4 processes overlapped

When we plot the gain achieved by adaptation using both over-partition (OP) and partition from scratch (ZP), we find the following figure. It is clear here that the adaptation is more expensive using partition from scratch than using over-partition. However, in this particular case, we used our HT factor f_{HT}=1.17. Here the second application (t60k) started 240 seconds after the start of wing. We could not have started t60k first, because in that case, by the time wing started after completion of its start up

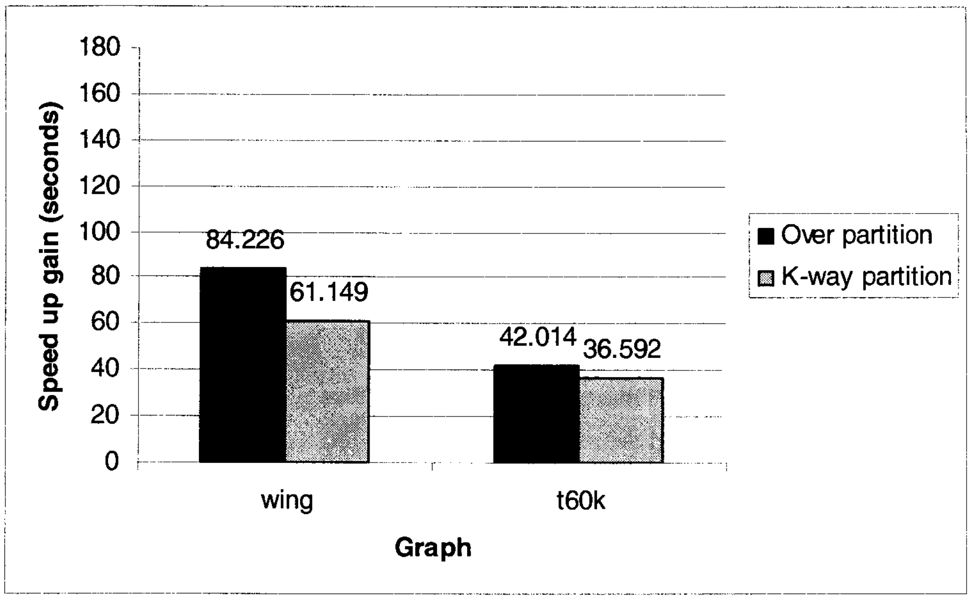

Figure 12. Comparison of adaptation gain using Over-partition and partition from scratch (wing 65% comp and t60k 85% comp).

time, t60k would have terminated. The adaptation cost using over partition is more than a second, while the speed up gained is 84.2 seconds more for wing and 42 seconds more for t60k. We noticed here that the speed up gained using both methods for t60k is almost same, because the second application did not adapt during its execution. It started with its adapted weight vector and completed before the completion of wing.

The following table presents the adapted execution time for application brack2 with 91% computation and application t60k with 85% computation. Both applications are run on 8 processors with 4 processor overlapped. The arrival delay between these two applications is 252 seconds, when brack2 starts first. Brack2 adapts at the arrival of t60k and again at the termination of t60k.

Application	Dedicated Runtime	Runtime Without adaptation	Adapted Runtime with Over-partition	Speed up (OP)	Adapted Runtime with partition from scratch	Speed up (ZP)
Brack2, 8P, 91% computation	608.4	987.4	822.7	16.7% 164.6 sec	851.7	13.7% 135.7 sec
T60k, 8P, 85% computation	384.6	612.9	503.0	17.9% 109.8 sec	522.5	14.7% 90.3 sec
$1/f_{HT}=1.15$			OP_Adapt cost= 1.178		ZP_Adapt cost=14.558	

Table 6. Adapted runtime Vs non-Adapted runtime, for brack2 and t60k, both with 8 processes and 4 processes overlapped

When we plot this execution time, we get the following speed up gain for the two applications using over partition (OP) and partition from scratch (ZP). For this particular case, over partition cost is about 12 times less expensive than partition from scratch. For the very same reason like previous case, we scheduled t60k after 252 seconds of brack2 schedule time. The HT factor that we used here for overlapped processes is $1/f_{HT}=1.15$.

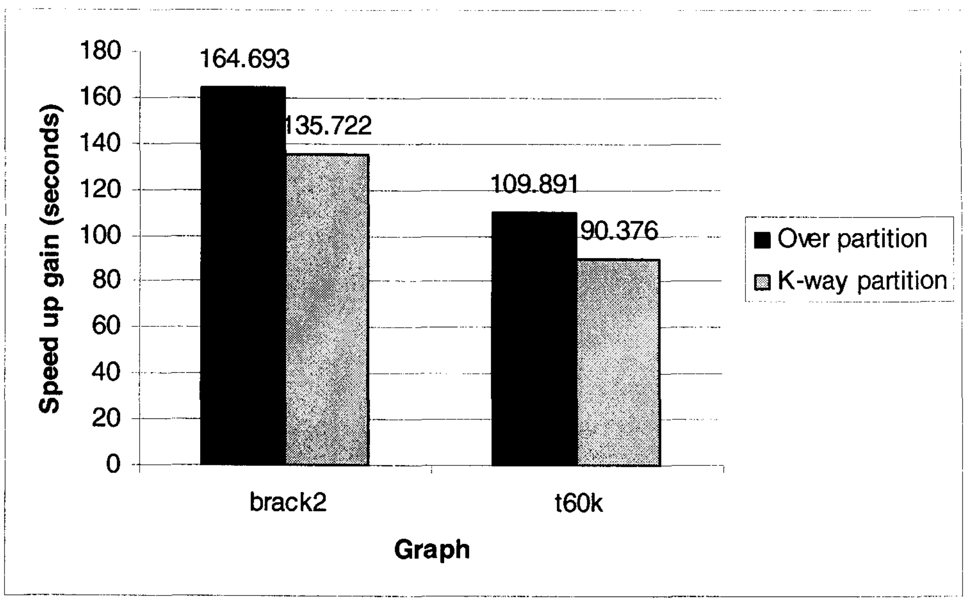

Figure 13. Comparison of adaptation gain using Over-partition and partition from scratch (brack2 91% comp and t60k 85% comp).

The following table presents the adapted execution time for application brack2 with 91% computation and application wing with 65% computation. Both applications are run on 8 processors with 4 processor overlapped. The arrival delay between these two applications is 252 seconds, when brack2 starts first. brack2 adapts at the arrival of wing and again at the termination of wing.

Application	Dedicated Runtime	Runtime Without adaptation	Adapted Runtime with Over-partition	Speed up (OP)	Adapted Runtime with partition from scratch	Speed up (ZP)
brack2, 8P, 91% computation	609.7	971.7	898.3	7.5% 73.3 sec	926.019	4.6% 45.6 sec
wing, 8P, 65% computation	831.8	1052.3	949.0	9.8% 103.2 sec	981.163	6.7% 71.1 sec
$1/f_{HT}=1.1$			OP_Adapt cost1=1.0823		ZP_Adapt cost1= 16.762	

Table 7. Adapted runtime Vs non-Adapted runtime, for brack2 and wing, both with 8 processes and 4 processes overlapped

We get the following figure after plotting these execution times and we found the adaptation using partition from scratch is more expensive than the adaptation using over partition.

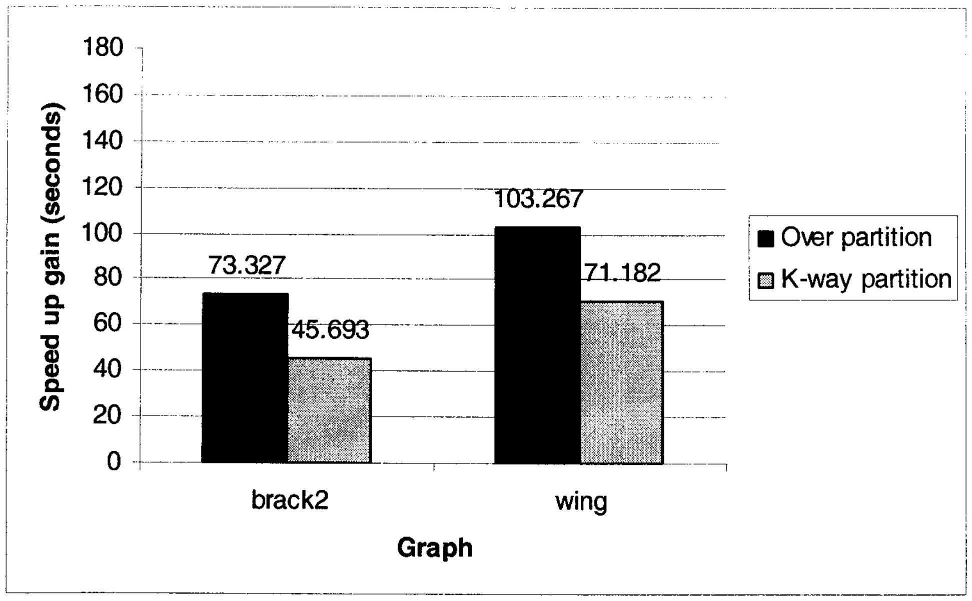

Figure 14. Comparison of adaptation gain using Over-partition and partition from scratch (brack2 91% comp and wing 65% comp).

It is noticed that for all three tests, we found partition from scratch is more expensive than over partition. This may not be true for every kind of application. The three test graphs that we considered indicate that adaptation using partition from scratch is more expensive than the adaptation using over-partition. However, these test results indicate that adaptation is meaningful, as the adaptation gain is greater than the adaptation cost. But if the application runs for a very short period of time, then adaptation might not be meaningful. We found that computation intensive applications can be benefited more than communication intensive applications.

In the following test we vary the overlapped processes (0,2,4,8). Both of the applications we used have 8 processes each. Note that there is no adaptation when the overlapped process is either 0 or 8, because in those situations there is no relative change of share among the processes of an application. In other situations (overlap 2 & 4) wing adapts at the arrival of t60k and t60k adapts at the termination of wing. The following results are found when we perform this test between the application with wing (with 65% computation) and t60k (with 85% computation). The HT factor we found is $1/f_{HT}=1.17$.

Application 1	Application 2	Overlapped process number	[1]T1 NA	[2]T2 NA	[3]T1 Adapted	[4]T2 Adapted	[7]Speed up 1	[8]Speed up 2	[5]C1	[6]C2
Wing 8 processes (65%)	T60k 8 processes (85%) arrival delay 240 sec	8	1283.7	715.7	1283.7	715.7	0.0%	0.0%	0.0	0.0
		4	1067.0	635.0	982.8	593.0	7.9%	6.6%	1.056	0.538
		2	955.1	582.4	876.8	513.5	8.2%	11.8%	0.449	0.469
		0	822.6	385.4	822.6	385.4	0.0%	0.0%	0.0	0.0

Table 8. Adapted runtime with varying overlapped processes (wing 65% comp and t60k 85% computation)

1. execution time (seconds) of application 1 without adaptation.
2. execution time (seconds) of application 2 without adaptation.
3. adapted execution time (seconds) of application 1 using over partition.
4. adapted execution time (seconds) of application 2 using over partition.
5. adaptation cost (seconds) incurred in application 1.
6. adaptation cost (seconds) incurred in application 2.
7. speed up achieved by application 1 after adaptation using over-partition.
8. speed up achieved by application 2 after adaptation using over-partition.

We found the figure 15 and 16, when we plot the adapted execution time with varying overlapped processes and non-adapted execution time with the varying overlapped processes. It is found that adaptation cost of wing for 4 processes overlapped is about 2.35 times than that of 2 processes, i.e. adaptation cost is less for less number of overlapped processes. However, for both of the application in this test achieved significant speed up by adaptation. Though it is not quite clear why the following two graphs have different shape. This could be due to the difference in their granularity.

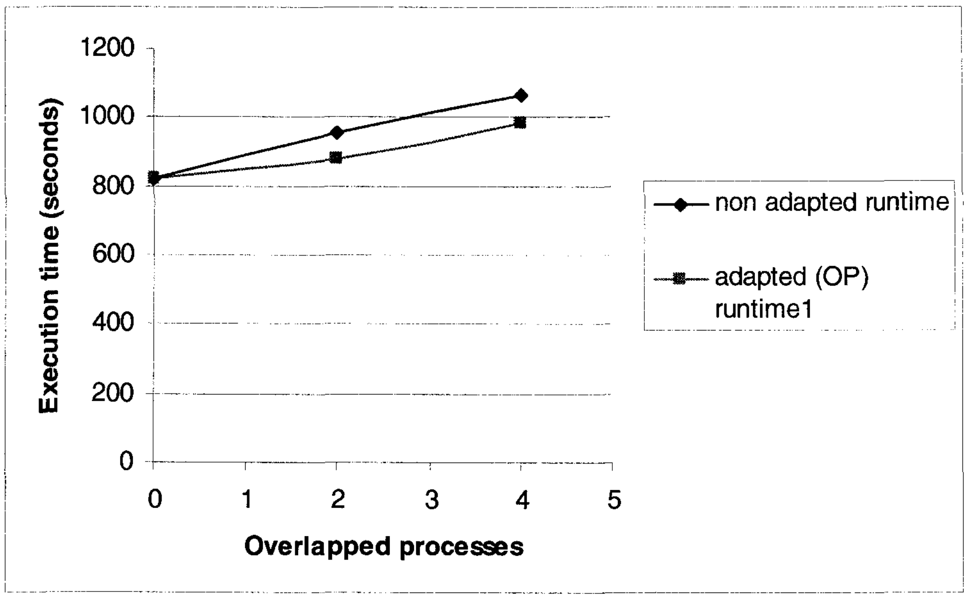

Figure 15. Adapted (OP) runtime Vs non-adapted runtime for wing (65% computation) while varying number of overlapped processes with t60k (85% computation)

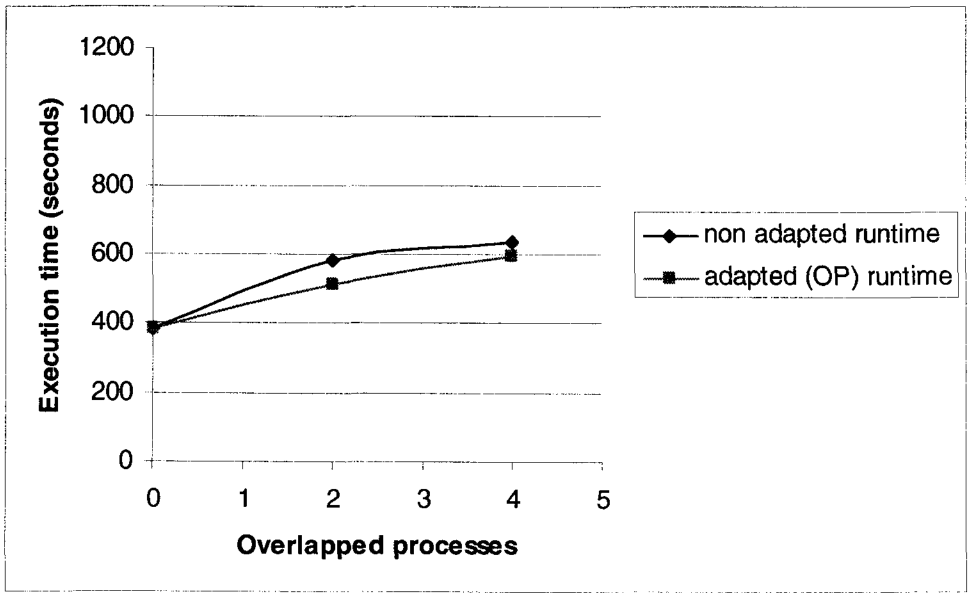

Figure 16. Adapted (OP) runtime Vs non-adapted runtime for t60k (85% computation) while varying number of overlapped processes with wing(65% computation)

The following results are found when we perform this test between the application with brack2 (with 91% computation) and t60k (with 85% computation) while varying the overlapped number of processes. The HT factor we found is $1/f_{HT}$=1.15 and total process of both of the application are 8. In 4 overlapped processes case, brack2 adapts at the arrival and completion of t60k. But in 2 overlapped processes case, brack2 adapts at the arrival of t60k and t60k adapts at the completion of brack2.

Application 1	Application 2	Overlapped process number	^{1}T1 NA	^{2}T2 NA	^{3}T1 Adapted	^{4}T2 Adapted	^{7}Speed up 1	^{8}Speed up 2	^{5}C1	^{6}C2
Brack2 8 processes (91%)	T60k 8 processes (85%) arrival delay 252 sec	8	1048.6	691.6	1048.6	691.6	0.0%	0.0%	0.0	0.0
		4	987.4	612.9	822.7	503.0	16.7%	17.8%	1.821 0.627	
		2	879.5	569.9	738.6	457.2	16.0%	19.6%	0.434	0.449
		0	608.4	384.6	608.4	384.6	0.0%	0.0%	0.0	0.0

Table 9. Adapted runtime with varying overlapped processes (brack2 91% comp and t60k 85% computation)

We found the following two graphs for the two applications. It is noticeable that the adaptation cost for 4 overlapped processes is about 4.19 times expensive that that of 2 overlapped processes.

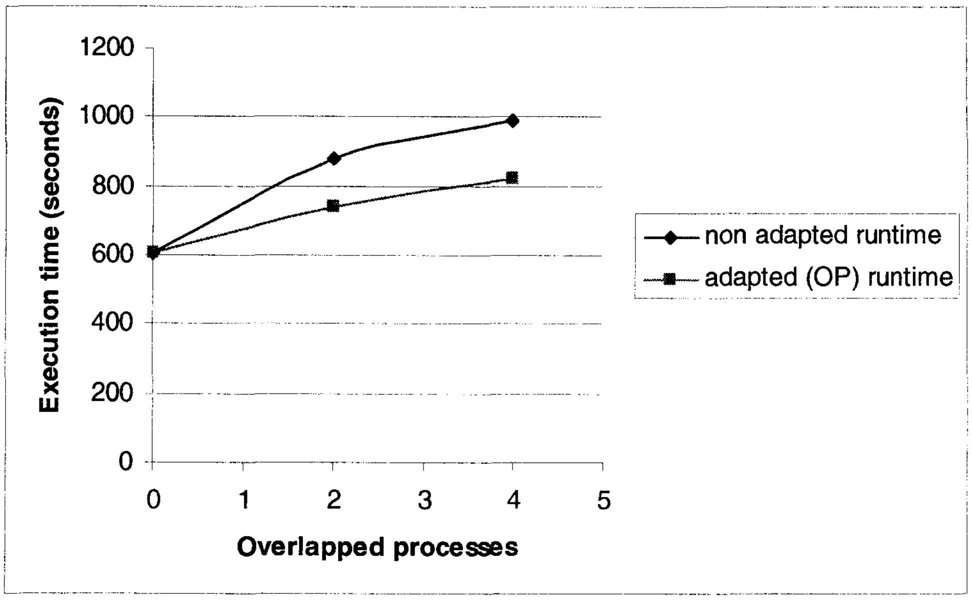

Figure 17. Adapted (OP) runtime Vs non-adapted runtime for brack2 (91% computation) while varying number of overlapped processes with t60k (85% computation)

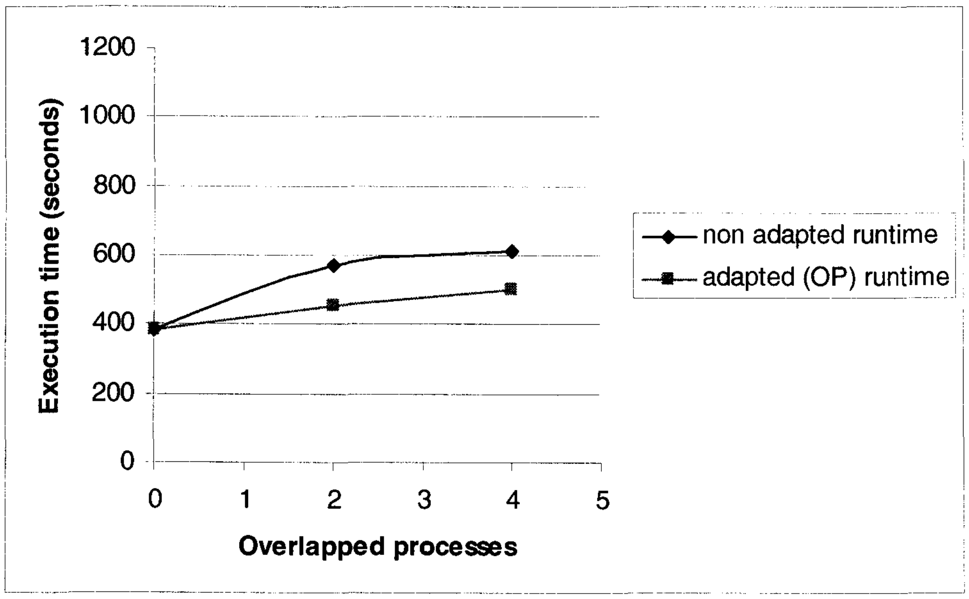

Figure 18

Figure 19. Adapted (OP) runtime Vs non-adapted runtime for t60k (85% computation) while varying number of overlapped processes with brack2 (91% computation)

In the following table we present the test result from brack2 (91% computation) and wing (65% computation) while varying the number of processes overlapped (0,2,4,8). Each application has 8 processes. The HT factor that we found for these two applications is $1/f_{HT}=1.1$. In 4 processes overlap case, brack2 adapts during arrival and departure of wing, but in 2 processes overlap case, brack2 adapts during the arrival of wing and wing adapts during the termination of brack2.

Application 1	Application 2	Overlapped process number	[1]T1 NA	[2]T2 NA	[3]T1 Adapted	[4]T2 Adapted	[7]Speed up 1	[8]Speed up 2	[5]C1	[6]C2
Brack2 8 processes (91%)	wing 8 processes (65%) arrival delay 252 sec	8	1141.8	1190.5	1141.8	1190.5	0.0%	0.0%	0.0	0.0
		4	971.7	1052.3	898.3	949.0	7.5%	9.8%	1.561 0.622	
		2	862.3	985.7	802.1	898.7	7.0%	8.8%	0.947	0.482
		0	609.7	831.8	609.7	831.8	0.0%	0.0%	0.0	0.0

Table 10. Adapted runtime with varying overlapped processes (brack2 91% computation and wing 65% comp)

We found the following two graphs for our applications. Both of them show significant speed up due to time adaptation.

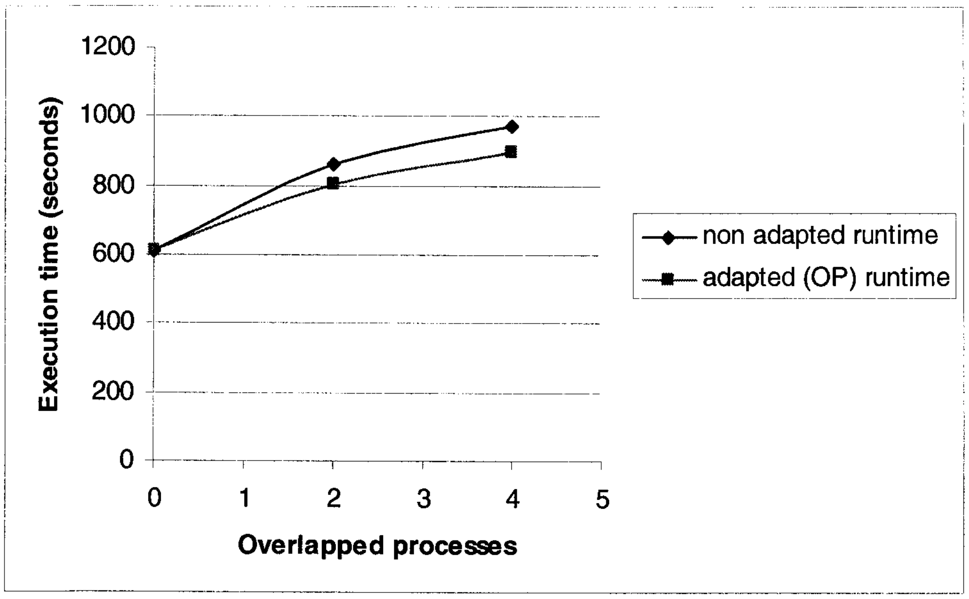

Figure 19. Adapted (OP) runtime Vs non-adapted runtime for brack2 (91% computation) while varying number of overlapped processes with wing (65% computation)

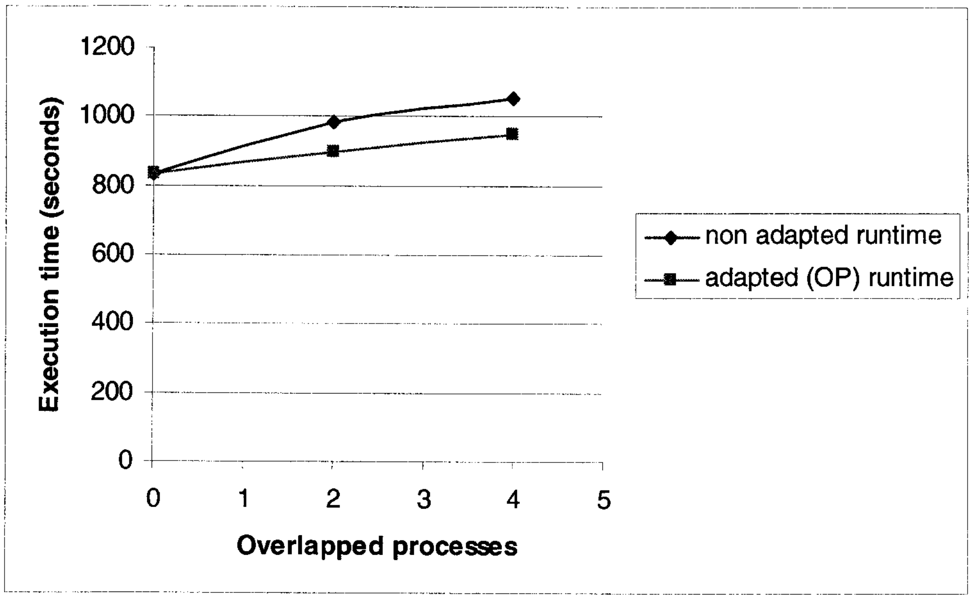

Figure 20. Adapted (OP) runtime Vs non-adapted runtime for wing (65% computation) while varying number of overlapped processes with brack2 (91% computation)

In these entire previous tests we used application with 8 processes. So that we can observe the adaptation affect on both of the application. We could have done similar kind of test for 16 processes, if we had a test environment with 32 processors. But we perform similar kind of test on 16 process application imposing 0, 2, 4,8 and 16 overlap with 0, 2, 4, 8 and 16 process application. But in this case, the second application is not going to adapt, only the first application with 16 processes can adapt while the overlapped processes are 2, 4 and 8. The first application does not adapt with 0 or 16 processes overlapped, as the relative resource share remain unchanged among the processes of an application. We get the following test results, which are the adapted execution time of our 16 process test applications, while having different overlap with different number of application processes. Wing arrives after 240 seconds of the arrival of t60k. T60k adapts at the arrival of wing and wing adapts at the termination of t60k.

Application 1	Application 2 ([14]init=451) shared	overlap	[9]T1 NA	[10]T1 Adapt	[11]T2 Adapt	[12]C1	[13]Speed up 1
t60k 16 processes (70%)	Wing 16p	16	533.0	533.0	794.5	0.0	0.0%
	Wing 8p	8	479.9	468.3	826.1	5.4	2.2%
	Wing 4p	4	470.6	451.0	1353.0	4.7	4.0%
	Wing 2p	2	418.6	397.3	2257.5	6.1	5.0%
	0p	0	370.1	370.1	0.0	0.0	0.0%

Table 11. Adapted runtime of t60k (70% computation) with Varying overlapped processes with wing (65% computation)

9. execution time (seconds) of application 1 without adaptation.
10. execution time (seconds) of application 1 with adaptation.
11. execution time of (seconds) application 2 with adaptation.
12. adaptation cost (seconds) of application1.
13. speed up achieved by application 1 due to adaptation.
14. initial start up time (seconds) of application 2 using over-partition with 8 processes.

In the following test, t60k arrives after 252 seconds of the arrival of wing. Wing adapts during the arrival of t60k and t60k adapts at the termination of wing. However it is found that the speed up increases with the less number of overlapped processes. The second application (t60k) does not adapt, because it is entirely contained within the processor set of first application (wing). That is why we only focused on the speed up of the first application that adapts.

Applicaiton1	Application2 ([14]init=409) shared	Overlap	[9]T1 NA	[10]T1 TA	[11]T2 TA	[12]C1	[13]Speed up 1
wing 16 processes (50%)	t60k 16p	16	1080.8	1080.8	659.2	0.0	0.0
	t60k 8p	8	931.2	897.7	536.5	3.229	3.6%
	t60k 4p	4	877.4	836.6	750.9	2.746	4.7%
	t60k 2p	2	803.5	761.3	1373.8	3.889	5.2%
	0p	0	695.4	695.4	0.0	0.0	0.0

Table 12. Adapted runtime of wing (50% computation) with Varying overlapped processes with t60k (85% computation)

In the following test, t60k arrives after 252 seconds of the arrival of brack2. brack2 adapts during both arrival and termination of t60k. It is important that the speed up brack2 is relatively less than that of wing, while both wing and brack2 had similar kind of overlapped processes with t60k. This is due the fact that brack2 is less computational intensive than wing.

Applicaiton1	Application2 ([14]init=409) shared	Overlap	[9]T1 NA	[10]T1 TA	[11]T2 TA	[12]C1	[13]Speed up 1
brack2 16 processes (40%)	t60k 16p	16	2995.2	2995.2	566.8	0.0	0.0%
	t60k 8p(82%)	8	2296.2	2248.8	621.7	4.1611	2.1%
	t60k 4p(86%)	4	2185.8	2105.1	1151.6	3.8716	3.7%
	t60k 2p	2	1783.6	1698.0	2194.0	4.5273	4.8%
	0p	0	1469. 9	1469. 9	0.0	0.0	0.0%

Table 13. Adapted runtime of brack2 (40% computation) with Varying overlapped processes with t60k (85% computation)

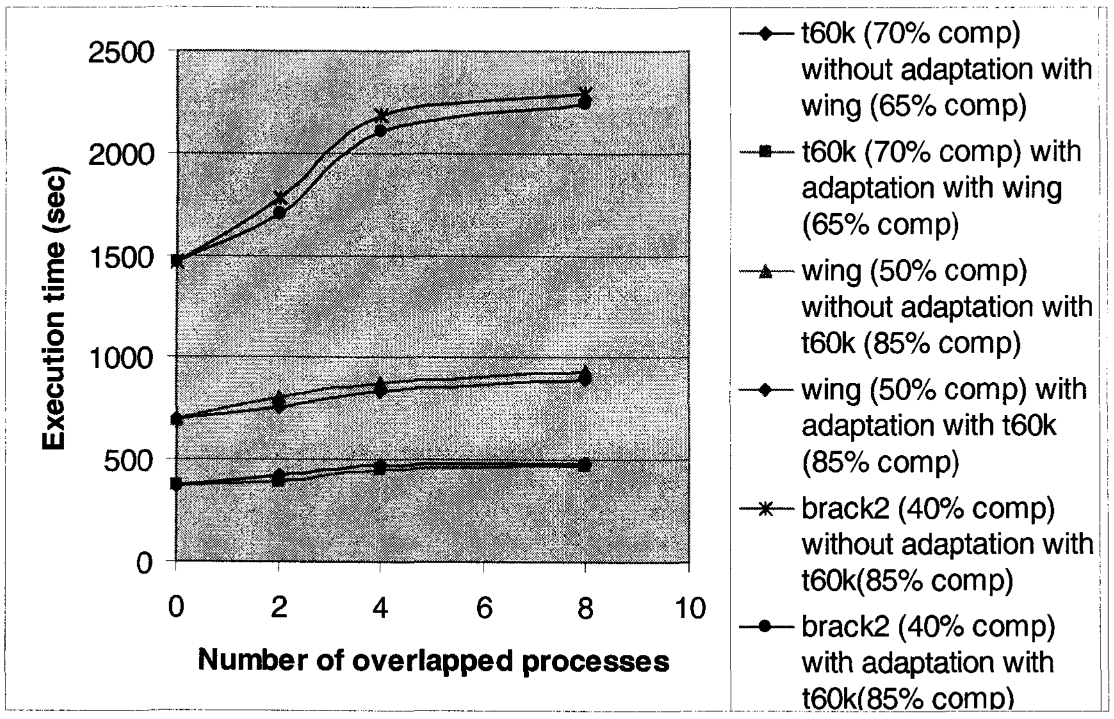

Figure 21. Execution time Vs number of overlapped processes

We found less speed up with more overlapped number of processes. This is due to the ratio of total weight in overlapped and non-overlapped processes. For an application with 8 processes and $1/f_{HT}=1.2$, with 4 processes overlapped and weight vector 1:1:1:1:0.6:0.6:0.6:0.6, this ratio is 0.6. But with 2 processes overlapped this ratio becomes 0.2.

This implies that time adaptation is more suitable for computation intensive parallel application than computation intensive applications. The following two charts depicts the cost for adaptation (using over-partition) for same pair of application but with different number of total application process. First chart depicts when 50% process is overlapped (4p from 8p application & 8p from 16p application) and the second chart depicts when same number of process is overlapped (both has 4p overlapped).

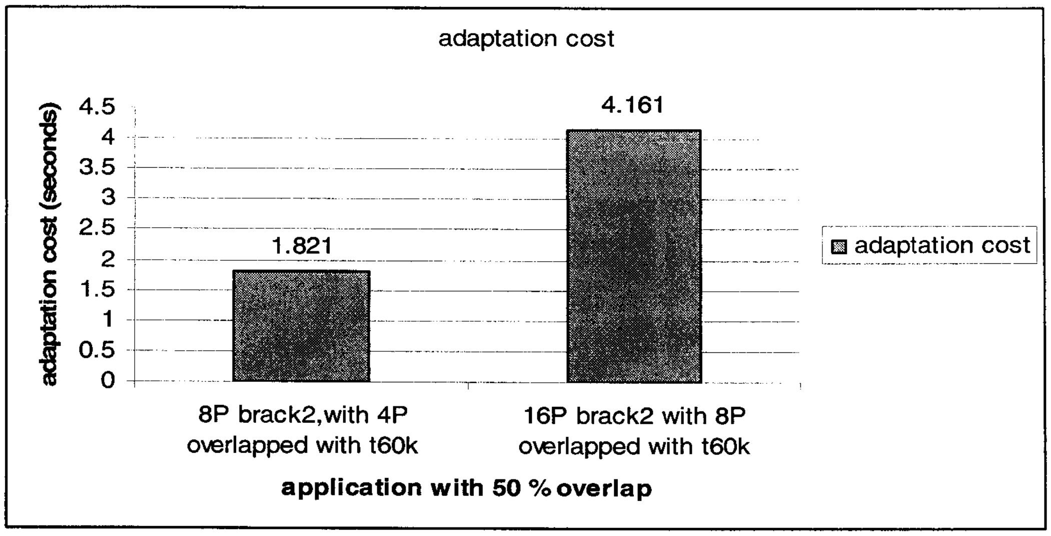

Figure 22. Adaptation cost for varying number of application process with 50% overlap

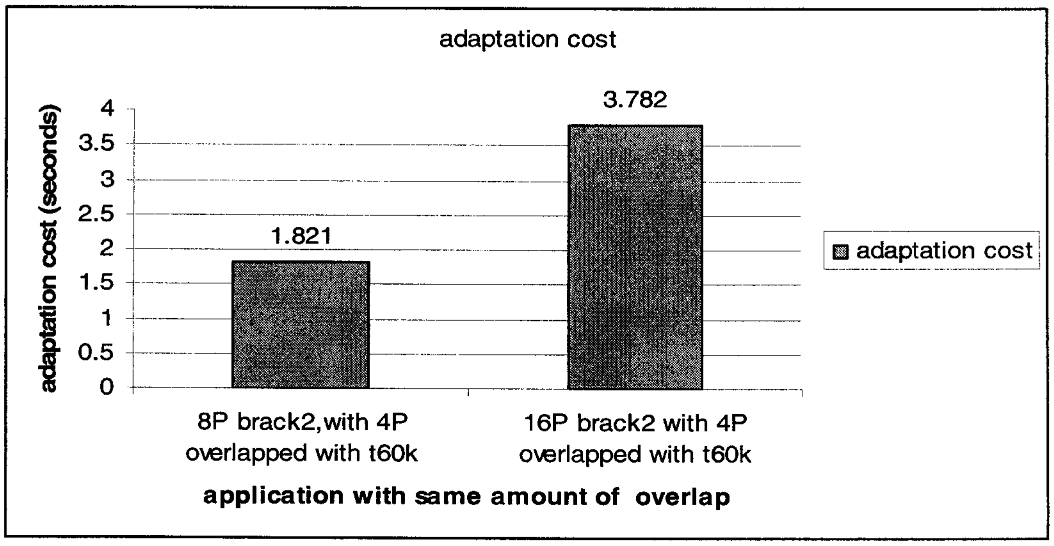

Figure 23. Adaptation cost for varying number of application process with same overlap

In the following test, t60k expands and shrinks the overlapped processes (0 → 4 → 8 → 4 → 0) while overlap with wing and brack2. This is depicted in the following figure. At T1, the overlapped process of t60k is 0. At T2, wing arrives, and overlapped process of t60k becomes 4. Again, at T3 brack2 arrives and the overlapped process becomes 8. The overlapped process of t60k becomes again 4 and 0 sequentially at T4 and T5 at the termination of brack2 and wing. So, here t60k adapts 4 times at T2 (0 → 4), T3 (4 → 8), T4(8 → 4) and T5(4 → 0). The test results are presented for this test in the following table.

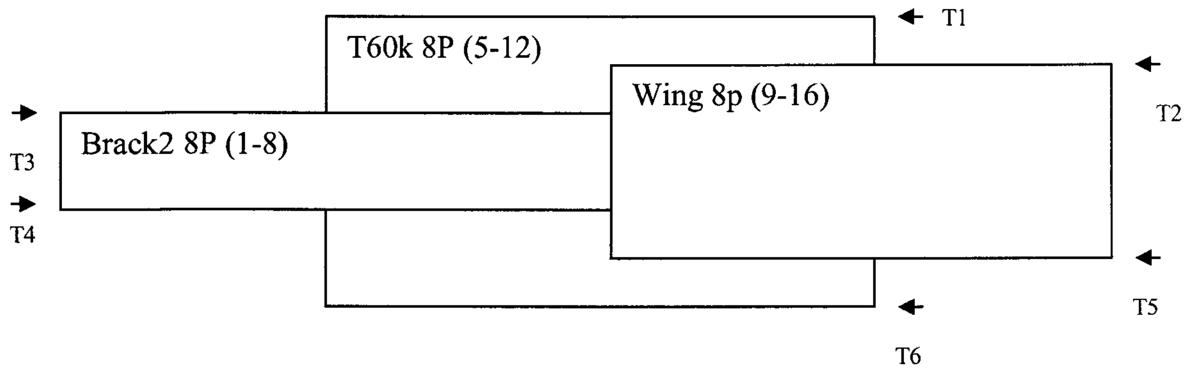

Figure 244. Expanding and shrinking of overlapped process for t60k (85% comp)

Application	Overlap	T1 (NA) second	T1 (OP) second	Speed up (OP)	Cost of adaptation (OP) second
T60k 8P 85% computation start time=T1	0	1098.1	1030.7	6.19%	0.0
	4				1.136 (wing arrival)
	8				1.548 (brack2 arrival)
	4				1.214 (brack2 termination)
	0				0.560 (wing termination)
Wing 8P 61% computation start time=T1+252 sec	4 $1/f_{HT}$=1.17	583.1	467.2	19.8%	0.0
Brack2 8P 85% computation start time=T1+384 sec	4 $1/f_{HT}$=1.15	316.7	265.1	16.1%	0.0

Table 14. Expansion and contraction of t60k with wing and brack2.

5. Conclusion and Future Work

In our approach, we have presented time adaptation for parallel application in time sharing environment exploiting the unbalanced resource allocation. We focused on adapting in time dimension, while adaptation in space dimension for space malleable application is equally important. Our adaptation framework uses over-partition and migration strategy from ATOP [11]. Our adaptation library provides option to bind either over-partition or partition from scratch. We let the application programmer decide which policy to select. But automatic and transparent binding of would be more promising if it can select the suitable method based on application and machine profiling, accessing from dynamic directory. A more appropriate weight vector can be formulated considering co-scheduling slowdown and granularity of applications that share the resources. Time adaptation in asynchronous manner as well as latency hiding is more promising.

In our design the dynamic directory is connected to the system scheduler, which can provide more accurate information about the resource share. But our implementation did not include the system scheduler. Even though, our result explains that adaptation in time dimension is meaningful, even multiple time adaptation would be meaningful with over partitioning if the application run for long enough.

Our approach is a sequence of integration of ATOP [11] approach and put that in a library so that parallel MPI application can use it for adaptation. The next phase would be to integrate the resource monitoring system and space adaptation as well. At the end this framework would be able to provide runtime adaptation for parallel application both in space and time dimension using the resource monitoring system and using the information from the dynamic directory.

References

[1] S. Ioannidis and S. Dwarkadas, Compiler and Run-Time Support for Adaptive Load Balancing in Software Distributed Shared Memory Systems, Proc. of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers, 1998.

[2] M. Zaki and W. Li and S. Parthasarathy, Customized Dynamic Load Balancing for a Network of Workstations, Journal of Parallel and Distributed Computing, volume 43, number 2, pages 156-162, 1997.

[3] H. Cai, O. Maquelin, P. Kakulavarapu, and G. R. Gao. Design and Evaluation of Dynamic Load Balancing Schemes under a Fine-grain Multithreaded Execution Model, Multithreaded Execution Architecture and Compilation Workshop, Orlando, Florida, Jan 9th, 1999.

[4] A. Rajagopalan, S. Hariri, An agent based dynamic load balancing system, Autonomous Decentralized Systems, 2000. Proceedings. 2000 International Workshop on 09/21/2000 -09/23/2000, 2000, page(s): 164-171.

[5] M. Bozyigit, History-driven dynamic load balancing for recurring applications on networks of workstations, Journal of Systems and Software Volume: 51, Issue: 1, April 1, 2000, pp. 61-72.

[6] L. V. Kale, M. Bhandarkar, and R. Brunner, Run-time Support for Adaptive Load Balancing, Parallel and Distributed Processing. Springer Verlag, 2000, ISBN 3-540-67442-X, Lecture Notes in Computer Science, Vol. 1800, (Proceedings of 4th Workshop on Runtime Systems for Parallel Programming (RTSPP) Cancun - Mexico, March 2000).

[7] R. Blumofe and C. Leiserson, Scheduling multithreaded computations by work stealing, Proceedings of the 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico., pages 356--368, year 1994.

[8] A. Sodan and X. Huang, SCOJO-Share Based Job Coscheduling with Integrated Dynamic Directory in Support of Grid Scheduling, Proc. Ann. Int. Symposium on High Performance Computing Systems (HPCS), Sherbrooke, Canada, May 2003, pp. 213-221.

[9] E. Frachtenberg, D. G. Feitelson, F. Petrini, Flexible Coscheduling: Mitigating Imbalance and Improving Utilization of Heterogeneous Resources, Proc. Int. Parallel and Distributed Processing Symposium (IPDPS'2003), April 2003.

[10] D. G. Feitelson, Job Scheduling in Multiprogrammed Parallel Systems, Extended version. IBM, RC 19790 (87657), August 1997.

[11] A. Sodan and H. Lin, ATOP-space and time adaptation for parallel and grid applications via flexible data partitioning, Proceedings of the 3rd workshop on Adaptive and reflective middleware, Toronto, Ontario, Canada, Pages: 268 - 276 , 2004

[12] K. Devine, B. Hendrickson, E. Boman, M. St.John, and C. Vaughan. Design of Dynamic Load-Balancing Tools for Parallel Applications. Proceedings of the International Conference on Supercomputing, Santa Fe, May, 2000.

[13] The Graph Partitioning Archive (source: http://staffweb.cms.gre.ac.uk/~c.walshaw/ partition/)

[14] Technical memo Dr. A. C. Sodan (supervisor), School of Computer Science, University of Windsor.

[15] M. Bhandarkar, L. V. Kale, E. Sturler, and J. Hoeflinger, Object-Based Adaptive Load Balancing for MPI Programs, Lecture Notes in Computer Science, Vol. 2074, Springer Ver;ag, pp 108-117, May 2001.

[16] G. Karypis and V. Kumar, Mulilevel k-way Hypergraph Partitioning, Proceedings of the 36th ACM/IEEE conference on Design automation, New Orleans, Louisiana, United States, Pages: 343 – 348, 1999, ISBN:1-58133-109-7

[17] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. Journal of Parallel and Distributed Computing, 1998

[18] ParMETIS: Parallel Graph Partitioning, http: //www-users.cs.umn.edu/ ~karypis/ metis/parmetis/index.html

[19] G. Karypis and V. Kumar, Analysis of Multi-level graph partitioning. In proceeding of Supercomputing 1995.

[20] L. V. Kale, M. Bhandarkar, and R. Brunner, Run-time Support for Adaptive Load Balancing, Parallel and Distributed Processing. Springer Verlag, 2000, ISBN 3-540-67442-X, Lecture Notes in Computer Science, Vol. 1800, (Proceedings of 4th Workshop on Runtime Systems for Parallel Programming (RTSPP) Cancun - Mexico, March 2000.)

[21] D. Cronk , P. Mehrotra, Load Balancing with Migrant Lightweight Threads, Publisher: Springer-Verlag Heidelberg ISSN: 0302-9743 Volume: Volume 1511/1998 Page: 153

[22] U. Rencuzogullari and S. Dwardadas, Dynamic adaptation to available resources for parallel computing in an autonomous network of workstations, ACM SIGPLAN Notices , volume 36, number 7, pages 72-81, year 2001

[23] A. Keren, A. Barak, Opportunity cost algorithms for reduction of I/O and interprocess communication overhead in a computing cluster, Parallel and Distributed Systems, IEEE Transactions on , Volume: 14 Issue: 1 , Jan. 2003 Page(s): 39 -50

[24] L. Xiao; S. Chen; X. Zhang, Dynamic cluster resource allocations for jobs with known and unknown memory demands, Parallel and Distributed Systems, IEEE Transactions on , Volume: 13 Issue: 3 , March 2002 , Page(s): 223 -240

[25] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, A framework for performance modeling and prediction, Proceedings of the 2002 ACM/IEEE conference on Supercomputing.

Vita Auctoris

NAME:	Ahsanul Arefeen
PLACE OF BIRTH:	Mirzapur, Bangladesh
EDUCATION:	1996-2001 B. Sc. Engg., Computer Science & Engineering Bangladesh University of Engineering & Technology Dhaka, Bangladesh.
	2002-2005 M. Sc., Computer Science University of Windsor Ontario, Canada
EXPERIENCE:	2000-2001 Software Developer DevNet Inc. Dhaka, Bangladesh