AN ADAPTIVE JOB SCHEDULER FOR TIMESHARED PARALLEL MACHINES

BY

SAMEER KUMAR

B. Tech. Indian Institute Of Technology Madras, 1999

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2001

Urbana, Illinois

Table of Contents

Chapter 1

Introduction

With the emergence of the grid [6] all the massive power generation centers would form unified grids to supply and meet the computational demands of the users. The grid, as the name might suggest provides a secure and coordinated resource sharing mechanism among dynamic collections of individuals and institutions, which comprise the grid. The institutions here would mainly consist of the super-computing centers which would like to disperse computational power to the users. In this scenario users would remotely submit jobs to these grids and view the outputs of their parallel programs on their desktops. Large parallel systems that form the grid could either be tightly coupled systems like Cray T3E, Origin 2000 [13, 12] etc, or be parallel clusters, like Turing, PSC TCS1 [16, 14] etc., connected by high bandwidth low latency networks.

Access to these parallel systems would be controlled by queueing systems which would handle the users requests and run their jobs on the parallel systems. The main purpose of these queuing systems would be to maximize the performance of the parallel systems. The common metrics, on which the performance of the parallel systems is measured, are system utilization and job response time. For commercial systems the major metric would be cumulative profit from the jobs accepted.

With current parallel jobs and current queuing systems it is often the case that systems are underutilized with jobs in the queue. This is mainly because jobs submitted to such systems are rigid, and their resource requirements conflict with those of other jobs. Consider the following scenario on a parallel system with 128 processors: Job 1 arrives in the system with a processor request of 64 processors and is started, some time later before Job1 finishes, Job2 arrives and requests 80 processors and is hence queued till the Job1 finishes. Thus despite having a non empty queue the system has 64 idle processors. Thus using current Queueing Systems [15, 4] may result low system utilization, as they can handle only rigid jobs.

In this thesis we try to provide an effective solution to the above mentioned problem. First we define an Adaptive Job to be a parallel job which can change the number of processors it is running on at runtime. Thus in a system which supports such adaptive parallel jobs the scenario described earlier would not lead to a loss of compute power. When Job2 arrives Job1 can be shrunk to 40 processors and Job2 can be started on the remaining 80 processors, or Job2 can be started on 64 processors and later be expanded to 80 or the complete set of 128 processors. This would depend on the scheduling strategy and the QOS requirements of the job. There is also a need for an Adaptive Job Scheduling system to manage these adaptive jobs and maximize the system utilization and job throughput and also improve the mean job response time.

In this thesis we use the Charm runtime system to develop adaptive jobs. This runtime system is versatile and currently supports MPI and Charm++ adaptive programs. Existing programs written in Charm++ and MPI can also be ported with ease. We have also developed an Adaptive Job Scheduling system which manages these adaptive jobs and the simulations and experiments on the Turing Linux Cluster [16] provide encouraging results.

1.1 Adaptive Jobs

An *adaptive job* is a parallel program which can increase (expand) or decrease (shrink) the number of processors it is running on at runtime. Traditional parallel programs are incapable of such adaptive behavior. We have developed a runtime system, on top of the Charm runtime system, that can make parallel programs in popular languages like Charm++ and MPI adaptive.

The Charm runtime system maps a parallel program to message driven objects. These objects can also be made migratable. To expand/shrink a job the objects are moved from the current set of processors to a new and larger/smaller set of processors.

To make an MPI program adaptive the program has to be linked with the Adaptive MPI (AMPI [1]) library. AMPI programs consist of a large number of *virtual* (or logical) MPI *processes*, implemented as user-level threads. The number of such threads is typically much larger than the number of processors. User programs are not aware of the physical processor on which each thread is running, and communicate using the rank of the corresponding MPI virtual process only. These threads are mapped to objects and can be made migratable and adaptive jobs can be implemented as described above.

There are many technical challenges that have to be overcome to develop such an adaptive runtime system. These will be described in detail in Chapter 2.

1.2 Adaptive Job Scheduling System

An adaptive job scheduler manages adaptive jobs and changes the number of processors each job is running on according to the system load. When the load is high the jobs are run on fewer processors and when the load is low the jobs can be run on their maximum number of processors. This also depends on the scheduling strategy. The Adaptive Job Scheduler consists of the following three components.

1. Job Manager. This component manages all the jobs and communicates their processor allocation maps to them. It also frees the job's resources when it finishes.

2. Strategy. The strategy distributes the processors and other resources among the jobs. The aim of the strategy could be to maximize system utilization, cumulative profit from the jobs etc.

3. Database. The scheduler logs all its activities into a database.

The details of the *Adaptive Job Scheduler* and its components are presented in Chapter 3.

1.3 Related Work

Initial efforts towards dynamic processor reconfiguration were mainly in the direction of dynamic process migration and load-balancing. Condor [9] supports runtime migration of a process from one workstation to another through check-pointing. This system only migrates sequential programs executing on one processor. Condor also supports the concept of negotiation, where each client bids for certain units of time.

Dome [11] is an object oriented distributed framework where applications are load-balanced by migrating objects from heavily loaded processors to lightly loaded ones. Dome does not explicitly support the concept of shrink and expand.

The ability to shrink and expand the number of processors allocated to a job was implemented by AMP (Adaptive Multiblock PARTI) framework [5]. This framework runs SPMD Fortran programs. Each processor that the program is started on can run an active process or a skeleton process. The active processes have all the load and the skeleton processes run in the background. Unlike our system the trigger to change the set of processors comes from the application. To expand, the load is moved from the active to the skeleton tasks. During a shrink or an expand the whole data is redistributed to the new set of active processors and the communication schedules are updated. The data redistributed is mainly the shared arrays.

DRMS [10] is another system that provides SPMD programs to reconfigure themselves dynamically. If the number of tasks in the program is changed after an SOP (Schedulable and Observable Point), then all the global data is redistributed. The global data is an array of basic data-types which is ordered as block, block-cyclic etc. The DRMS scheduler [17] tries to minimize the average response times of the jobs and also the response time variance. Care is taken to prevent starvation of small jobs.

DRMS (like AMP) is mainly useful for data-parallel applications. The redistribution of data, divides the data evenly among all the processors in use, but this does not ensure that all the processors being used by the job are load-balanced.

Our framework differs from these projects in its broader applicability, (i.e. we can handle any MPI program, and our strategies are well suited for even irregular problems) and its use of measurement-based load balancing, which is more accurate than equal redistribution of data.

1.4 Thesis Outline

Chapter 2 introduces the basic concepts involving adaptive jobs and also describes the Charm runtime system which is used to write adaptive jobs. Next we describe the *Adaptive Job Scheduling System* and its components in detail in Chapter 3. Chapter 4 describes simulation and experimental results of the performance of the *Adaptive Job Scheduling System.* Finally the conclusion and the future work are presented in Chapter 5.

Chapter 2

Adaptive Jobs

An *adaptive job* is a parallel program which can dynamically (i.e. at run-time) shrink or expand the number of processors it is running on, in response to an external command. The number of processors can vary within bounds specified when the job is started.

All adaptive jobs have the following resource requirements

1. *minpe*, the minimum number of processors requested by the job. This would be governed by the memory requirements of the job.

2. *maxpe* is the maximum number of processors on which the job can run. This is related to the speedup of the job. Users would like their job to run at the maximum speed, and if the speedup ceases to become monotonically increasing at some point then maxpe can be used to restrict the number of processors allocated to the job.

3. *profit*, the amount paid by the user if the system executes the job.

4. *deadline*, the deadline before which the job should be finished.

The following is the system job class which is used to store all the data related to a job. The fields are self explanatory and comments have been provided for further explanations. The *Adaptive Job Scheduler* maintains two lists of this data structure, namely, (i) waitq which is the job queue, (ii) runq which is the list of running jobs. Details will be presented in Chapter 3.

```
class Job{
 public:

    unsigned int port,ip,pid;   // The Port and the IP for CCS.
    int type;                   // The type of the job, charm, mpi, uni.
    int min_proc, max_proc;     // The minimum and maximum number of processors.
    int num_allocated_proc;     // The number of Processors allocated to the Job.
    char argbuf[MAX_BUF];       // Job Arguments.
    char bit_map[MAX_PROC];     // The Job processor Map.
    char *working_directory,*Stdin, *Stdout, *Stderr;
    int id;                     // Job Identifier.
    double startTime, arrivalTime;
    double profit;              // Profit from the job.
    double deadline;            // Relative Deadline.
       .
       .
       .
};
```

The Adaptive Jobs use the Charm runtime system. We will first describe the Charm runtime system in the next section, followed by a brief description of the MPI implementation of the parallel adaptive jobs. We will then describe the mechanism by which adaptive jobs operate and also present some performance results as a proof of concept for the adaptive jobs.

2.1 Charm System Overview

Charm++ is an asynchronous message passing language with data driven objects. The parallel program is mapped to a large number of parallel objects which communicate with each other by passing messages to each other. The messages invoke entry methods on the destination objects. The invocation is asynchronous which enables overlapping of computation and communication. The memory model is assumed to be distributed though efficient message passing primitives are provided for shared memory systems.

The objects are C++ objects which have local and entry methods. The entry methods of the objects are accessed through proxies. Charm++ also has an IDL similar to CORBA in which all the objects and their entry methods are defined. The Charm compiler then generates the code for the proxies from the IDL.

Arrays [8] are a class of objects that can be dynamically created and destroyed and are accessed by a unique array index. The arrays are managed by an array manager which directs messages to the array elements. The array manager is also responsible for creating and destroying array elements. The array manager can also migrate these array elements between processors.

The Charm system also provides a load-balancer [3] which balances the load between all the processors. The load-balancer can be centralized or distributed. The load-balancing step is asynchronous and application invoked. First all the array elements must call a synchronization barrier and then the load-balancer collects the load data from all the processors and load-balances them according to a statically linked strategy[1]. The strategy re-maps the objects among the processors and then the load-balancing framework migrates the necessary objects to their destination processors.

Currently various load-balancing strategies have been implemented. Some them are Random, Greedy, Metis etc. Random strategy randomly throws elements from heavily loaded processors to lightly loaded ones. Greedy strategy moves load from the heaviest processor to the lightest one. Metis uses the metis graph partitioning strategy to partition the load graph (where objects that communicate with each other are connected by edges) and allocates these partitions to the processors, thus ensuring that objects that communicate with each other are on the same processor.

The Charm system also provides a client server interface (CCS) which allows external programs to communicate with the parallel program. The external program makes a TCP connection to the parallel program and sends it a message and a handler id. The Charm system handles this message and invokes the handler with the data in the message.

2.2 Adaptive MPI Programs

Traditional MPI jobs, using a conventional implementation of MPI, are incapable of such adaptive behavior. We use an adaptive implementation of MPI (AMPI [1]) to dynamically change the set of processors being actively used by a job. To use AMPI, a Fortran 90 MPI program does not have to be changed at all. It is preprocessed by a source-to-source translator, and linked with the AMPI library, instead of the usual MPI library. C based MPI programs have to be modified, but the modification is simple and mechanical: All global variables must be encapsulated in a global dynamically allocated structure. This is necessary because AMPI allows multiple virtual processes per processor and each of them runs as a separate user level thread. Hence each process

[1] Charm++ also provides a Continuous Load-balancer along with the above mentioned periodic load-balancer

must have its global variables separate from the others. For Fortran90, our preprocessor makes the necessary transformation. Similar translation can also be done for C programs, with additional compiler-preprocessor support.

AMPI [1] programs, as mentioned above, consist of a large number of *virtual* (or logical) MPI *processes*. The number of such threads is typically much larger than the number of processors. User programs are not aware of the physical processor on which each thread is running, and communicate using the rank of the corresponding MPI virtual process only. This virtualization provides the system the ability to dynamically adapt its behavior. The Charm runtime system [7] maps the threads and objects to physical processors under the control of a load-balancer. The load-balancing framework keeps track of the load presented by each thread and object, and when triggered by either an internal or external trigger, redistributes the threads and objects to balance the load. AMPI uses a sophisticated scheme to permit migration of user level threads, as described in [1].

2.3 The Adaptive Job Framework

The Adaptive Job Framework has been implemented as a module in Charm. The module defines the *processor map handler* which is registered in the initialization call. The *processor map handler* is invoked by the Adaptive Job Scheduler to communicate the processor map to the parallel program. The handler then passes the processor map to the load-balancing framework. In the next load-balancing cycle the objects are migrated form the passive processors to the active processors in the processor map.

To write an adaptive program the users must include this module and call the initialization function. The Charm runtime system provides both periodic and continuous load-balancing. If periodic load-balancing is being used then the period of the load-balancer should be of the order of a few minutes. Large periods would result in a delayed response to the new bitmap. This could make two parallel programs share the same processor which is undesirable.

To enable MPI and Charm++ programs to be adaptive, we modified the load-balancer so that it takes into account the set of processors allocated to a job. We use the client-server interface (CCS) to allow the *Adaptive Job Scheduler* to communicate the processor map to the *processor* *map handler*. The handler communicates the new processor map to the load-balancing module. The load-balancer then triggers a migration phase to move the threads (in MPI programs using AMPI) and Array Objects (in Charm programs) out of the deallocated processors.

This is an easy but inefficient way to implement the adaptive job framework. This is because the new job and the old job share the processors till the next load-balancing operation in the old job. A better implementation of making the load migration and load-balancing independent is being worked on.

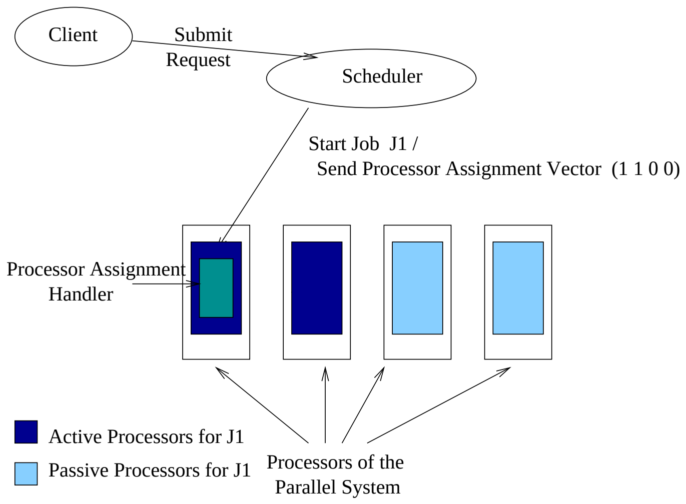
Client
Submit Request
Scheduler
Start Job J1 / Send Processor Assignment Vector (1 1 0 0)
Processor Assignment Handler
Active Processors for J1
Passive Processors for J1
Processors of the Parallel System

Figure 2.1: System Components.

A skeleton process is left behind on each deallocated processor to forward messages meant for objects/threads that were previously housed on that processor. The load on the skeleton processor consists of a transient period of forwarding messages, and a periodic (but nominal) participation in global operations such as reductions and load-balancing.

Two natural questions arise about the overhead of this method: (i) How quickly can we shrink (or expand) a job? And (ii) How much interference do the residual processes cause to the performance of another job on the same processor? We now describe experimental results to quantitatively answer these questions.

2.4 Performance

We conducted several experiments on our system by running a large molecular dynamics simulation, which is a simplified version of NAMD[2]. The program simulates more than 400,000 atoms, and has a total memory requirement of about 44.8 MB. We ran the job on the Turing Linux Cluster [16], which (at the time) consisted of about 192 dual 550 MHz Pentium III CPUs connected by both Ethernet and Myrinet. We obtained the following average times for shrinking the job.

Processors before and after shrink	Time on Ethernet, s	Time on Myrinet, s
8 to 4	1.477	.435
16 to 8	1.164	.362
32 to 16	1.315	.330
64 to 32	2.802	N.A.

Table 2.1: Shrink Time for MD Program (400K atoms, 44.8 MB)

As shown in Table 2.4, the reconfiguration time is quite small. Hence the overhead due to shrink and expand of jobs at each scheduling decision, which would typically occur once in several minutes, is negligible.

When shrunk, the job leaves a residual process on the processors it vacates as described in Section 2.3. Figure 2.2 shows this load on one of the processors after the job has been vacated. This load is zero most of the time but has periodic peaks of about 2 percent. Figure 2.3 demonstrates that the load does not interfere with another job. It shows the processor utilization of two jobs. When Job2 arrives, first the scheduler shrinks Job1 and then starts Job2. In this way the shrinking of Job1 is overlapped with the startup time of Job2 (when the load of Job2 is low) and processor sharing between the jobs is minimized. When Job2 finishes, the scheduler asks Job1 to expand. Thus, despite the presence of Job1, Job2 gets almost all the CPU.

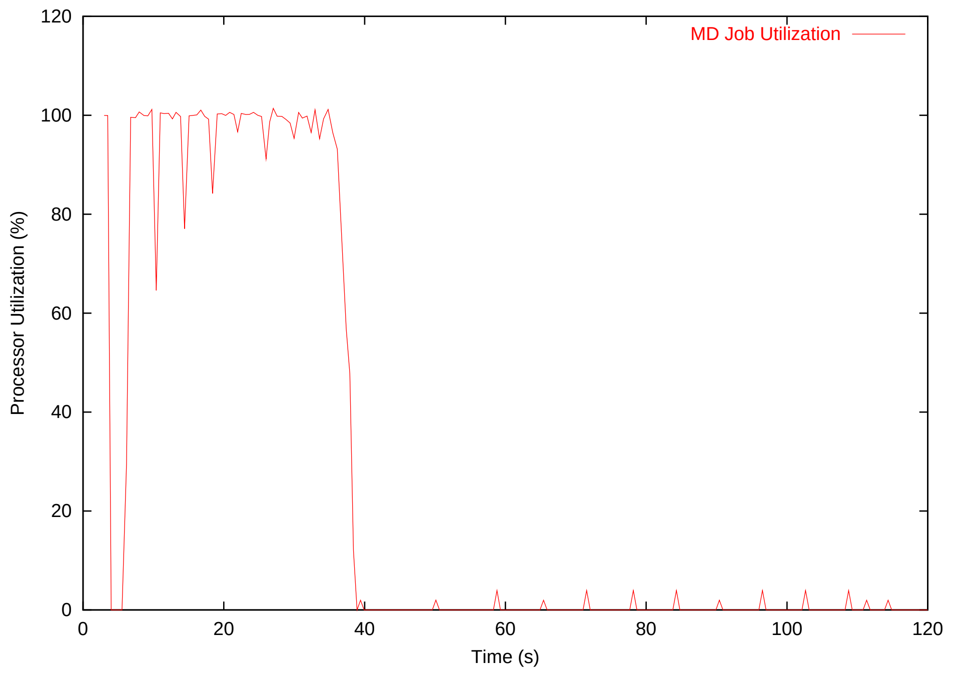

Figure 2.2: Residual Load after Shrinking.

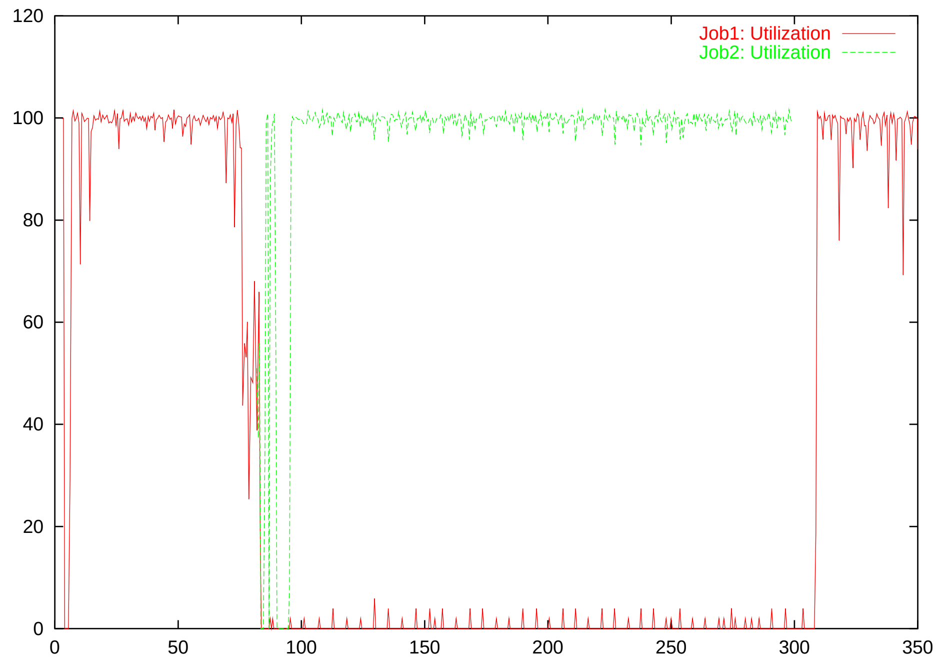

Figure 2.3: Proof of Concept of Shrink/Expand.

Chapter 3

Adaptive Job Scheduling System

To manage the Adaptive Jobs we have developed an *Adaptive Job Scheduling System.* The scheduling system is modular, multi-threaded and reliable. The scheduler is aware of the adaptive nature of the jobs and can use it to maximize the utilization of the system or minimize the response time or maximize the profit of the system. This depends on the scheduling strategy that is plugged into the system. The adaptive job scheduling system has three basic components, (i) Job Manager, (ii) Strategy, (iii) Database.

3.1 Job Manager

The *Job Manager* is invoked at two control points where the state of the scheduler changes. These are (i) Job Arrival, (ii) Job Termination. At each of these control points the Job Manager calls the strategy and can do any of the following, (i) start a new job, (ii) checkpoint and suspend a running job, (iii) shrink a running job to fewer processors, and/or (iv) expand a running job to more processors.

The Job Manager is multi-threaded and has four basic components. The components are independent of each other and handle events that occur at different frequencies. Hence for maximum performance the four components are implemented as separate threads. They are (i) Accept Thread, (ii) Execute Thread, (iii) Remove Thread, (iv) Nodelist manager.

3.1.1 Accept Thread

The *Accept Thread* accepts job requests from the clients. It is implemented as an independent thread that sleeps on a TCP socket through the *accept()* system call and is awakened when a new client connects to the scheduler. The request is sent from the client as a null terminated string. The request contains the name and path of the executable to run, its working directory, its arguments, the minpe and the maxpe, and the standard output and standard error file paths. The request could also be a command, which could kill an existing job or ask the scheduler the current state of the system.

The *Accept Thread* parses the arguments of the request and creates a job record and stores the job record in the database. If the request is a command it executes the command. If the request is a job the *Accept Thread* sets the *EVENT_OCCURED* flag so that the *Execute Thread* can be awakened.

3.1.2 Execute Thread

The *Execute Thread* is awakened when the *EVENT_OCCURED* flag is set. Currently this implemented as a polling mechanism in which the *Execute Thread* wakes up every two seconds and polls the flag. If the flag is set the database is queried for the *waitq* (the the list queued jobs) and the strategy is called and the *waitq* and *runq* are passed to it.

The strategy schedules some or all of the jobs in the waitq and reallocates the processors among all the scheduled and the running jobs. The *Execute Thread* then sends the new processor maps to all the running jobs and starts the scheduled ones on their current processor maps.

Currently three types of jobs are supported by our system, (i) MPI parallel jobs, (ii) Charm++ parallel jobs, (iii) Single Processor jobs. These jobs have their own starting programs which are *mpirun*, *charmrun* and *rsh* respectively. To start a job the execute thread forks a new process, changes its current directory to the directory given in the request, closes the stdout and stderr file descriptors, and opens the files requested by the user. After these initialization steps the new process execs a shell script which contains the starting program, the job executable and its arguments.

3.1.3 Remove Thread

The *Remove Thread* periodically (every two seconds) awakens itself and checks the status of the jobs in runq. If any job has terminated, the *Remove Thread* moves the processors allocated to the job to a free list, closes all its file descriptors, deletes its nodelist file and sets the *EVENT_OCCURED* flag. This would awaken the *Execute Thread* which would call the strategy and allocate the free processors to the queued and running jobs.

3.1.4 Nodelist Manager Thread

The *Nodelist Manager* is responsible for managing the system nodelist from which the nodelists for the parallel jobs are generated. It maintains a map for all the node id's, which are bits in the processor map, to the host names of the machines in the parallel cluster. It also periodically (every 30 seconds) polls the machines to see if they are alive and if any node goes down it replaces that node with a redundant node.

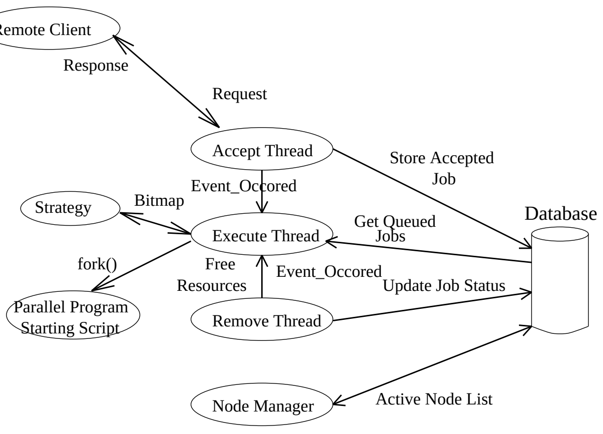

Figure 3.1: Adaptive Job Scheduling System Components

Figure 3.1 shows the interaction between the different components of the *Adaptive Job Scheduling System*. When the client request arrives it is handled by the *Accept Thread* and if the job request is accepted it is stored in the database and the *EVENT_OCCURED* flag is set. When the *Execute Thread* awakens it requests all the queued jobs from the database and invokes the strategy. The strategy reallocates the processors among the running and queued jobs and returns the new processor allocation. The *Execute Thread* will then start the jobs by forking and execing the *start* *shell script* of the job.

If a job finishes the *Remove Thread* will free all its resources and set its status in the database. It also sets the *EVENT_OCCURED* flag. The nodelist manager periodically keeps checking the nodes and maintains the list of all working nodes.

3.2 Scheduling Strategy

The *Scheduling Strategy* makes decisions on which jobs to schedule and on how many processors to be allocated to each job. We have implemented two types of strategies which are (i) Performance Driven Allocation, (ii) Profit Driven Scheduling.

Before going into the details of the different strategies we will first describe the API through which any strategy can be written and plugged into the system.

3.2.1 Scheduling Strategy API

All scheduling strategies that can be plugged into the *Adaptive Job Scheduling System* must inherit from the *SchedulingStrategy* abstract class and define its methods.

```
class SchedulingStrategy{
 public:
    virtual int is_available(Job *j, Job *waitq, Job *runq) = 0;
    virtual void allocate_processors(Job *waitq, Job *runq) = 0;

    char *free_proc_vector;
    int num_free_proc;
    int nproc;
};
```

This class defines two methods which are called by the threads of the job manager. The *Execute Thread* calls the *allocate_processors* function when ever an event occurs in the system. If a client request is a query that asks the *Adaptive Job Scheduling System* if it can accept the job in the query, then the query is executed by the *Accept Thread* by calling the *is_available* function. To both these functions, the list of running jobs (runq) and the list of queued jobs (waitq) is passed.

In allocate_processors the strategy sets the status of all jobs in waitq, to ACCEPTED if the job has been accepted by the strategy, and to REJECTED if it has been rejected. If an accepted job is scheduled its bit map would also have to be set.

3.2.2 Performance Driven Strategies

The purpose of *Performance Driven Strategies* is to maximize a performance metric like *system utilization*, *job throughput*, *mean job response time* etc. These strategies also assume that jobs do not have deadlines and can be executed at any time after their arrival.

We now present a simple scheduling strategy which maximizes the system utilization. The strategy schedules jobs in a First Fit fashion and among the scheduled jobs processors are allocated uniformly.

Each incoming job specifies the minimum and maximum number of processors it can use. When a new job arrives, the scheduler recalculates the number of processors allocated to each running job. All jobs, including the new one, are allocated their minimum number of processors. Leftover processors are shared equally, subject to each job's maximum processor usage. If it is not possible to allocate the new job its minimum number of processors, it is enqueued. When a running job finishes, the scheduler applies the above algorithm to each job in the queue again.

After running this algorithm some jobs might shrink, some might expand, and some remain unchanged. The results are communicated to the running jobs by sending each a bit-vector (processor map) of the processors available to it. The jobs will then resize themselves.

The above strategy maximizes the utilization of the system but the response time of the jobs may suffer. Consider the following scenario in which Jobs J1, J2 and J3 arrive close to each other. The system has 128 processors and the minpe of each Job is 32 processors. Moreover all the jobs run for 100 seconds on 128 processors and have a linear speedup. If the above algorithm is used then the jobs will be allocated 43, 43 and 42 processors respectively. The utilization of the system will be 100 percent till the first job finishes and the mean response times would be 297.6, 297.6 and 304 seconds respectively. Now if the jobs were scheduled one after another the mean response time would be $(100 + 200 + 300)/3 = 200$ seconds and the system would become empty after 300 seconds. In both cases the job throughput is similar, about 3 jobs in 300 seconds, but the mean response time is better in the second case.

Now the latter system assumes that the jobs have linear speedups, similar execution times and clustered arrivals. But jobs typically have sub-linear speedups (hence the efficiency of the jobs is higher at fewer processors), varied execution times and uniform arrivals. Hence running the jobs together would have a better system utilization.

Though the scheduling strategy we have implemented is simple it provides improved performance over traditional queuing systems as will be shown in Chapter 4. The scheduler infrastructure allows us to plug in different scheduling strategies, and more sophisticated strategies are being explored.

3.2.3 Profit Driven Strategies

The scheduling decisions for the *Profit Driven Strategies* are based on maximizing a profit metric, which basically determines the amount the user would pay if the system finished executing his job. Each job also has a deadline and all accepted jobs have to be finished before their deadline. Here the scheduling strategy tries to maximize the profit of the system and finish all the accepted jobs by their deadlines. So the scheduling strategy must accept job only if the it is profitable to execute it and the system has resources to finish it by its deadline.

We have studied two such strategies. The first is a simple strategy which looks at the resources available in the system at job arrival time, and if the resources are available and the job is profitable the scheduling strategy accepts the job. The obvious disadvantage of this strategy is that, even though the resources are not available for the job at its arrival time, they may be available in future before its deadline. The second scheduling strategy maintains a look ahead and if the job can be scheduled in future at the look ahead the strategy accepts it.

3.2.4 Simple Profit Driven Strategy

The scheduling decisions here are made with the current resources only. When a new job arrives the scheduling strategy computes its *minpe* (minimum number of processors needed to meet deadline). If the minpe can be allocated, the scheduling strategy checks if scheduling the job is profitable and if it is the strategy accepts the job otherwise the job is rejected. The formal algorithm is presented below.

- *runq* is the list of running jobs.
- *new_job* is the new job in the system.

```
boolean AcceptJob(runq, new_job)
begin
    /* Compute the maximum processors available in the system */
    max_alloc = nproc;
    for all jobs j in runq
         max_alloc -= j.min_proc; //min_proc is the minpe of the job.
    if(max_alloc < new_job.min_proc)
        new_job.status = REJECTED;
        return false;
    /* Check if job is profitable */
    init_minheap(h);
    heapRecord hr;
    for all jobs j in runq
        if(j.allocated_proc > j.min_proc)
            hr.key = compute_loss(j, 1);
            hr.job = j;
            h.insert(hr);
    profit = 0;
    while((new_job.allocated_proc < new_job.min_proc)&&(!h.empty())
        hr = h.delete_min();
        job = hr.job;
        profit += compute_profit(new_job, 1);
        profit -= hr.key;
        proc = job.virtual_delete();
        new_job.virtual_add(proc);
        if(job.allocated_proc > job.min_proc)
            hr.key = compute_loss(job, 1);
            hr.job = job;
            h.insert(hr);
    if(profit < 0)
        new_job.status = REJECTED;
        return false;
    commit(runq, new_job); //Commits the virtual accepts and deletes.
    new_job.status = ACCEPTED;
    /* Allocate more processors if it is profitable */
    while((new_job.allocated_proc < new_job.max_proc)&&(!h.empty())
        hr = h.delete_min();
        job = hr.job;
        profit = compute_profit(new_job, 1);
        profit -= hr.key;
        if(profit > 0)
            proc = job.delete();
            new_job.add(proc);
        else break;
        if(job.allocated_proc > job.min_proc)
            hr.key = compute_loss(job, 1);
            hr.job = job;
            h.insert(hr);
    return true;
end
```

The algorithm takes *O(p + plog(p))* time to execute, where p is the number of processors. This is because computing the maximum number of processors available in the system takes *O(p)* time as there are at most p running jobs in the system and a maximum of p delete_min's on the heap would take *O(plog(p))* time. The call to *commit* commits the virtual adds and deletes and takes *O(p)* time. The functions compute_profit and compute_loss calculate the profit and loss if a processor is allocated or deleted from the job. This takes constant time and is determined by the profit function of the job.

Before calling this algorithm the system sets the minpes of all the jobs according to their deadlines and their memory requirements. Moreover if there are any idle processors in the system they are also allocated to the running jobs. This is done in a similar way except that max-heaps will have to be maintained.

3.2.5 Lazy Schedule Strategy

The *Lazy Schedule Strategy* delays executing a job if it is not profitable to execute it. It defines a *lookahead* and if it is possible to schedule the job at the look ahead then it postpones the execution of the job to the look ahead. When a new job arrives it checks if it is profitable to schedule the job at arrival time. If the job is not profitable execute at arrival time, the strategy checks if it is possible to schedule the job at the *lookahead* time and if it is the strategy accepts the job, otherwise it rejects the job.

As the strategy is invoked whenever a new job arrives or an old job finishes, the lookahead must guarantee that one of these events occurs before the lookahead time. The lookahead should also be small to accept more jobs, as large lookaheads may lead to many jobs missing their deadlines. *Simple Strategy* infact has an infinite look ahead. We choose the lookahead to be the minimum deadline among all the accepted jobs in the system. Hence at the look ahead the minpe processors of the job whose deadline is the lookahead, are freed. This may be enough to schedule a current job that cannot be scheduled now. Moreover the lookahead cannot be changed till the lookahead job finishes. The algorithm is formally presented below.

- *runq* is the queue of running jobs.
- *acptq* is the queue of accepted and not running jobs (delayed till look ahead).
- *new_job* is the new job that arrives into the system.

```
boolean LazyAcceptJob(runq, acptq, new_job)
begin
    lookahead = getLookAhead(runq, acptq);
    for all jobs j in acptq
        if(!AcceptJob(runq, j))
            if(!isSchedulable(runq, j, acptq, lookahead))
                forceAllocate(runq, j);   //Accpeted Job has to execute.
            //else leave the job in acptq.
    if(!AcceptJob(runq, j)){
        if(!isSchedulable(runq, new_job, acptq, lookahead))
            new_job.status = REJECTED;
            return false;
        else new_job.status = ACCPETED;
    }
    else new_job.status = ACCPETED;
    return true;
end

Job getLookAhead(runq, acptq)
begin
    static Job currentLookAhead;
    if(currentLookahead.status == FINISHED)
        for all jobs j in runq and acptq
            lookahead = job with least deadline;
        currentLookahead = lookahead;
    return currentLookahead;
end

void forceAllocate(runq, j)
/* Same as AcceptJob except that profit is not checked for and the
   processor is removed from the minimum loss job and added to j */

void isSchedulable(runq, j, acptq, lookahead)
/* Checks at the future lookahead time if the job can be scheduled. It
uses the future minpe for all accepted and not running jobs and
current minpe for the running jobs. */
```

The complexity of the algorithm is *O(nplog(p))* where p is the number of processors and n is the number of jobs in the *acptq*. Note that n is not bounded by p.

3.3 Database

The database serves three purposes, (i) it makes the system reliable as the state information of the scheduler is saved in the database and if the scheduler goes down, after coming up the state of the system is made consistent to before it crashed. (ii) the logged job-related information can later be mined for interesting information like the nature of jobs submitted, the average number of processors requested etc. (iii) all job related data is logged and any security breaks can be easily tracked.

Chapter 4

Simulation and Experimental Results

In this chapter we describe simulation and parallel cluster experimental results for the *Adaptive Job Scheduler*. First we compare the performance of the *Adaptive Job Scheduler*, with the utilization driven *Uniform First Fit Strategy*, with existing systems that do not support adaptive jobs. Then we compare the performance of the two profit driven strategies described in Chapter 3, with both adaptive and traditional jobs[1].

4.1 Scheduler with the Utilization Driven Strategy

We claim that our *Adaptive Job Scheduling System* with the *Uniform First Fit Strategy* has better *System Utilization* and improved *Mean Response Time* over current systems which do not support adaptive jobs. To demonstrate this, we simulated our Job Scheduling System through an event driven simulation and also performed experiments on a parallel Linux cluster. These experiments were performed separately for both adaptive and traditional jobs. The benchmark program used for the experiments was a 3D simulation that computes the molecular dynamics interactions of about 50,000 atoms in a cube. The program takes approximately 64.5 seconds to complete 100 iterations on 64 processors. It uses a naive parallel algorithm, and has a sub-linear speedup characteristic, as shown in Table 4.1. We used it as a realistic example application. However, we also simulated the performance of our scheme for jobs assumed to speedup perfectly.

[1]Traditional Jobs run on a fixed number of processors

Processors	1	2	4	8	16	32	64
Speedup	1.0	1.8	3.4	6.3	11.2	18.1	26.3

Table 4.1: Benchmark Program Speedup

4.1.1 Simulations

The simulations modeled an Adaptive Job Scheduler with 64 processors. Jobs arrive with an exponentially distributed arrival time. These simulations modeled the benchmark program mentioned above. To model the variance of job execution times, the number of iterations which the job executed was distributed exponentially with a mean of 100 iterations. The simulations were used to compute the mean response time and the mean utilization of the parallel system.

The results of the simulations, which were performed for 10,000 jobs are shown in Table 4.2, 4.3 below. Here λ is the mean arrival rate, and $1/\lambda$ is the mean arrival time and lf is the load factor (i.e. $lf = \lambda * (execution\ time\ on\ 64\ processors)$). The tables show the *Mean Response Time* and the *System Utilization* for different arrival rates, for both traditional and adaptive jobs. The simulations closely model our Adaptive Job Scheduling System and the Traditional Job Scheduling systems like PBS [15] and DQS [4]. Table 4.2 shows the simulation results for the Traditional and Adaptive jobs with linear speed up [2].

Table 4.3 shows the performance of the two systems for a benchmark program with sub-linear speedup described in Section 4.1. The sub-linear speedup makes the response time reasonable even for load factors slightly greater than 1 as shown in the table. [3]

$1/(\lambda)$ (s)	Adaptive Jobs		Traditional Jobs		lf
	MRT	Utilization	MRT	Utilization	
500	69.93	12.84%	129.37	12.52%	0.129
200	82.55	32.05%	162.43	31.29%	0.322
100	114.64	63.91%	280.46	62.57%	0.645
64.5	291.88	98.46%	22042.39	91.09%	1.0
60	14811.46	99.97%	39920.00	91.51%	1.075
45	86825.74	99.99%	107023.48	91.86%	1.43

Table 4.2: Mean Response Time (MRT) and System Utilization for jobs with a Linear Speedup

[2] In the adaptive case the *minpe* (minimum number of processors requested by each job) is a random number uniformly distributed from 16-64, and the *maxpe* (maximum number of processors requested) is set to 64. For the Traditional Jobs using a Traditional Scheduler the number of processors allocated is a uniformly distributed from 16 to 64.

[3] Here for the Adaptive Jobs the minpe is uniformly distributed from 1 to 64 and the maxpe is set to 64. For the traditional jobs the the number of processors is uniform between 1 and 64.

1/(λ) (s)	Adaptive Jobs		Traditional Jobs		lf
	MRT	Utilization	MRT	Utilization	
500	67.87	12.78%	165.26	9.19%	0.129
200	76.30	31.45%	185.86	22.98%	0.322
100	96.39	60.42%	233.07	45.94%	0.645
64.5	142.91	87.75%	395.99	70.99%	1.0
60	164.04	92.46%	487.76	76.27%	1.075
45	8677.23	99.98%	13985.08	95.33%	1.43

Table 4.3: Mean Response Time (MRT) and System Utilization for jobs with sub-linear speedup

4.1.2 Experiments on the Linux Cluster

We also performed experiments on a Linux Parallel Cluster with 32 nodes and with each node having two 1 GHz Pentium III processors. A random job generator was used to fire jobs to the Job Scheduler with the same *Mean Arrival Time* and *Execution Times* as in the simulation. The seeds of the random number generator were also synchronized with those used for the simulation. The same benchmark program was also used. The experiments were performed on the parallel Linux cluster for about 50 jobs for each arrival rate, as executing 10,000 jobs is prohibitive. Table 4.4 presents the Mean Response Time and the Mean System Utilizations for both Adaptive and Traditional jobs. The traditional jobs are submitted to our scheduler that emulates traditional schedulers like PBS [15] and DQS [4]. Adaptive jobs are submitted to the Adaptive Job scheduler described in Section 3.2.

1/(λ) (s)	Adaptive Jobs		Traditional Jobs		lf
	MRT	Utilization	MRT	Utilization	
500	89.5	17.4%	109.0	9.3%	0.129
200	70.0	29.3%	107.6	22.6%	0.322
100	76.4	68.4%	115.5	48.7%	0.645
60	211.8	99.2%	302.9	74.0%	1.0
45	295.2	99.6%	N/A	N/A	1.075

Table 4.4: Results for experiments on the Linux cluster with benchmark application

The results show a similar pattern of improvement with adaptive job scheduling, as do the simulations. The results differ because the response time and utilization of the scheduler does not converge for 50 jobs. The simulations reflect the long term steady state behavior of the system. On running the simulations for 50 jobs the results obtained matched closely with experiments on the cluster.

4.2 Profit Driven Strategies

Tables 4.5, 4.6 show the performance of the Simple and the Lazy scheduling strategies. The jobs submitted to the scheduler have a linear speedup and the minpe of the jobs is varied from 1 to 64, using a uniform random number generator. The job inter-arrival time is exponential and the maximum profit of each job is also varied exponentially with a mean of $1000. The profit function for the above simulation stays at the maximum profit till the time $arrivalTime + (deadline/2)$ and then linearly decreases to zero at the deadline. This would give an incentive to the strategies to finish the jobs earlier. The mean deadline for the above experiments is the same as the mean computation time on one processor. The bench-mark program chosen is the same as the one described in Section 4.1. All the simulations were performed for 10,000 jobs and hence the maximum possible profit achievable is $10 Million. Tables 4.6, 4.5 show that adaptive jobs increase the profit of the Parallel System by about 8-15%.

$1/(\lambda)$ (s)	Adaptive Jobs			Traditional Jobs		
	MRT(s)	Utilization(%)	Profit($)	MRT(s)	Utilization(%)	Profit($)
500	57.1	10.8	9205698	181.5	9.5	8412825
200	56.7	24.9	8531220	170.7	20.8	7342599
100	55.7	43.2	7455378	157.2	34.0	6019540
60	56.2	61.1	6295054	144.2	45.5	4873167
55.8	56.0	63.9	6134390	142.2	47.0	4726711
42	56.4	74.4	5362995	134.6	54.2	4039729

Table 4.5: Simulation Results for the Simple Profit Driven Strategy

$1/(\lambda)$ (s)	Adaptive Jobs			Traditional Jobs		
	MRT(s)	Utilization(%)	Profit($)	MRT(s)	Utilization(%)	Profit($)
500	60.4	11.1	9414019	180.4	9.8	8574479
200	62.7	26.1	8880660	168.4	21.3	7506604
100	65.3	46.4	7944163	146.5	34.4	6139762
60	66.4	65.4	6668211	130.2	45.3	4957229
55.8	66.6	68.2	6462683	128.2	46.9	4800582
42	66.1	78.4	5615287	114.8	52.7	4115911

Table 4.6: Simulation Results for the Lazy Schedule Strategy

Chapter 5

Conclusion and Future Work

We described the motivation, design and implementation of *adaptive* jobs and an *Adaptive Job Scheduling System* for parallel machines. The scheduler builds upon a measurement based load balancer, and can be used by applications written in a variety of languages including MPI and Charm++. The original load balancers, which aimed at resolving application induced load imbalances, were extended to shrink, expand or to change the sets of processors allocated to a job. The load-balancing strategies accomplish this by migrating user level entities (such as MPI threads and Charm++ objects) across processors. Mechanisms for controlling the behavior of the load balancer via bit vectors of available processors were implemented and validated. We also described a few simple job scheduling strategies, and some preliminary performance data.

The system described is a part of a wider *faucets* project, which aims at supporting the metaphor of computing power as a utility. Our future work will include expanded notions of *quality of service* contracts, and correspondingly sophisticated scheduling strategies that attempt to optimize more complex utility metrics than just processor utilization. Also more complex profit driven strategies which employ multiple lookaheads, and further improve the profit of the system are being explored.

The current system has been tested on clusters of workstations. We plan to port and evaluate the system on dedicated parallel machines, such as IBM SP, which allow socket based communication with external processes. Integrating the job scheduler with an automatic check pointing system, rather than relying on application-specific check pointing, is another direction for future work. We are also hopeful that our scheduler will be in production use on the 400 processor cluster at Illinois, for running ASCI Rocket Center jobs.

The faucets framework will include sophisticated *quality of service* contracts, competitive bidding processes, web based submission, monitoring and interaction with parallel jobs, along with services such as authentication, and file upload and downloads. We plan to utilize the common components in the Globus framework, and interoperate with Globus, so that the job scheduler runs as a Globus server.

A demonstration of the faucets project is available at:

http://charm.cs.uiuc.edu/Rsearch/faucets/faucet.html

References

[1] Milind Bhandarkar, L.V.Kalé, Eric de Sturler, and Jay Hoeflinger. Object-based adaptive load balancing for MPI programs. Technical Report 00-02, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, Sept. 2000. Submitted for publication.

[2] John A. Board, L. V. Kalé, Klaus Schulten, Robert Skeel, and Tamar Schlick. Modeling biomolecules: Larger scales, longer durations. *IEEE Computational Science and Engineering*, 1(4), 1994.

[3] Robert K. Brunner and Laxmikant V. Kalé. Adapting to load on workstation clusters. In *The Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 106–112. IEEE Computer Society Press, February 1999.

[4] D. W. Duke, T. P. Green, and J. L. Pasko. Research toward a heterogenous networked computing cluster: The distributed queueing system version 3.0. Technical report, Florida State University, May 1994.

[5] G. Edjlali, G. Agrawal, A. Sussman, and J.Saltz. Data parallel programming in an adaptive environment. In *Proceedings of the 9th International Parallel Processing Symposium*, 1995.

[6] S.Tuecke I. Foster, C. Kesselman. The anatomy of the grid: Enabling scalable virtual organizations. In *to be published in Intl. J. Supercomputer Applications*, 2001.

[7] L. V. Kale, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan, and Joshua Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, April 1996.

[8] O. Lawlor and L. V. Kalé. Supporting dynamic parallel object arrays. In *Proceedings of International Symposium on Computing in Object-oriented Parallel Environments*, Stanford, CA, Jun.

[9] M. Litzkow and M. Solomon. Supporting checkpointing and process migration outside the unix kernel. In *Usenix Winter Conference*, 1992.

[10] J. E. Moreira and V.K.Naik. Dynamic resource management on distributed systems using reconfigurable applications. *IBM Journal of Research and Development*, 41(3):303, 1997.

[11] J. Nagib, C. Árebe, A. Beguelin, and Bruce Lowekamp. Dome: Parallel programming in a distributed computing environment. In *Proceedings of the International Parallel Processing Symposium*, 1996.

[12] Ncsa, silicon graphics origin2000. http://archive.ncsa.uiuc.edu/SCD/Hardware/Origin2000/.

[13] Pittsburg supercomputing center, cray t3e. http://www.psc.edu/machines/cray/t3e/t3e.html.

[14] Pittsburg supercomputing center, the terascale computing system. http://www.psc.edu/machines/tcs/.

[15] Portable batch system. http://http://pbs.mrj.com/.

[16] Turing cluster. http://turing.cs.uiuc.edu/.

[17] V.K.Naik, Sanjeev K. Setia, and Mark S. Squillante. Processor allocation in multiprogrammed distributed-memory parallel computer systems. *Journal of Parallel and Distributed Computing*, 1997.