Space and time adaptation for parallel applications via data over-partitioning.

Lin Han
University of Windsor

Space and Time Adaptation for Parallel Applications via Data Over-Partitioning

by

Lin Han

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

2004

Your file Votre référence
ISBN: 0-612-96377-2
Our file Notre référence
ISBN: 0-612-96377-2

Abstract

Adaptive resource allocation is a new feature to run parallel applications. It is used to obtain better space and time sharing according to current workload, to schedule around obstacles through reservation and to cope with lack of accurate predictability on heterogeneous resources. The implementation of resource adaptation is potentially very expensive if total remapping or partitioning from scratch has to be performed. The existing popular run-time systems include AMPI and Dome. AMPI, which uses huge numbers of threads in MPI process to implement resource adaptation, suffers from frequent thread switches and loss of cache locality; and Dome, an object-based migration environment, suffers from lack of general language supports.

When resource adaptation occurs, load balancing techniques are used to allocate the workload fairly across processors, so that each processor takes roughly the same time to execute the processes assigned to it, and that every processor has the same workload to obtain the best performance and maximize resource utilization.

This thesis proposes a novel approach – *Adaptive Time/space sharing via Over-Partitioning (ATOP)* – to implement resource adaptation with better performance in terms of time overhead. Total workload is represented by a data graph. ATOP performs over-partitioning on the graph to create a certain number of workload pieces, or partitions, while processing partitions per processor as one data collection in a single MPI process. Typically, the number of partitions is set equal to the number of processors potentially allocated. This approach is feasible for the applications using 2^n processors. In the cases where our over-partitioning approach does not perform well, or non-fitting numbers of resources need to be chosen, ATOP still provides the alternative option to repartition from scratch.

Dedication

To

My wife, my parents

and

in loving memory of my grandparents

Acknowledgement

I would like to gratefully acknowledge the enthusiastic supervision of Dr. A. C. Sodan during this thesis work. Without her time and effort, this endeavour would not have been possible. I thank Dr. Karen Devine from Sandia National Laboratories (SNL) for the technical discussions and advices in installing Zoltan library. I acknowledge my thesis committee members, Dr. Drake, Dr. Li and Dr. Wu, who have been all generous and patient. Their confidence in my abilities has been unwavering, and has helped to make this thesis a solid work.

In particular, I am grateful to my friends and colleagues for their help and true friendship.

Finally, I am forever indebted to my wife and my parents for their understanding, endless patience and encouragement when it was most required. They are the continuous source of my strength and hope.

Table of Contents

List of Tables

List of Figures

1. Introduction

Parallel computing is an effective approach to transcend the physical limitations of processing capabilities of the traditional sequential computers. In a parallel environment, workload of an application is assigned on the processors in two steps: partitioning and mapping. In the partitioning (also called "decomposition") step, the workload is divided into multiple processes, which, as abstract software entities, execute assigned work on processors [XL97]. Each process has one or several work units (such as threads or objects) that are the smallest units of concurrency the parallel program can exploit. In the "mapping" (also called "migration" in load balancing) step, such processes are mapped to available processors by a job scheduler or load balancer. In two conditions, the workload imbalance may occur. First, the workload of a parallel application may be not static but dynamic – which means the new workload on each processor can be generated during the run-time. Second, resources allocated to the application may be adaptive – which means the job scheduler can change the resources allocation according to current workload or task reservation. Thus, it may be that some processors will complete their work and become idle while others are busy. Ideally, we want all processors to perform the tasks continuously and simultaneously, and to complete their tasks at the same time roughly in order to obtain the minimum execution time and maximum resource utilization. Load balancing is used to achieve this goal by spreading the overall workload evenly across all processors and by minimizing the overhead of communications.

In general, according to the stage when load balancing is performed, it is categorized into static and dynamic. Static load balancing is performed during compile-time, and distributes the workload on processors before real execution. One obvious advantage of static load balancing is that there is no run-time cost created during the load balancing procedure. On the other hand, dynamic load balancing is based on the redistribution of workload at execution time. During execution of an application, the dynamic load balancer invokes the load redistribution to transfer some workload from heavily loaded processors to lightly loaded processors or idle processors when imbalance is beyond a special predefined threshold, or the resources allocated on this application change. Dynamic load balancing must ensure the advantage gained by the load balancing is more than the disadvantage caused by additional overhead it generates. Because the load balancer has to collect the global workload status and calculate the new distribution, partitioning overhead in dynamic repartitioning is very expensive. Chapter 2 discusses these two categories in more detail.

Traditionally, parallel applications are executed on a fixed number of processors, and the workload on each processor is roughly the same. This kind of application is not flexible enough for job scheduling. More and more researchers have focused on the problem of adaptive resource allocation in order to obtain better space and time sharing according to current workload, to schedule around obstacles through reservation and to cope with lack of accurate predictability on heterogeneous resources. Before the following discussion, we need to clarify some concepts more precisely: "space adaptation", "time adaptation". See Figure 1. [SH04B]

Definition 1 *Space Adaptation*: The number of discrete processors allocated to a job can dynamically be changed during its execution.

Definition 2 *Time Adaptation*: The time shares allocated to the job can be different on different processors and can be changed dynamically during the job's execution.

Both space and time adaptation provides the chance for better resource utilization and better average response time. Space adaptation requires jobs to be space malleable; time adaptation requires jobs to be time malleable.

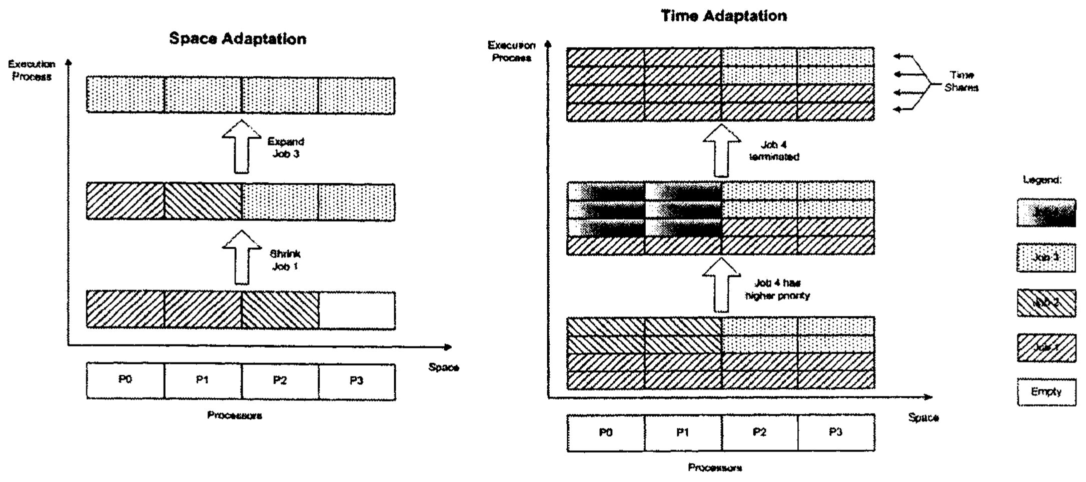

Figure 1. Space (left) and time (right) adaptation

As mentioned above, the overhead caused by partitioning in dynamic load balancing is very expensive. When resource adaptation happens, the workload has to be repartitioned and migrated to the corresponding processors. Just as with the dynamic load balancing operation, repartitioning for resource adaptation is an extremely expensive step. Table 1 shows the motivation of our research: load balancing overhead consists of partitioning cost and migration cost, of which the partitioning cost is the main part. The potential percentage of time saved by avoiding partitioning is up to 56% for this specific step. Adaptation time differs according to the adaptation situation: the results in the table reflect the middle step of adaptation (16→8 processors) with 16 partitions.

Table 1. Partitioning and migration overhead by partitioning from scratch

Graphs	Partitioning Time	Migration Time	Total Time	% of Migration Time	% of Saving
3elt	0.359	0.285	0.644	44%	56%
wing	9.517	8.081	17.598	46%	54%

The main innovative idea in our approach ATOP (Adaptive Time/space sharing via Over-Partitioning) is to employ so-called over-partitioning to create more data partitions than the processors allocated on the application. If the resource allocation changes, the load balancer simply migrates corresponding partitions and avoids recalculating how to separate the graph into new partitions. By doing so, almost all partitioning overhead can be eliminated. For space adaptation, a certain constraint is applied to the number of processors – the number of partitions divided exactly by the number of processors allocated without remainder. Therefore, we may create 32 partitions for a maximum of 32 processors and permit space allocations of 32, 16, 8, 4, 2, and 1—but not, e.g., of 5. In the cases where over-partitioning does not perform well or a non-fitting processor number is chosen, ATOP provides an alternative option to partition from scratch. For the time adaptation, we create a huge number of partitions which is much larger than the processors allocated, and approximately remap these partitions according to allocation of certain time shares. Partitioning from scratch is also provided for time adaptation if it can give better performance or distribution results.

The remainder of the thesis is organized as follows. The state of the art is reviewed in Chapter 2, including the problem of load balancing, job malleability, and run-time system supports. Chapter 3 focuses on a discussion of two related libraries: ParMETIS and ZOLTAN. The main algorithms, functions and interfaces are introduced. Chapter 4 describes ATOP approach in detail, introduces the motivation, general concepts, partition allocation strategy and cost model. Chapter 5 illustrates the experimental results and test analysis. Finally, Chapter 6 concludes by reiterating the contributions of this method and proposes future research directions.

2. Review of the State of the Art

The load balancing problem has been investigated for many years. In this chapter, we will review the existing load balancing categories, algorithms, and run-time systems. As well, the malleability problem will be introduced in this chapter, including space malleability and time malleability. Finally, we will describe a popular approach to deal with the data distribution problem, namely graph partitioning, and various algorithms used to partition a graph.

2.1 Load Balancing Problem

There are two categories of load balancing approaches according to the stage when the load balancing approach is performed. Load balancing can be attempted as static load balancing before the execution of an application, only in compile-time. Or it can be performed as dynamic load balancing during the execution of an application, and performs workload redistribution in run-time. There are several algorithms for each of these two categories.

2.1.1 Static Load Balancing

Static load balancing is usually referred to as the "mapping problem" or "scheduling problem" [Bok81]. It typically uses optimization techniques that greatly rely on the information regarding the task execution time and resources allocation. This information is assumed to be known before the execution of a parallel application at compile-time. The primary advantage of static load balancing is that the run-time overhead caused in the load balancing procedure can be completely avoided. Moreover, static load balancing sometimes is the only choice for such parallel applications whose sizes are too large to be migrated among processors during run-time.

Two static load balancing algorithms are:

- Round Robin algorithm

  Assigning tasks in sequential order of processors, coming back to the first one when all processors have been allocated one task
- Randomized algorithm

  Allocating the tasks randomly on processors

However, there are several fundamental disadvantages to static load balancing. First of all, it is not suitable for the applications whose workload may be created dynamically at run-time. Because the workload and necessary resources of such an application are unpredictable, the data distribution on processors created by static load balancing is inherently inaccurate. In fact, without executing a parallel application, it is very difficult to estimate accurately the execution times of various parts of the program [WA02]. In addition, some applications have an indeterminate number of steps to gain the final results. For example, with search algorithms in an irregular tree structure, typically it is unknown how many paths one must traverse or how deep to go through the tree, and we do not know whether it should be done in parallel or sequentially at compile-time. Furthermore, in a heterogeneous environment, the computational speeds of processors are different. It happens generally that the workload, which is divided fairly by static load balancing at compile-time, completes earlier on faster processors than on slower processors. Finally, static load balancing approaches do not work for malleable applications. Since "when" and "how" the adaptation will happen is unknown before execution of a malleable application, it is impossible to use static load balancing approaches in these circumstances.

2.1.2 Dynamic Load Balancing

Before discussing the matter further, it is necessary to explain terminology in advance. Load balancing allocates *processes* onto *processors* and as usual, each processor is mapped with only one process. A process is the software entity of the corresponding processor. The workload of a parallel application which can be represented by a *data graph* is divided into many *objects* or *work-shares* that are represented as vertices in the graph. In such a data graph, *partition* is a set of objects that always migrate together. Each process performs several objects or work-pieces on one processor. (See Section 2.3 for details.) So the terms *process* and *processor* are used interchangeably in this thesis.

In dynamic load balancing, the above factors – execution time, resources needed and current step – have been taken into consideration when dividing workload according to the status of execution and resource utilization. Therefore, it has higher efficiency than static load balancing. Dynamic load balancing must also ensure the advantage obtained by the load balancing is more than the disadvantage caused by additional overhead it generates.

As mentioned above in the introduction, during the execution of an application, workload imbalance occurs in two conditions: the workload on processors changes during the execution time; or the resources allocated on the application change dynamically. Traditional dynamic load balancing works in the former case by monitoring the workload status on each processor and invoking load redistribution operations to transfer some workload from heavily loaded processors to lightly loaded processors when the imbalance is beyond a predefined threshold. Such kind of dynamic load balancing operations can be initiated by the system periodically or by the application explicitly. In the latter case, when system resources change, an overall remap or total data redistribution is generally required. When resource adaptation occurs, the load balancer performs an overall repartitioning and transfers some data among the processors according to the current workload status. The problem of existing approaches in this case is that the overall repartitioning (or partitioning from scratch) is very expensive. In Chapter 4, our ATOP approach will be introduced to solve this problem.

According to the number of specific load balancers, traditional dynamic load balancing can be classified into centralized or distributed. If there is one or several master processors working as the load balancers to control the slave processors and direct the data migration, it is called centralized dynamic load balancing. Otherwise, if there is no specific load balancer and all processors are equal when doing load balancing, it is distributed dynamic load balancing. Furthermore, according to the communication type in dynamic load balancing operation, it can also be categorized as synchronous approach and asynchronous approach. If the application has to be stopped during the execution of load balancing, it is synchronous load balancing; otherwise it is asynchronous one. We will introduce these two categories in detail.

2.1.2.1 Centralized vs. Distributed Dynamic Load Balancing

In centralized dynamic load balancing, a master/slave structure exists. The master processor calculates the repartitioning and controls every salve processor directly.

In the case of so-called divide-and-conquer, the workload of an application first is partitioned into many work-shares and kept in a work-share queue or pool maintained on the master processor. The master processor passes a work-share to the slave processor when the slave processor completes one work-share and requests more. If all work-shares have the same priority, it is best to hand out the work-shares that take a longer time and cost more resources first to avoid slave processors sitting idle and waiting for the larger work-shares to be finished. If the work-shares have different priorities, a general way is to maintain a separate queue for each priority, and hand out the work-shares from the queues in descending order. Figure 2 shows the divide-and-conquer technique with three different priority queues.

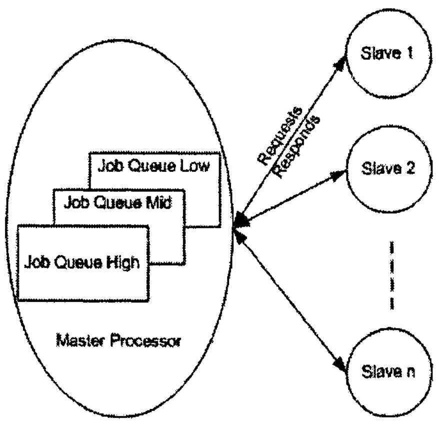

Figure 2. Centralized job queues with different priorities

In another case of centralized strategies, the workload of an application is partitioned and broadcasted to all allocated processors in the initial stage. And a centralized dynamic load balancer will make repartitioning when imbalance occurs and direct the data migration among slave processors. This approach usually depends on a global view of overall workload status across processors. The master processor must obtain such view before performing a load balancing operation [BOS96]. In practice, the master processor collects and maintains the workload information of all slaves and initiates dynamic load balancing by special trigger policies – periodical initiation or demand initiation. Both the workload information and the workload itself can transfer among all processors either directly or in a relay fashion. In this case, in addition to assigning the work-shares to slaves and keeping workload status information, the master processor can even adjust the global thresholds periodically according the runtime status and therefore balance the workload for better distribution [XH90].

The centralized strategies led to the best results in general cases [TVC+02]. But there must be a master processor, and because the master process can only issue one work-share at a time, a bottleneck in communication could develop. Hence, centralized load balancing strategies are only suitable for the parallel environment with few slaves and the work-shares are computationally intensive [WA02].

The centralized approach can be improved by using a hierarchy structure to solve the bottleneck problem. A variation of the above centralized work-shares queue approach is to separate the centralized queue or pool into several sub-queues or sub-pools, and to distribute these small sub-queues to several master-agents. Each master-agent controls a group of slave processors. During load balancing, a master-agent would find an optimum in its local group, and transfer the optimum to the master processor. The master processor can select the optimal solution from all optimums collected from those master-agents. Figure 3 shows the distributed work-share queues.

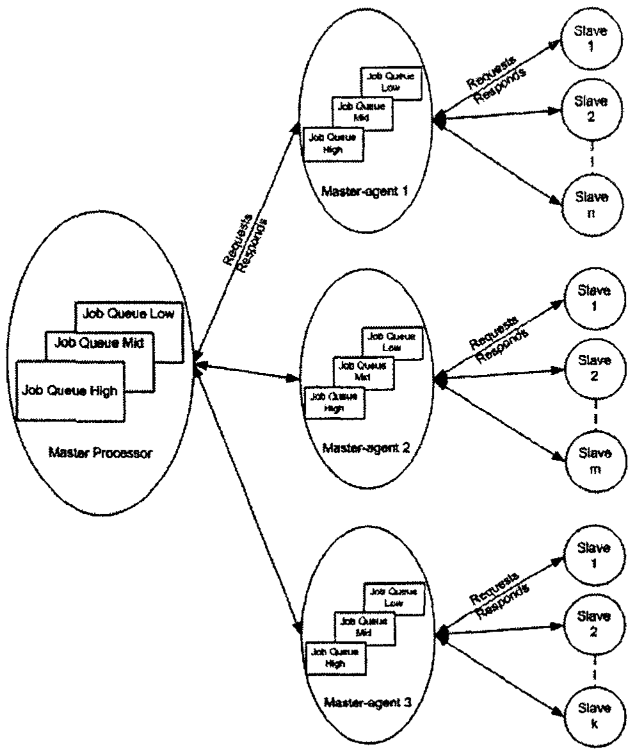

Figure 3. Distributed job queues with 3 master-agents

By contrast to a centralized load balancing approach, the distributed load balancing algorithms do not depend on one or several master processors and can perform load balancing operations locally. All processors are the same in distributed dynamic load balancing approaches. Thus it avoids the bottleneck of communication and allowing for better scalability. The most popular distributed algorithms are nearest neighbour algorithms, including diffusion algorithm, dimension algorithm, and gradient algorithm. In this thesis, we use the diffusion algorithm to demonstrate the main advantage of distributed algorithms.

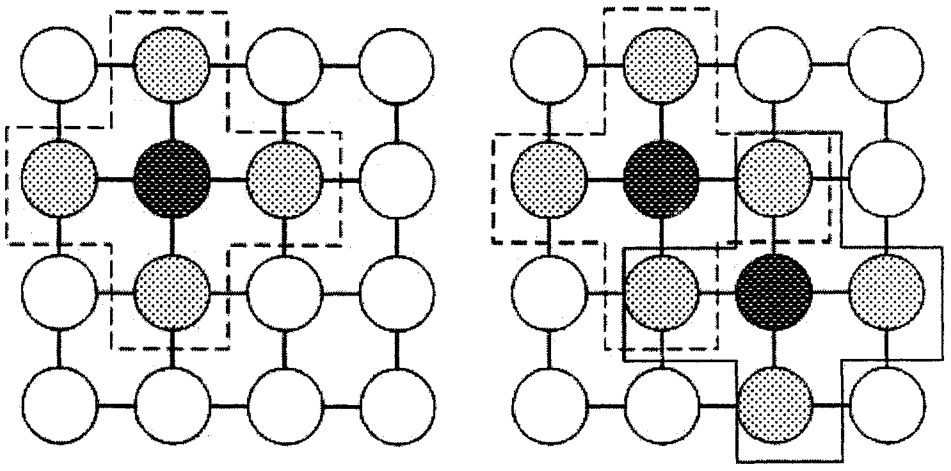
Figure 4. Neighbours and overlapping neighbours in 4 x 4 mesh

In the diffusion method, a processor balances the workload with all of its nearest neighbours simultaneously, gives some work-shares to one or more neighbours, and maybe requests some workload from other neighbours as well. After one load balancing operation, the workload among the processor and all its neighbours is balanced. Figure 4 (left) shows an example of the neighbours in a 4 x 4 distributed computing system, and Figure 4 (right) shows two overlapping domains of neighbourhood on a mesh topology. In the figure, the grey nodes are the neighbours of the dark nodes. In the right figure, two dark nodes' neighbour domains overlap. In [CRS+99A], performance of a diffusion method was compared to another popular distributed load balancing strategy: Generalized Dimension Exchange (GDE) and the experimental results show that the diffusion method obtains better load balance in ring and hypercube topologies. In Chapter 5, a specific diffusion algorithm, the adaptive repartitioning algorithm in ParMETIS [SKK03] is used to compare our over-partitioning approach. More load balancing algorithms are described in Chapter 3 within the Zoltan library.

2.1.2.2 Synchronous vs. Asynchronous Dynamic Load Balancing

In addition to centralized and distributed categories, the dynamic load balancing can also be classified as either synchronous or asynchronous. Synchronous dynamic load balancing stops the application execution, performs a redistribution of the workload, and then continues the execution. Because the application on all processors has to be halted during performing load balancing, synchronous approaches are also called "semi-static dynamic load balancing". Such balancing can be split into the two basic steps of (1) data partitioning (calculating the new distribution of the workload) and (2) data migration (moving the workload to the new location). In general, semi-static synchronous approaches are more feasible where the quality of distribution is important [SKK00].

On the other hand, asynchronous dynamic load balancing does not stop the execution of applications. If any processor needs more work-shares, it informs the load balancer (in centralized model) or exchanges work-shares directly with another processor which has additional workload (in distributed model). The most well-known example of the asynchronous dynamic load balancing approaches is "work stealing" proposed by the MIT group. In work stealing, if a processor runs out of workload, it asks another processor which is chosen randomly for work. Thus, other processors continue performing the application without interruption.

There are two main synchronous approaches to deal with the workload distribution problems in load balancing: geometric partitioning and graph partitioning. Geometric partitioning is based on a geometric representation of workload. And graph partitioning is on a graph without coordinates. In this thesis, we only discuss the graph partitioning in section 2.3. Our ATOP approach is a semi-static (or synchronous) approach that performs the graph partitioning to calculate the initial data distribution first; then it stops the application execution and performs the partition redistribution and migration when resource adaptation occurs; finally, it continues executing the application again.

2.2 Malleability Problem

Typical parallel applications are executed on a fixed number of processors; the resources – the processor number or time shares – cannot be changed in run-time, and the workload allocated on each processor is roughly the same. This kind of application is not flexible enough for job scheduling, and cannot improve the system utilization [SH04A]. In a study with Supercomputing-Center users, it was found that 98% of the jobs were moldable, i.e. can run with different initial resource allocations [CB03]. In [SH04A], it was shown that response times are better by up to 50% if jobs are malleable. Figure 5 shows such a case of inflexibility. Job 1 is executed on three processors, and because Job 2 needs two processors to run, it is impossible to perform Job 1 and Job 2 simultaneously on a four-processor system. One processor is idle even there is a Job 2 in the waiting queue of the system. However, if the Job 1 can be executed with an adaptive resource, its space can be dynamically "shrunk" on two processors; Job 2 is now able to run with Job 1 simultaneously without any idle processor. It will of course improve the system utilization and obtain better overall response time.

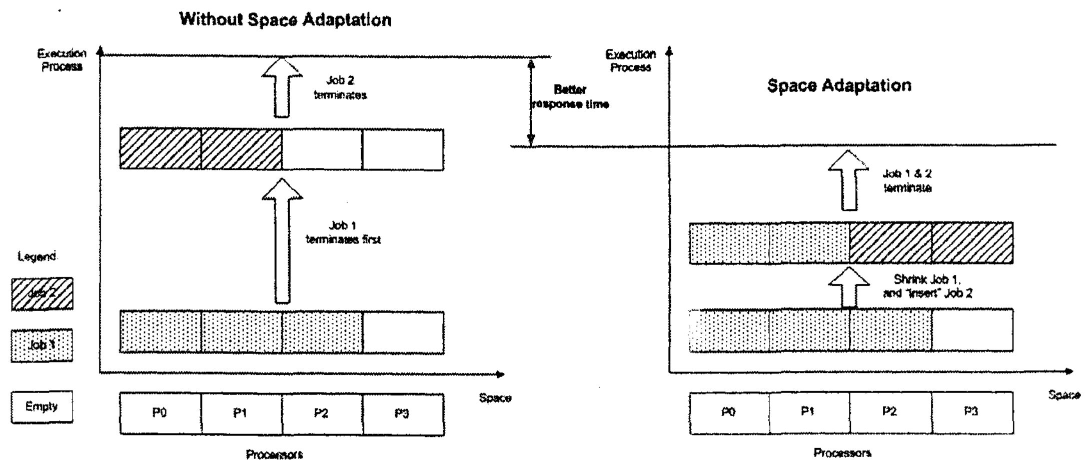

Figure 5. Improvement by space malleable jobs

Here we give another two related definitions: “space malleable” and “time malleable” [SH04B].

Definition 3 *Space Malleable*: A job is space malleable if it is capable of running with dynamically chosen and potentially changing numbers of processors.

Definition 4 *Time Malleable*: A job is time malleable if it is able to run with different time shares and potentially changing time shares on different processors.

At present, two kinds of techniques are used to implement adaptive parallel applications: object-based technique and thread-based technique. In the former category, the migration units are parts of an object or data members of objects; while in the latter the threads are the smallest units to migrate across the processors. In the following part of this section, we introduce two runtime systems that support resource adaptation by using the above techniques: Dome (object-based), and AMPI (thread-based).

2.2.1 Dome (Distributed Object Migration Environment)

Dome (Distributed Object Migration Environment) [NAB+96] supports the reconfiguration of executing applications by allowing the modification of parallelism of applications. It provides a distributed object library for parallel programming to support dynamic load balancing at run-time. In the library, Dome defines some classes (Dome objects), which hide the detailed implementation of parallelism. When such an object is instantiated, it is automatically decomposed and distributed within the parallel environment. Workload is represented as the data members of the objects and such members constitute the basic migration units of the objects. Dome clones the program on every node and keeps track of all Dome variables in the program. During partitioning (or decomposing) the objects, three methods can be chosen: *whole*, *block* and *dynamic*. In the *whole* partitioning, all data elements of the Dome object are cloned on the distributed processes; the *block* partitioning method discomposes the object workload evenly on each node; the *dynamic* method divides the object elements evenly initially, and repartitions these elements periodically among the processors according to the dynamic workload status.

In order to balance the workload periodically, a timer is added to the Dome operations and it measures the amount of time spent on each processor for doing computations. During the load balancing phase, these times are collected and compared to remap the workload in every node. Therefore, in Dome, the load balancing operation decisions are made based on the time taken for each workload during the last computing phase. In addition, Dome implements both global and local load balancing algorithms. In the global method, a master node collects the time information, remaps the workload and broadcasts the new ideal workload distribution. Although this method will always yield the more accurate balance result, it may cause a large amount of data migration, as well lead to the limitation of scalability. In the local method, by contrast, every node in Dome just exchanges the workload with its neighbours. It will not result in a global optimal balance, but it is easy to expand to a large number of processors and requires less data migration.

The problem of Dome is that Dome does not support the most popular MPI library.

2.2.2 AMPI (Adaptive MPI)

Adaptive MPI (or AMPI for short) is a variation of standard MPI (Message Passing Interface). AMPI enables the parallel application to be executed adaptively, which is not supported by the traditional MPI programming model. "The basic idea behind AMPI is to separate the issue of mapping workload onto processors from that of identifying work to be done in parallel [HLK03]." In traditional MPI, a parallel application is always divided into a special number of processes, which are allocated on the same number of physical processors. However, in AMPI, a huge number of logical MPI processes are created. The number of such logical processes is generally much larger than the number of the physical processors. That means on every physical processor, there will be many logical MPI processes to execute. As a result, the programmers will never be restricted by the limitation of the number of the processors. The programmers only focus on deciding what to do in the parallel applications, while the run-time system decides when and where to run these programs.

In practice, the logical processes are implemented as user-level threads in MPI programs. The smallest migration unit in AMPI is such a thread (or logical process). By transferring a number of logical processes among the processors, the parallel system gains the load balance at run-time. In conventional MPI programs, because there is only one process executing on each physical processor, there is no problem for MPI processes to keep only one set of global variables. However, such a unique set of global variables is not suitable for AMPI, where many logical processes are allocated on one physical processor. The global variables in each logical MPI process must be different from the ones in other logical processes. Therefore, the AMPI programs have to be modified to "encapsulate the global variables into a dynamically allocated structure" [KKD02]. Only by doing such encapsulation can AMPI provide good language support for easily creating adaptive programs.

Although AMPI provides applications with resource adaptation by allocating dynamically the workload that is represented as logical MPI processes across processors, it still suffers from inefficient partitioning and a rigid number of threads. The main problem of this approach is that when a larger number of threads is created on each processor, the overhead of the thread-switch is very expensive and will greatly slow down the program execution. Another shortcoming is that the number of logical processes or threads cannot be changed dynamically. Once it is set at the initiation stage, the application can only use such a number of threads. However, from our research in chapter 4, partitioning with a certain number of partitions (similar as the threads in AMPI) does not always obtain the best performance.

2.3 Graph Partitioning / Repartitioning

The main idea of graph partitioning is to represent the computational application as a weighted graph. The weights of vertices in the graph correspond to the amount of computation. The edge-weights in the graph usually correspond to communication costs. In graph partitioning, each vertex is allocated into one of K possible sets called partitions, and the total weights of the edges between different partitions are called edge cuts. The aim of graph partitioning is to find a partitioning of the graph that minimizes the edge cuts subject to the partitions having approximately equal vertex weights. In graph repartitioning, a partitioning already exists. The problem is to find a good partitioning that divides the vertices of a graph into roughly even sets "similar" to the existing partitioning. This keeps the migration cost low. These two kinds of problems described above are NP-hard problems. Therefore, no efficient exact algorithm is known. In this thesis, we use two dominant graph partitioning algorithms: the parallel multilevel K-way graph partitioning algorithm [SKK93] and the adaptive repartitioning algorithm [SKK97] to compare the performance with our over-partitioning.

The K-way graph partitioning algorithm is based on a serial partitioning algorithm – multilevel K-way algorithm described in [KK98]. It consists of three phases: graph coarsening, initial partitioning and uncoarsening/refinement. In the graph coarsening phase, a relatively coarser graph is created by combining adjacent vertices in the original input graph together. This coarsening step is performed several times and a coarsest graph is created. On this coarsest graph, the Kernighan-Lin partitioning algorithm is used to calculate the initial partitioning. Since the graph is coarsest, the domain of the problem becomes very small, and the calculation will be completed much quicker than partitioning from the original graph. After getting the partitioning, partition refinement is performed to convert the coarsest graph back to the finest (i.e., original graph). In the cases where it is impossible to get the balance in a coarser level, the graph needs to be uncoarsensed one level to increase the vertex number, and then to do the calculation again. Figure 6 shows the Multilevel K-way Partitioning.

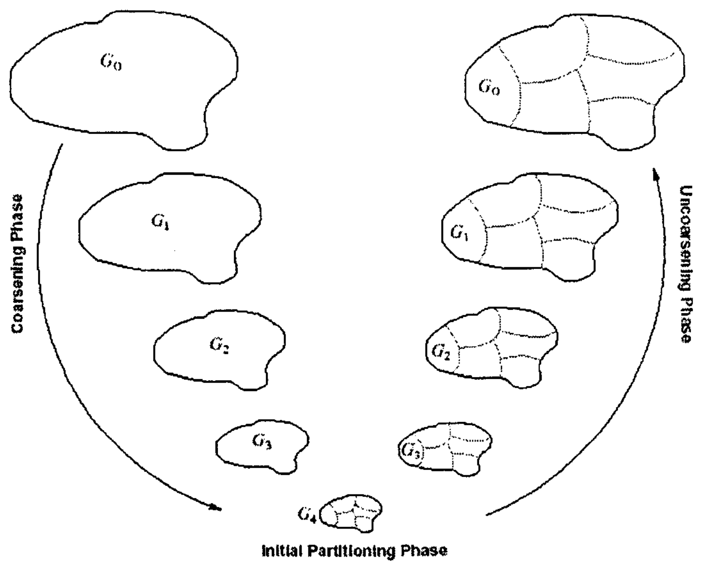

Figure 6. Multilevel K-way Partitioning [SKK03] Page 6

The adaptive repartitioning algorithm is an incremental variation of the diffusion partitioning algorithm, taking the initial partitioning status into consideration to decrease the migration cost during repartitioning. The main difference is that while calculating the vertices to be transferred, the adaptive repartitioning algorithm only considers the border vertices. If a vertex is in an over-balanced partition, it will migrate to a non-overbalanced neighbour partition. If more than one non-overbalanced neighbour partition exists, the vertex will migrate to the partition that results in the least edge cut. After all border vertices are visited once, this phase will be repeated until balance is obtained. Similarly, if it is impossible to get the balance in a level, uncoarsening has to be performed again.

3 Zoltan library

Zoltan is a collection of data management services for unstructured, adaptive and dynamic applications [BDH+02]. It includes a suite of parallel partitioning algorithms, data migration tools, unstructured communication services, and dynamic memory management tools. Therefore Zoltan is strictly speaking not a method or algorithm but rather a collection of methods and services. The data structure of Zoltan is not dependent on the applications data structure; rather it is a data-structure neutral library. This feature allows it to be used by a variety of applications without imposing a particular data structure on an application. Zoltan is a highly portable library. It runs on various platforms and operating systems, such as SUN/Solaris, IBM/AIX, ASCI Red, and several Linux clusters. In addition to several of its own load balancing approaches, Zoltan incorporates the Jostle and ParMETIS approaches.

3.1 Zoltan's General Interface

A great feature of Zoltan is its general interface [DHB+00] that enables the application developers to continue using their own data structures. In order to keep the interface simple, Zoltan employs a set of function interfaces called "callback function interfaces" or "query function interfaces", in which Zoltan queries the application for needed information and the application must implement these callback functions to answer Zoltan's queries. The callback function is "registered" in Zoltan by passing it a pointer to the function, and then Zoltan calls that function when corresponding information is needed. Basically, query functions can be divided into General Zoltan Query Functions and Migration Query Functions. Table 2 shows the main callback (or query) functions we used in this thesis.

Table 2. Zoltan's query functions

QUERY FUNCTIONS	RETURNED INFORMATION
ZOLTAN_NUM_OBJ_FN	Number of objects that are 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 assigned to the processor & the next object assigned to the processor
ZOLTAN_PARTITION_MULTI_FN or ZOLTAN_PARTITION_FN	A list of partitions to which given objects are assigned or partition to which a given object is assigned
ZOLTAN_NUM_EDGES_MULTI_FN or ZOLTAN_NUM_EDGES_FN	Number of edges for each object in a list of objects or number of edges for a given object
ZOLTAN_EDGE_LIST_MULTI_FN or ZOLTAN_EDGE_LIST_FN	Lists of global IDs, processor IDs, and edge weights for object sharing edge(s) with a given object or objects list
ZOLTAN_OBJ_SIZE_FN	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

When different partitioning algorithms are performed in Zoltan, different query functions are needed for the library. For example, when performing "PartKway - multilevel Kernighan-Lin partitioning" algorithm, the following query functions have to be implemented in advance: ZOLTAN_NUM_OBJ_FN, ZOLTAN_OBJ_LIST_FN, ZOLTAN_NUM_EDGES_FN, and ZOLTAN_EDGE_LIST_FN. In addition, the ZOLTAN_PRE_MIGRATE_PP_FN and ZOLTAN_POST_MIGRATE_PP_FN query functions are optional for migration. If they are registered, they will be called at the beginning, and end of migration routine of Zoltan.

Table 3. Zoltan's operations

OPERATIONS CATEGORIES	ZOLTAN'S OPERATIONS	SEMANTICS OF OPERATION
Initialization and Finalization	Zoltan_Initialize	initiates MPI for Zoltan
	Zoltan_Create	creates a Zoltan instance, allocates memory for Zoltan information and sets the default values.
	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	provides Zoltan the registration information of application-supplied query functions (call back functions).
	Zoltan_LB_Free_Part	returns the memory allocated for the import and export arrays during execution of load balancing and migration.
	Zoltan_Destroy	frees the memory allocated to a Zoltan structure and sets the structure to NULL
Partitioning	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 (zero)
	Zoltan_LB_Partition	invokes the real load-balancing routine that was specified using Zoltan_Set_Param function with the LB_METHOD parameter.
Migration	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

In addition to Zoltan's query functions, this library of course provides a suite of operations to perform the graph partitioning and data migration. When load balancing is performed in Zoltan, these operations are executed in three stages: initialization stage, partitioning stage, migration stage, and finalization stage. In partitioning stage, Zoltan's partitioning routine returns import/export lists, describing the object movements necessary to implement the new partitioning based on the old one.

Data migration using Zoltan's migration tools can be accomplished in two different ways: auto-migration or user-guided migration. For some applications, only the objects used in balancing must be migrated and no auxiliary data structures must be moved, such as the particle simulation applications, in which the load balancing is based on the number of particles per processor and we have to move only the particles and their data to establish the new distribution. For such applications, Zoltan's auto-migration tools can be used. Other applications, such as finite element methods, perform load balancing on the objects of a finite element mesh, but objects that are moved to new processors also need to have their connection information moved to the new processors. In this complex case, more user-controlled approaches to data migration are required than the auto-migration. Users have to implement their own data migration functions. In the cases of graph partitioning, the objects and their connection information have to be moved to new processors. Thus the auto-migration is not applicable for our ATOP approach for resource adaptations. Table 3 lists the main Zoltan's operations based on Zoltan Version 1.52

3.2 Algorithms provided by Zoltan

As mentioned above, Zoltan provides several of its own algorithms and incorporates the Jostle and ParMETIS approaches as well. The following dynamic load-balancing algorithms are included in the Zoltan library of version 1.52 [BDH+02]:

- Recursive Coordinate Bisection (RCB)
- Recursive Inertial Bisection (RIB)
- Hilbert Space-Filling Curve (HSFC)

- Refinement Tree Based Partitioning (REFTREE)
- Octree Partitioning (OCTPART)
- ParMETIS (PARMETIS)
- Jostle (JOSTLE)

The first three approaches, RCB, RIB and HSFC, need geometric information. The fourth and fifth algorithms, REFTREE and OCTPART, use a tree-based workload representation.

We do not consider Jostle but ParMETIS in the context of this thesis. ParMETIS [KSK03] is a parallel library that performs the partitioning in parallel. The following three partitioning approaches are related in this thesis:

- PartKway - multilevel Kernighan-Lin partitioning
- RepartGDiffusion - diffusion algorithm
- AdaptiveRepart - adaptive repartitioning

PartKway is a multilevel K-way partitioning algorithm that divides the graph into K partitions. RepartGDiffusion is a global diffusion algorithm. And AdaptiveRepart appears only in ParMETIS 3.0 and higher version. It is an incremental algorithm with small migration cost. The first two algorithms are used in this thesis to compare their performances with our over-partitioning algorithm.

In Zoltan, users choose the algorithm to perform by setting a string parameter – LB_METHOD through calling the function Zoltan_Set_Param (see table 3).

3.3 Implementing Resource Adaptation with Zoltan

In this section, the methods we used to implement resource adaptation by exploiting the Zoltan library are discussed.

For space adaptation, (1) the function Zoltan_LB_Partition is executed to create more data partitions than the processors currently allocated on the application. (2)

From the returned import/export lists, we get the information on which object is allocated to which partition and which partition is mapped on which processor. (3) After data migration by calling Zoltan's migration routine Zoltan_Migrate, the workload is balanced on current resources and a global view of the workload across the processors allocated on the application is established. (4) When the number of allocated processors changes, we modify the import and export lists with corresponding objects and execute the Zoltan_Migrate again. (5) Finally, one processor broadcasts the refreshed global workload view across the new resource domain. For time adaptation, in addition to the above Zoltan_LB_Partition function, Zoltan_LB_Set_Part_Sizes is used to change the relative weights of partitions dynamically.

Because Zoltan, ParMETIS and Jostle do not support resource adaptation explicitly, and do not provide interfaces to shrink or expand the available processors, we have to use empty partitions to implement space adaptation. However, empty partitions are not permitted for most algorithms in ParMETIS, except K-way and Global Diffusion algorithms. Thus we can only use these two approaches in space adaptation. Diffusion is not used for initial distribution but for re-distribution with the most incremental nature of balancing. Since space adaptation is changing the workload radically on some processors (e.g. empty processors), we consider diffusion only for time adaptation.

4 Our Approach – ATOP

4.1 General Architecture of our Approach

In our approach ATOP, we employ a so-called "over-partitioning" approach to address the goals of eliminating the partitioning overhead and providing resource adaptation. The over-partitioning concept was proposed in [GK98] in 1998. However, it was not originally designed for load balancing with resources adaptation, but for traditional dynamic load balancing. When workload imbalance occurs, the partitions are exchanged between a heavily loaded processor and a lightly loaded processor to balance the workload, while the partition number on each processor remains the same during all procedures. In our approach ATOP, we employ over-partitioning to create more data partitions than the processors currently allocated on the application. When the resource adaptation occurs, the corresponding partitions will migrate among processors without re-calculating a new partitioning. Therefore, the overhead for repartitioning is almost eliminated. In order to keep the workload on each processor roughly the same, the numbers of partitions and processors must satisfy a specific constraint, that is, the number of partitions divided by the number of processors must be without a remainder. For example, we may create 16 partitions for a maximum of 16 processors and then permit using 16, 8, 4, 2, and 1 processors — but not, e.g., of 3 or 10 processors. In addition, we investigate another case where the created partitions are significantly more than the available processors in a system (e.g. 128 partitions on a cluster with 16 processors). Then it is possible to allocate these partitions on a more arbitrary number of processors. However, a risk of increasing edge cuts exists under this condition. With the cases where over-partitioning does not perform well or non-fitting resource numbers need to be chosen, ATOP still provides an alternative option to do partitioning or repartitioning from scratch.

For time adaptation, because the total partition numbers on all processors may not be a multiple of the available processor number, we employ much finer grains when performing over-partitioning, e.g. creating 128 partitions on 8 processors. Then we can allocate the corresponding number of partitions on each processor according to the relative weights approximately. Certainly, in time adaptation the data distribution after over-partitioning will be better if much finer grains (or a larger partition number) are created. Figure 7 shows the general architecture of our ATOP approach.

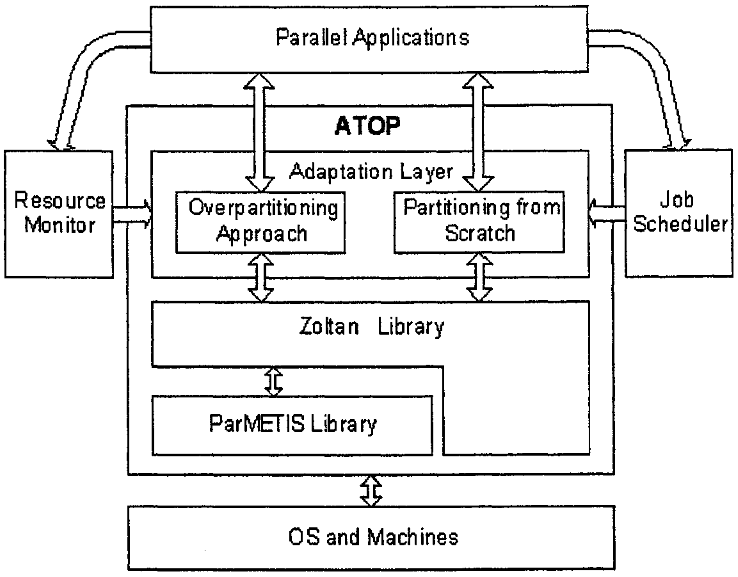

Figure 7. The general architecture of our adaptation framework

4.2 Partitioning and Migration Strategy

As mentioned above, in space adaptation, we have two options to perform over-partitioning – to create the partitions equal to the maximum number of processors that may be allocated to the application, say the number of total processors on a cluster, or to create many more partitions than the number of available processors. The results, especially the edge cuts, from these two options will be investigated in our experiments. Furthermore, in time adaptation, a significantly larger number of partitions should be created in order to make accurate partitioning in proportion to the expected relative weights.

In addition to deciding the partition numbers, partitioning and migration strategy also determine how to allocate partitions across processors. For time adaptation, it is relatively simple – ATOP uses a serial manner to allocate the partitions. Because the time shares on processors in the next adaptation step are totally unpredictable, we have to allocate partitions one by one on each processor sequentially. However, in space adaptation the problem becomes more complicated. In order to improve the quality of distribution of over-partitioning, ATOP should not only keep the workload on all processors roughly the same, but also try to decrease the edge cuts. The ideal solution is to create a hierarchical structure (see Figure 8) and use it internally by the over-partitioner. Therefore, if several partitions should be allocated on one processor, we should allocate neighbouring partitions or the partitions that have the same parent partition together. This ideal solution always obtains the lowest possible edge cuts. Currently, we assume that the partitions are delivered from Zoltan in a sequential order that reflects the neighbourhood relation of the partitions. We call this kind of migration with sequential order of partitions "structure-oriented order" that allocates the first P/N partitions to the first processor, the second P/N to the second processor and so on, for P partitions and N processors. However, it will cause more data migration than the following "migration-oriented order"

In contrast to "structure-oriented order", "migration-oriented order" focuses mainly on decreasing the migration overhead. It calculates the destination processor by $P_{roc\_id} = P_{art\_id} \bmod N$ with P_{roc_id} being the destination processor id and P_{art_id} the partition id. Although "migration-oriented order" can decrease the migration overhead, it will increase edge cuts at the same time. The tradeoff will be investigated in the experiment.

As we mentioned above, the potential approach to improve the quality of distribution with over-partitioning is to create a hierarchical structure of partitions if internally used by the partitioner (see Figure 8). Then, if multiple partitions should be allocated to a processor, we can allocate neighbouring partitions or siblings of the same parent partition together to obtain the lowest possible edge cuts. To implement this internal hierarchical structure, we have to modify the internal load balancer in ParMETIS. It is not considered in this thesis. However, we will investigate whether the partitions are delivered in a sequential order that reflects the geometric neighbourhood. Figure 8 (left) shows a recursive partitioning approach with binary hierarchical partition structure compared with a K-way partitioning case (right).

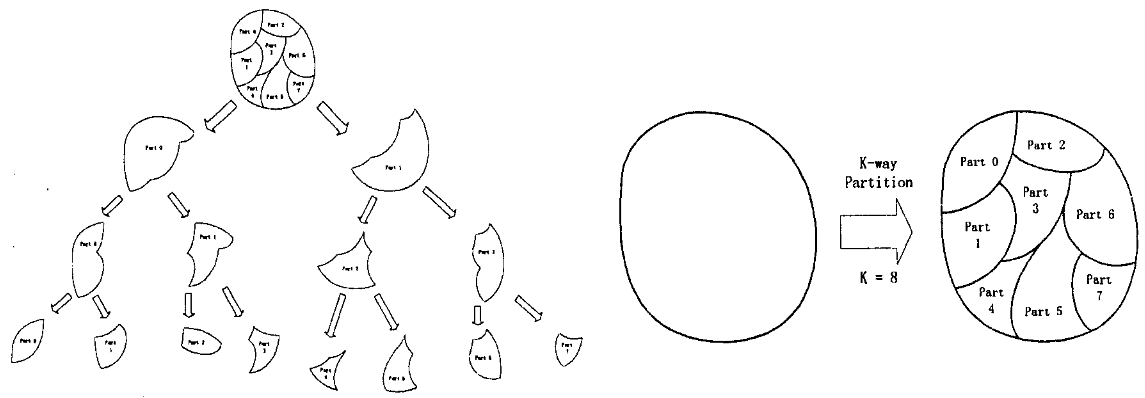

Figure 8. Binary partitioning (left) and K-way partitioning (right)

4.3 Cost model

Originally, the load balancing approaches did not include any cost estimations for partitioning and data migration [SH04B]. The first aim of load balancing is to divide the workload fairly across processors. And the second aim is to create as few edge cuts as possible. Some approaches such as the diffusion algorithm perform better in terms of partitioning/migration cost and worse where edge cuts are concerned, i.e. the quality of the distribution obtained.

Here we propose a model to evaluate our over-partitioning approach compared with partitioning from scratch. In our model, the application (1) executes computation and communication work for several iterations, then (2) performs load balancing for resource adaptation once, and (3) repeats the above two phases until completing all calculations. In addition, we do not take dynamic workload changes into consideration. That is, the overall workload of an application always remains unchangeable during all phases. As well, we have to assume the computation and communication phases are mostly synchronous in iterations.

The following arguments will be used for calculating the cost of load-balancing and evaluation the benefits of our ATOP approach:

- T_{Mig} : Migration time for partitioning from scratch
- $T_{OverMig}$: Migration time for over-partitioning
- T_{Par} : Partitioning/repartitioning time for partioning from scratch
- $T_{OverPar}$: Partitioning/repartitioning for overpartioning
- N_{iter} : Number of iterations between adaptations
- $N_{vert}(i)$: Number of vertices on *ith* processor
- P_{num} : Number of processors used in each adaptation step
- $T_{PerVert}$: Time used to simulate the computation overhead on each node
- T_{PerEdg} : Time used to simulate the communication overhead on each edge cut
- $T_{Comp}, T_{OverComp} = Max(N_{vert}(i)) * T_{PerVert}, i = [0, P_{num})$:

Computational time in each iteration, time used for performing the longest job on all processors

- $T_{Comm}, T_{OverComm} = T_{PerEdg} * \sum_i E_{cuts}(i), i = [0, P_{num})$:

  Overall communication time in each iteration, sum of the time used for communication through all edge cuts among processors

- $IMB = Max(N_{vert}(i))/((\sum_i N_{Vert}(i))/P_{num}), i = [0, P_{num})$:

  Imbalance among allocated processors: the maximum processor weight, $Max(N_{vert}(i))$, divided by the optimal weight, $(\sum_i N_{Vert}(i))/P_{num}$.

Thus, we calculate the following two formulas:

– Time saved during adaptation by using over-partitioning,

$S = (T_{Par} + T_{Mig}) - (T_{OverPar} + T_{OverMig})$

– Time for application communication cost caused by more edge cuts

$C = N_{iter} * ((T_{OverComp} + T_{OverComm}) - (T_{Comp} + T_{Comm}))$

$T_{PerVert}$ and T_{PerEdg} should be set according to the different application properties. If S > C, it is worth employing the over-partitioning; otherwise we should do a traditional partitioning from scratch. In addition to the trade-off between saving in adaptation and additional application (computation and communication) overhead caused by over-partitioning, we should investigate another important criterion – relative benefit, which depends on the interval time between two resource adaptations, i.e. the smaller the time interval, the higher the relative benefit of over-partitioning.

5 Experimental Results

5.1 Experimental Framework

We perform all experiments on our Horus cluster with 16 nodes each of which has two 2 Ghz Xeon processors (though we only allocate one process per node), 512 Mbyte memory, and 512 Kbyte L2 cache. The interconnection is Myrinet. The MPI library version is MPICH-GM 1.2.5.12bs, Zoltan is Version 1.52 and ParMETIS is Version 3.1.

We use benchmark graphs taken from the University of Greenwich Graph Partitioning Archive [GPA] as the test input graphs. The graphs are selected with different sizes and different properties, that is, different ratios of edges vs. vertices (see Table 4, where |V| is the number of vertices and |E| is the number of edges in the mesh). These graphs describe the graph structure only and do not have any weights attached to vertices or edges. In our experiments, we set all weights uniformly, though we could easily use different weights.

Table 4. Benchmark graphs with different properties

Graphs	|V|	|E|	Description
3elt	4720	13722	2D finite element mesh
wing_nodal	10937	75488	3D finite element mesh
4elt	15606	45878	2D finite element mesh
fe_sphere	16386	49152	Not Available
cti	16840	48232	Not Available
wing	62032	121544	3D finite element mesh
brack2	62631	366559	3D finite element mesh
finan512	74752	261120	Not Available

To explore the performance and distribution quality of our adaptation approach, we perform the following tests:

- Comparison between our over-partitioning approach and partitioning from scratch, both using K-way partitioning (a ParMETIS approach); furthermore, we compare with adaptive partitioning via the diffusion algorithm (also a ParMETIS approach) in time adaptation
- Adaptation between different numbers of allocated processors for space adaptation
- Use of different maximum resource allocations (8 or 16 processors) to test the scalability. Adaptation with the number of partitions set according to the maximum resource allocation
- Comparison of number of partitions set to the maximum allocation on 16 processors (using 16 partitions) and set to a much larger number (using 128 partitions)
- Comparison of migration-oriented and (approximate) structure-oriented allocation of partitions to processors
- Use of different weight combinations for time adaptation

To evaluate the performance, we employ the following measurements and metrics:

- Time for each step of adaptation as a total and differentiated into partitioning and migration cost
- Edge cuts per adaptation step in total
- Vertices per processor (to check imbalance)

5.2 Experimental Results for Space Adaptation

To examine the overall performance of our over-partitioning approach, we test our approach compared to partitioning from scratch with eight benchmark graphs described above which have different properties. In all following tables, the unit of measurement of time is in seconds. Table 5 shows the results from over-partitioning vs. partitioning from scratch, using 8 processors with 8 partitions. It includes the edge cuts in each adaptation step and the total overhead for four space adaptations in terms of time. In any case of space adaptation, we apply an initialization step which we show separately. In this initial step, we read the graph on 1 processor (i.e. in one process) and partition and distribute it for the first configuration.

Table 5. Adaptation time and edge cuts of over-partitioning (upper table) vs. partitioning from scratch (lower table) on 8 processors

Over-partitioning with 8 partitions

Graphs	**Overall edgecuts**				**Init Step**	**Time of Adaptation steps**
	4	**8**	**4**	**2**		
3elt	338	505	338	156	0.3	0.056
4elt	419	665	419	194	2.934	0.572
cti	2143	2282	2143	1141	3.069	0.612
fe_sphere	1076	1331	1076	704	2.666	0.645
wing_nodal	4911	5963	4911	3622	1.4	0.289
wing	2390	3261	2390	1858	202.33	25.8
brack2	5047	8528	5047	4069	204.09	26.85
finan512	405	648	405	324	295.95	49.1

Partitioning from scratch

Graphs	**Overall edge cuts**				**Init Step**	**Time of Adaptation steps**
	4	**8**	**4**	**2**		
3elt	297	471	259	92	0.298	0.18
4elt	410	643	364	171	2.997	1.039
cti	1253	2399	1182	415	3.744	1.666
fe_sphere	936	1395	878	490	3.062	2.283
wing_nodal	4213	6093	3759	1759	1.572	0.903
wing	2129	3081	2036	947	215.32	102.67
brack2	3163	8221	3113	747	222.96	88.83
finan512	324	648	324	162	328.71	199.82

Figure 9 is created from the data in Table 5. From these data, we observe that although the overhead in the initialization step for two approaches are almost the same, ATOP outperforms partitioning from scratch by up to 75% in terms of adaptation cost. At the same time, ATOP only moderately increases overall edge cuts.

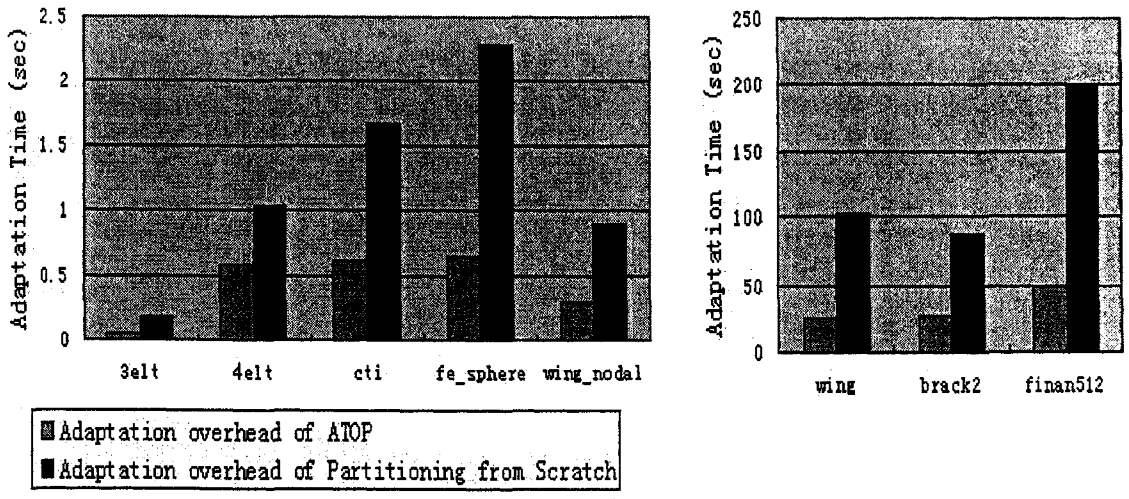

Figure 9. Adaptation overhead for different graphs on 8 processors

To test the scalability of our ATOP approach, we make similar experiments on an expanded allocation with a maximum of 16 processors. The upper and middle tables in Table 6 show the results from using over-partitioning vs. partitioning from scratch on 16 processors with 16 partitions. The results show good scalability for our ATOP approach, which again performs significantly better on any number of processors. For the graph "wing", however, the saving is now about 33%, whereas on 8 processors, it was about 25% but also the problem size is relatively smaller if running on up to 16 nodes. Figure 10 reflects such saving.

Table 6. Adaptation time and edge cuts of over-partitioning with 16 partitions (upper table), Partitioning from scratch (middle table), Over-partitioning with 128 partitions (lower table)

Over-partitioning with 16 partitions on 16 processors

Graphs	**Overall edge cuts**				**Init Step**	**Time of Adaptation steps**
	8	**16**	**4**	**2**		
3elt	543	896	326	147	0.267	0.064
4elt	912	1045	432	167	3.296	0.518
wing	4632	4897	4183	2274	210.667	20.263
brack2	11926	13434	9527	7868	222.85	22.6
finan512	1296	1296	1053	810	312.75	37.98

Partitioning from scratch on 16 processors

Graphs	Overall edge cuts				Init	Time of Adaptation
	8	16	4	2	Step	steps
3elt	473	882	252	117	0.298	0.176
4elt	662	1087	382	154	3.624	2.237
wing	3461	4781	2191	989	234.38	61.516
brack2	8875	13884	3074	746	245.17	56.51
finan512	648	1296	324	162	354.58	112.96

Over-partitioning with 128 partitions on 16 processors

Graphs	Overall edge cuts				Init	Time of Adaptation
	8	16	4	2	Step	steps
3elt	414	675	234	105	0.397	0.066
4elt	792	1232	514	253	4.076	0.414
wing	3906	5650	2657	1397	215.78	17.508
brack2	9182	15043	3964	1005	224.32	18.08
Finan512	784	1533	331	169	326.25	30.09

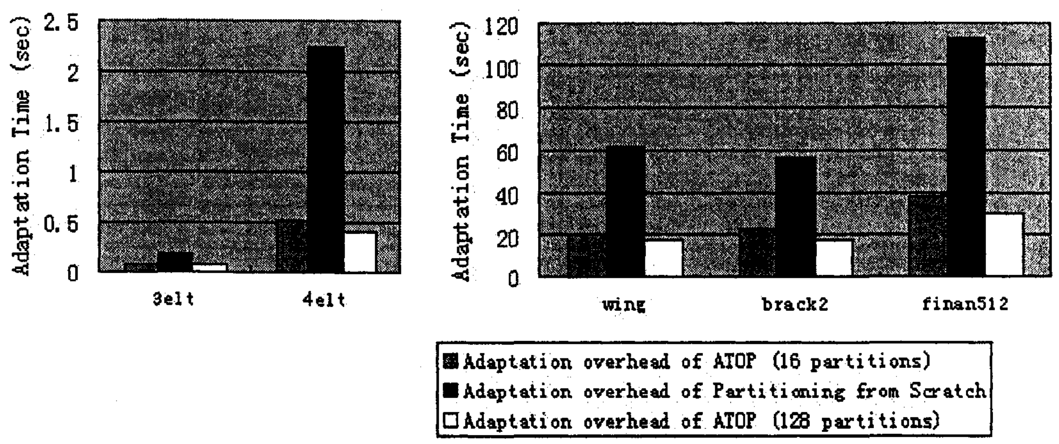

Figure 10. Adaptation time of ATOP with 16 partitions (left column), partitioning from scratch (middle column) and ATOP with 128 partitions (right column)

In addition to show the flexibility of ATOP, we compare the number of partitions set to maximum allocation on 16 processors with 16 partitions and set to a much larger number with 128 partitions. The lower table in Table 6 also demonstrates the effect from using such larger numbers of partitions. We increase over-partitioning on 16 processors from 16 partitions to 128 partitions. The results show that larger numbers of partitions do not significantly affect performance of partitioning and migration (the differences in edge cuts are minor and in some cases even the differences are on the positive side). (see lower table in Table 6 and Figure 10.) This is a promising result as larger numbers of partitions provide more flexibility in resource allocation (arbitrary numbers of processors chosen) and may—because being finer-grain—also increase cache locality.

Table 7. Adaptation time of over-partitioning vs. partitioning from scratch per adaptation step

Over-partitioning with 16 partitions

Graphs	Adaptation time on 16 processors						
	Init	8->16		16->4		4->2	
		Par_time	Mig_time	Par_time	Mig_time	Par_time	Mig_time
3elt	0.267	0	0.016	0	0.015	0	0.033
4elt	3.296	0	0.221	0	0.084	0.001	0.212
wing	210.667	0.005	5.902	0.005	1.664	0.005	12.682
brack2	222.85	0.004	5.786	0.005	2.567	0.005	13.628
finan512	312.75	0.015	12.473	0.006	3.375	0.006	22.149

Partitioning from scratch

Graphs	Adaptation time on 16 processors						
	Init	8->16		16->4		4->2	
		Par_time	Mig_time	Par_time	Mig_time	Par_time	Mig_time
3elt	0.298	0.048	0.02	0.024	0.018	0.036	0.03
4elt	3.624	0.822	0.336	0.359	0.285	0.165	0.27
wing	234.38	19.14	6.527	9.517	8.081	4.944	13.307
brack2	245.17	17.613	6.221	8.481	6.263	4.263	13.676
finan512	354.58	27.534	9.615	13.361	11.239	7.118	21.368

Table 7 and Figure 11 differentiate the adaptation cost per adaptation step for over-partitioning vs. partitioning from scratch, using 16 processors with 16 partitions. In both table 7 and figure 11, Par_time is partitioning time, and Mig_time is migration time. The results demonstrate that in each step of adaptation, our ATOP over-partitioning approach outperforms the partitioning from scratch by almost eliminating the partitioning overhead. From Figure 11, we find that the primary overhead of adaptation, i.e. partitioning cost (dark column), is almost zero in our over-partitioning approach, and the migration costs of the two approaches do not make a difference. Therefore, our ATOP over-partitioning approach always obtains better performance in each step of adaptation than the partitioning from scratch in terms of time cost.

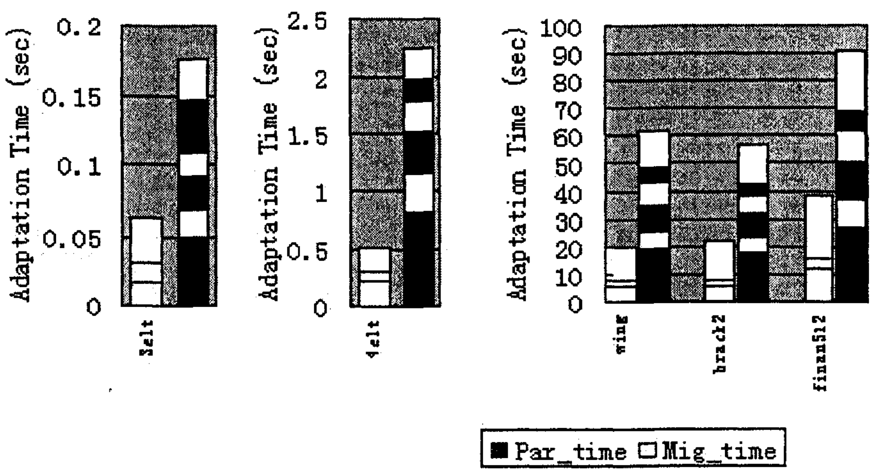

Figure 11. Accumulative adaptation overhead in different steps of over-partitioning with 16 partitions (left column) vs. partitioning from scratch (right column) for space adaptation

Table 8. Imbalance of the data distributions by using over-partitioning (upper table) vs. partitioning from scratch (lower table)

Over-partitioning on 16 processor

Graphs	**Workload per processor**											
	1->8			**8->16**			**16->4**			**4->2**		
	Max	**Min**	**IMB**	**Max**	**Min**	**IMB**	**Max**	**Min**	**IMB**	**Max**	**Min**	**IMB**
3elt	590	590	00.0%	295	295	00.0%	1180	1180	00.0%	2360	2360	00.0%
4elt	2090	1797	07.1%	1065	841	09.2%	4080	3674	04.6%	7852	7754	00.6%
wing	8090	7376	04.3%	4199	502	08.3%	15864	14778	02.3%	31390	30642	01.2%
brack2	8305	7347	06.1%	4276	383	09.2%	16501	14725	05.4%	32101	30530	02.5%
finan512	9348	9341	00.0%	4675	670	00.1%	18691	18687	00.0%	37378	37374	00.0%

Partitioning from scratch

Graphs	Workload per processor											
	1->8			8->16			16->4			4->2		
	Max	Min	IMB	Max	Min	IMB	Max	Min	IMB	Max	Min	IMB
3elt	592	585	00.3%	295	295	00.0%	1180	1180	00.0%	2370	2350	00.4%
4elt	2053	1875	05.2%	1053	795	08.0%	3936	3846	00.9%	7854	7752	00.7%
wing	8083	6981	04.2%	4201	285	08.4%	16338	15055	05.4%	31022	31010	00.0%
brack2	8558	7124	09.3%	4290	3273	09.6%	16969	15005	08.4%	31960	30671	02.1%
finan512	9345	9342	00.0%	4675	4668	00.1%	18689	18687	00.0%	37376	37376	00.0%

In addition to the edge cuts, the vertex distribution is another important factor to evaluate the workload redistribution. From the above cost model, the computational time per iteration step depends on the largest work chunk on all allocated processors. In this thesis, we use a matrix of IMB to describe the quality of data redistribution, which is the percentage of the maximum processor weight divided by the optimal weight. Table 8 shows the data distribution results with imbalance percentage (IMB) per adaptation step for over-partitioning vs. partitioning from scratch. We find that for the graph "3elt", ATOP is 0.3% worse than partitioning from scratch; whereas for the graph "wing", ATOP is 2.9% better. For the graph "brack2", the performance is different in each different adaptation step. ATOP is 3% better in the first step, but 1% worse in the fourth step. Therefore, ATOP is at least not worse than the traditional partitioning from scratch as regards the imbalance factor. In Table 8, max and min show the maximum and minimum number of vertices among the nodes.

Table 9. Adaptation time and edge cuts by using structure-oriented order (upper table) vs. migration-oriented order in over-partitioning (lower table) on 16 processors

Structure-oriented order

Graphs	Overall edge cuts				Init	Time of Adaptation steps
	8	16	4	2		
3elt	543	896	326	147	0.267	0.064
4elt	912	1045	432	167	3.296	0.518
wing	4632	4897	4183	2274	210.667	20.263
brack2	11926	13434	9527	7868	222.85	22.6
finan512	1296	1296	1053	810	312.75	37.98

Migration-oriented order

Graphs	Overall edge cuts				Init	Time of Adaptation steps
	8	16	4	2		
3elt	859	896	852	675	0.331	0.056
4elt	1023	1045	1013	645	4.708	0.42
wing	4573	4897	3920	2974	216.61	17.16
brack2	11793	13434	11793	9339	202.35	21.05
finan512	1134	1296	1134	810	318.25	26.15

We compare the edge cuts and adaptation overhead with two migration strategies: structure-oriented order vs. migration-oriented order. Table 9 shows the results obtained on 16 processors for ATOP, comparing structure-oriented order and migration-oriented order of partition allocation. We observe that using structure-oriented order generally decreases the overall edge cuts on all processors, i.e. provides better distribution quality. In some general cases, the decrease of edge cuts is around 10% to 50%. Using migration-oriented order, migration cost is slightly improved. Although using migration-oriented order to allocate workload among processors creates less adaptation overhead than using structure-oriented order, the overall advantage generated by using such strategy is insignificant. Because using structure-oriented strategy potentially creates better edge cuts, we have used it in all other tests.

5.3 Experimental Results for Time Adaptation

As mentioned in the above section, we employ much finer grains when performing over-partitioning in time adaptation, e.g. 128 partitions on an 8-processor cluster. Thus we can allocate the corresponding number of partitions on each processor according to the relative weights approximately. That is, using finer grains (or a larger partition number) we can create better data distribution after over-partitioning in time adaptation.

We compare the results of time adaptation on 8 processors, using over-partitioning with 128 partitions. In the initial step, the workload is allocated on the processors with the relative weights 2:2:2:2:1:1:1:1, and then changes following the sequence: 2:2:2:2:1:1:1:1 → 1:1:1:1:1:1:1: → 1:1:1:1:2:2:2:2 → 1:1:1:1:2:2:3:3. In the experiments, we employ two repartitioning algorithms: K-way partitioning and adaptive repartitioning by diffusion, to compare their performance. Since we do not change the number of processors, the diffusion approach can be used in time adaptation. Table 10 and Figure 12 are the results of the comparison, in which Ecuts are edge cuts, Part is partitioning time, and Migr is migration time.

Table 10. Adaptation time and edge cuts for time adaptation by using over-partitioning (upper table) vs. K-way partitioning from scratch (middle table) vs. diffusion adaptive partitioning (lower table) on 8 processors

Over-partitioning with 128 Partitions

Graphs	**Init**	**1:1:1:1:1:1:1:1**			**1:1:1:1:2:2:2:2**			**1:1:1:1:2:2:3:3**		
		Ecuts	**Part**	**Migr**	**Ecuts**	**Part**	**Migr**	**Ecuts**	**Part**	**Migr**
3elt	0.37	414	0	0.01	692	0	0.01	783	0	0.006
4elt	2.89	772	0	0.09	1250	0	0.15	1286	0	0.04
wing	221.91	3795	0.001	3.92	4986	0.001	4.58	4697	0.001	0.91
brack2	222.34	9410	0.004	3.724	14865	0.004	4.477	12975	0.003	0.926
finan512	347.65	660	0.005	4.328	2784	0.005	6.014	2607	0.004	0.999

K-way partitioning

Graphs	**Init**	**1:1:1:1:1:1:1:1**			**1:1:1:1:2:2:2:2**			**1:1:1:1:2:2:3:3**		
		Ecuts	**Part**	**Migr**	**Ecuts**	**Part**	**Migr**	**Ecuts**	**Part**	**Migr**
3elt	0.33	493	0.04	0.02	466	0.02	0.008	496	0.03	0.009
4elt	5.62	668	0.57	0.23	707	0.25	0.15	715	0.11	0.05
wing	226.93	3313	20.28	1.85	3345	17.69	11.8	3084	1.15	0.72
brack2	229.98	8002	21.24	4.556	7437	14.57	9.913	6917	1.12	0.978
finan512	354.52	648	18.998	3.186	1302	14.4	10.14	1409	1.704	1.496

RepartGDiffusion partitioning (adaptive repartitioning via diffusion)

Graphs	Init	1:1:1:1:1:1:1:1			1:1:1:1:2:2:2:2			1:1:1:1:2:2:3:3		
		Ecuts	Part	Migr	Ecuts	Part	Migr	Ecuts	Part	Migr
3elt	0.33	387	0.05	0.01	381	0.03	0.004	381	0.03	0.004
4elt	4.29	749	0.46	0.1	692	0.35	0.02	690	0.34	0.01
wing	225.94	3858	20.14	2.15	3599	17.03	0.04	3540	16.96	0.04
brack2	236.29	19761	12.67	0.248	19396	12.35	0.045	19357	12.33	0.04
finan512	358.48	6817	13.81	0.105	6485	13.72	0.095	6346	13.61	0.06

From the Table 10 and Figure 12, we find that over-partitioning always obtains better performance than partitioning from scratch (K-way) by down to 15% in terms of time cost. The additional number of edge cuts is moderate. Because the diffusion approach just takes the vertices on the border of partitions into consideration, the data distribution created by this approach is worst and its migration time is significantly better only for big graphs, i.e. "wing", "brack2" and "finan512". The specifications w1:w2:w3:w4:w5: w6:w7:w8 show the different weights per processor.

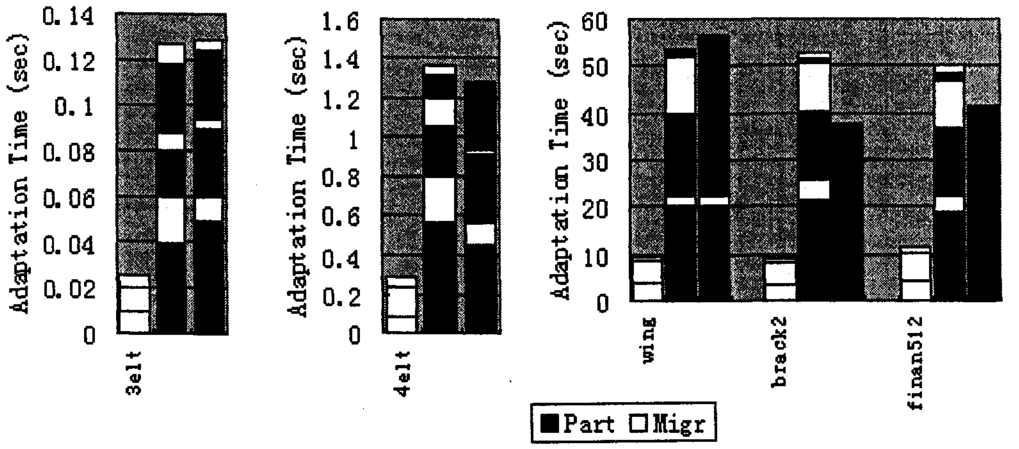

Figure 12. Adaptation time in each step of ATOP with 128 partitions (left column) vs. K-way partitioning from scratch (middle column) vs. diffusion adaptive partitioning (right column)

Table 11 shows the vertex distribution results after time adaptation for over-partitioning vs. K-way partitioning. The results demonstrate that the vertex distribution created by over-partitioning is at least not worse than the traditional partitioning approaches.

Table 11. Number of vertices per processor per adaptation step using over-partitioning vs. K-way partitioning on 8 processors

Over-partitioning with 128 partitions on 8 processors

Graphs	2:2:2:2:1:1:1:1							
	P0	P1	P2	P3	P4	P5	P6	P7
3elt	774	775	775	776	367	367	370	516
4elt	2567	2543	2567	2571	1212	1211	1215	1720
wing	10165	10137	10069	10200	4895	4813	4903	6850
brack2	10336	10214	10330	10142	4927	4934	4940	6808
finan512	12281	12289	12247	12300	5803	5818	5775	8239

Over-partitioning with 128 partitions on 8 processors

Graphs	1:1:1:1:1:1:1:1							
	P0	P1	P2	P3	P4	P5	P6	P7
3elt	591	590	589	591	591	588	590	590
4elt	1966	1945	1952	1946	1952	1943	1942	1960
wing	7719	7730	7620	7768	7795	7818	7757	7825
brack2	7907	7803	7850	7764	7779	7907	7809	7812
finan512	9363	9350	9374	9314	9341	9344	9348	9318

Over-partitioning with 128 partitions on 8 processors

Graphs	1:1:1:1:2:2:2:2							
	P0	P1	P2	P3	P4	P5	P6	P7
3elt	369	368	370	368	776	775	771	923
4elt	1226	1224	1216	1211	2559	2569	2550	3051
wing	4887	4780	4818	4832	10051	10236	10189	12239
brack2	4970	4901	4779	4984	10253	10132	10417	12195
finan512	5786	5915	5841	5858	12251	12319	12342	14440

Over-partitioning with 128 partitions on 8 processors

Graphs	1:1:1:1:2:2:3:3							
	P0	P1	P2	P3	P4	P5	P6	P7
3elt	332	331	333	333	663	665	994	1069
4elt	1105	1110	1089	1089	2210	2185	3281	3537
wing	4366	4342	4325	4320	8627	8734	13201	14117
brack2	4437	4386	4414	4425	8896	8695	13204	14174
finan512	5311	5194	5287	5280	10520	10497	15770	16893

Partitioning from scratch

Graphs	2:2:2:2:1:1:1:1							
	P0	P1	P2	P3	P4	P5	P6	P7
3elt	787	787	786	787	391	394	394	394
4elt	2601	2694	2470	2680	1368	1313	1237	1243
wing	9039	10897	10913	10898	5472	5475	3810	5528
brack2	11154	10422	10145	11154	3657	5155	5723	5223
finan512	11823	11694	11979	13703	6020	6852	6601	6080

Partitioning from scratch

Graphs	1:1:1:1:1:1:1:1							
	P0	P1	P2	P3	P4	P5	P6	P7
3elt	590	590	590	590	590	590	590	590
4elt	1976	1884	2057	1762	1781	2018	2112	2016
wing	8101	8094	8078	7555	7547	7549	7553	7555
brack2	8096	8570	8522	7920	7514	8313	5633	8063
finan512	9343	9343	9344	9343	9345	9344	9345	9345

Partitioning from scratch

Graphs	1:1:1:1:2:2:2:2							
	P0	P1	P2	P3	P4	P5	P6	P7
3elt	394	394	394	382	789	789	789	789
4elt	1356	1408	1243	1126	2543	2690	2607	2633
wing	4951	5569	4910	5682	9856	10143	10465	10456
brack2	5498	5445	4556	5678	9460	11026	10976	9992
finan512	5793	6226	6851	6851	13090	12581	11680	11680

Partitioning from scratch

Graphs	1:1:1:1:2:2:3:3							
	P0	P1	P2	P3	P4	P5	P6	P7
3elt	338	337	337	338	671	675	1012	1012
4elt	1109	1034	1162	1057	2290	2446	3314	3194
wing	3439	4436	4716	3937	9218	9746	13982	12558
brack2	4721	3996	4566	3153	9641	8537	14450	13567
finan512	5133	5510	4981	5869	9357	11209	16345	16348

5.4 Scenario Study and Experimental Summary

To demonstrate the implication of different cost factors, we present a scenario based on the graph "wing". The number of allocated processors changes from 16 to 4 (in step 16→4).

In the first case, we assume that the computation cost per vertex is $T_{PerVert} = 0.075ms$, and the communication cost per edge cut is $T_{PerEdg} = 0.005ms$. The iteration steps between two space adaptations are $N_{iter} = 100$ .Using the formulas for S and C from Section 3, we calculate that in the adaptation:

$S = (9.517 + 8.081) - (0.005 + 1.664) = 15.929\sec = 15929ms$,

$C = 100 * ((15864 * 0.075 + 4183 * 0.005) - (16338 * 0.075 + 2191 * 0.005)) = -2559ms.$

The time between these adaptation steps is

$T_{iter} = 100 * (15864 * 0.075 + 4183 * 0.005) = 121.07\sec \approx 2\min.$

Since $S \geq C$, it is worth applying over-partition. The saving is 18.5 sec which is about 15% of 2 min time interval.

In another case where weight of edge cuts is dominant, we set $T_{PerVert} = 0.1ms$, $T_{PerEdg} = 0.5ms$, and $N_{iter} = 33$. Then $S = 15929ms$, $C = 31303.8ms.$and the time between this adaptation is $T_{iter} = 121.37\sec \approx 2\min$. The adaptation cost is 15.4 sec (12.8%) worse than partitioning from scratch of 2 min time interval.

From the above scenarios, we conclude as a rough approximation that if the communication cost per edge cut is dominant for an application, or the interval is very long, partitioning from scratch is likely to gain better performance, whereas over-partitioning will obtain better performance under the remaining conditions.

6 Conclusion and Future Work

We have presented a flexible approach for adaptive resource allocation in both space and time dimension. Our ATOP approach employs a standard load-balancing library (Zoltan) to accomplish the data redistribution. Zoltan enables parallel application developers to keep the original data representation of the application. We provide two possible approaches: partitioning from scratch and over-partitioning. The latter significantly reduces adaptation cost by saving partitioning time and only slightly decreases the distribution quality of the data. Our results have shown improvements by up to a factor of 4 in space adaptation and of 6.5 for time adaptation. Over-partitioning appears to be especially useful if there are frequent adaptations as may be more likely in time-shared environments.

Defining the callback functions for Zoltan, however, proved to be a bit tedious—thus, as future work our AlphaMeta group will provide a higher-level automatic interface as part of our own load balancing library. Furthermore, our group plans to integrate allocation of partitions with the internal multi-level structure by making the latter explicitly available, and also plans to improve migration cost by hiding the communication latency. Another potential project is to integrate our work with the typical dynamic load balancing frameworks which focus on handling the application-internal imbalance at run-time.

Most importantly, our ATOP load balancing approach is one link of a whole chain that is formed by our AlphaMeta Lab. Our ATOP will be integrated with an adaptive MPI runtime environment, a job scheduler, and a resource monitor to provide a flexible and efficient platform for parallel applications development.

Appendix: References

[BDH+02] Erik Boman, Karen Devine, Robert Heaphy, Bruce Hendrickson, William F. Mitchell, Matthew St. John, Courtenay Vaughan. *Zoltan: Data-Management Services for Parallel Applications User's Guide.* Sandia National Laboratories 2002

[Bok81] S. H. Bokhari, *One the Mapping Problem.* IEEE Trans. Comput., Vol. C-30, No.3, pp 207-214.

[BOS96] Rupak Biswas Leonid Oliker Andrew Sohn *Global Load Balancing with Parallel Mesh Adaptation on Distributed-Memory Systems* NAS-96-013 1996

[CB03] W. Cirne and F. Berman. *When the Herd is Smart: Aggregate Behavior in the Selection of Job Request.* IEEE Trans. on Parallel and Distributed Systems, Vol. 14, No. 2, Feb. 2003.

[CRS+99A] A. Cortes, A. Ripoll, M.A. Senar, P. Pons, E. Luque *On the Performance of Nearest-Neighbours Load-balancing Algorithms in Parallel Systems* Seventh Euromicro Workshop on Parallel and Distributed Processing 1999

[DHB+00] Karen Devine, Bruce Hendrickson, Erik Boman, Matthew St.John and Courtenay Vaughan *Design of Dynamic Load-Balancing Tools for Parallel Applications.* ICS 2000 Santa Fe New Mexico

[GK98] Hans Georg Galbas and Otto Kolp *Dynamic Load Balancing in Crashworthiness Simulation.* Proc. 3rd Internat. Conf. on Vector and Parallel Processing (VECPAR '98), Porto, Portugal, June 1998

[GPA] The online Graph Partitioning Archive, http://staffweb.cms.gre.ac.uk/~c.walshaw/partition, retrieved July 2004

[HLK03] Chao Huang, Orion Lawlor, L. V. Kale *Adaptive MPI* Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03) 2003

[KK98] G. Karypis and V. Kumar. *Multilevel k-way partitioning scheme for irregular graphs.* Journal of Parallel and Distributed Computing, 1998

[KKD02] L. V. Kale, Sameer Kumar, and Jayant DeSouza *A Malleable-Job System for Timeshared Parallel Machines* 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2002), May 21-24, 2002, Berlin, Germany. 2002

[KSK03] George Karypis, Kirk Schloegel, and Vipin Kumar. *ParMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering Library Manual*, 2003

[LRC+95] E. Luque, A. Ripoll, A. Corts, T. Margalef *A Distributed Diffusion Method for Dynamic Load Balancing on Parallel Computers* Proc. of the EUROMICRO Workshop on Parallel and Distributed Processing. San Remo, Italy 1995

[NAB+96] J. Nagib, C. Arebe, A. Beguelin, and B. Lowekamp *Dome: Parallel programming in a distributed computing environment* Proceedings of the International Parallel Processing Symposium 1996

[SH04A] A. C. Sodan and X. Huang. *Adaptive Time/Space Sharing with SCOJO.* Proc. Internet. Conf. on High Performance Computing Systems (HPCS), Winnipeg, May 2004.

[SH04B] A. C. Sodan, L. Han, *ATOP—Space and Time Adaptation for Parallel and Grid Applications via Flexible Data Partitioning.* In process.

[SKK00] Kirk Schloegel, George Karypis and Vipin Kumar. *A Unified Algorithm for Load-balancing Adaptive Scientific Simulations.* Proc.

IEEE/ACM Supercomputing Conf. (SC2000). Dallas/TX, USA, Nov. 2000.

[SKK03] Kirk Schloegel, George Karypis, Vipin Kumar *ParMETIS Parallel Graph Partitioning and Sparse Matrix Ordering Library, user's manual*, University of Minnesota, August 2003

[SKK93] Kirk Schloegel, George Karypis, Vipin Kumar *A Multi-level Diffusion Methods for Dynamic Load Balancing* Parallel Computing 1993

[SKK97] K. Schloegel, G. Karypis, and V. Kumar *Multilevel diffusion schemes for repartitioning of adaptive meshes* Journal of Parallel and Distributed Computing 1997

[TVC+02] Viviane Thome, Daniela Vianna, Roberto Costa, Alexandre Plastino, Otton T. da Silveira Filho *Exploring Load Balancing in a Scientific SPMD Parallel Application* 2002 International Conference on Parallel Processing Workshops (ICPPW'02) 2002

[WA02] Barry Wilkinson, Michael Allen, *Parallel Programming Techniques and applications*. Pearson education north Inc. 2002

[XH90] Jian Xu, Kai Hwang *Heuristic Methods for Dynamic Load Balancing in A Message-Passing Supercomputer* IEEE, Proceedings of the 1990 conference on Supercomputing 1990

[XL97] Chengzhong Xu and Francis C.M.Lau *Load-balancing in Parallel Computers Theory and Practice*, Boston 1997

Vita Auctoris

Name:		Lin Han
Place of Birth:		Beijing, China
Education:		
	2002-2004	M.Sc., Computer Science University of Windsor Windsor, Ontario, Canada
	1991-1995	B.Sc., Computer Science Beijing Jiaotong University Beijing, P.R.China
Working Experience:		
	1995-2002	Software Engineer & Project Manager SinoRails Info. Tech. Corp. Beijing, P.R.China