DYNAMIC LOAD BALANCING JOB SCHEDULER FOR CLUSTER SYSTEMS

A PROJECT REPORT

Submitted by

SIMON D (31507104096)

VIGNESH A (31507104111)

YOGESH J (31507104119)

in partial fulfillment for the award of the degree

of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING

SRI SIVASUBRAMANIYA NADAR COLLEGE OF ENGINEERING

SSN NAGAR, KALAVAKAM-603110

ANNA UNIVERSITY :: CHENNAI 600 025

APRIL 2011

ACKNOWLEDGEMENTS

We wish to express our sincere gratitude to **Dr. Aravindan C** for guiding us through our project. We thank him for the invaluable guidance and the time he has devoted in helping us with the project.

We extend our heartfelt thanks to Mr. Karunagaran for helping us by installing all the necessary tools and setting up an environment in High Performance Computing laboratory that led us to implement and complete our project without any difficulties.

We also express our gratitude to Ms. Jaishakthi for helping us by providing vital inputs about classification and their tools which helped us finish our project in time.

We also take immense pleasure in expressing out our sincere gratitude to our friends and family members who have helped us during the course of the project.

ABSTRACT

Critical scientific and engineering applications such as weather modeling, climate modeling, protein folding, aircraft simulation, classifiers etc. are computation intensive and take a long time to finish execution. In order to reduce their time for execution, these applications are parallelized and executed in a parallel environment such as distributed clusters, GPUs etc. In order to extract the best performance while executing in a parallel environment it is necessary that the resources in the parallel system are utilized effectively. Also, these applications, characterized by long turn-around time, should be executed in a fault tolerant manner as occurrence of faults (such as node failure in a cluster) would result in wastage of computational time. In this report, we present a scheduler capable of scheduling jobs and dynamically rescheduling them in a cluster in a load-balanced manner in order to use the resources of a cluster effectively and uniformly. Also, the scheduler is capable of detecting failure of a node in a cluster and automatically reschedules the jobs being executed in the failed node to other nodes so that the entire application need not be executed from the beginning. To show the capabilities of the scheduler designed, we have also developed a parallel application which increases the accuracy of a classifier tool, libsvm 3.0, by performing a grid search over the parameters of the libsvm.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

BLCR	Berkeley Lab’s Linux Checkpoint/Restart
SVM	Support Vector Machines
MPI	Message Passing Interface

CHAPTER 1
INTRODUCTION

The computational demands of applications have been increasing over time. The applications are found in areas of climate research, biology, astrophysics, high energy physics etc. The computational demands of these applications cannot be met even with the increase in processor performance. The usage of clusters of computers is one of the most promising means by which we can bridge the gap between the computational demands and the available resources. The applications are broken down into parallel sub-problems and these sub-problems are run on the individual nodes of the clusters where each node in a cluster is an independent computing system with its own operating system and possibly with dedicated peripheral devices.

To effectively utilize the resources of the clusters, multiple applications can be run on these clusters. As a result, there could be contentions among the applications to access the resources. Efficient scheduling mechanisms are required to resolve the contentions among the applications for resources. The need for efficient **schedulers** that makes best use of resources available and ensures faster execution of applications arises. So, a highly efficient scheduler must allocate tasks to the cluster based on the arrival time, amount of execution time and the resources they utilize while performing the computation. This feature of scheduler which incorporates task scheduling fairly is called as **load balancing..**

Load balancing can be broadly classified into two major types from which the hierarchies of methodologies and load balancing strategies follow. **Static load balancing** is a technique which involves the allocation of resources statically once the job is submitted to the scheduler and is never meant to be modified during the runtime of the application, whereas **dynamic load balancing** is more of a reactive type of load balancing which allocates tasks as they are submitted to the cluster and are dynamically reallocated periodically so that each node in the cluster is effectively used for computational purposes by the application it also dynamically schedules for the resources which the application requires.

1.1 MOTIVATION

The major factors that motivated us to take up this project are listed as follows.

- CPU cycles wasted by idle machine.
- Amount of power consumed by a node gets wasted in case of failure.
- Unutilized resources.
- Redundancy in processing due to node failure.

1.1.1 NEED FOR PARALLELIZATION

Sequential processing and serial applications modeling were the trend of computing in the past years. With the advent of high speed processors and faster computer that can perform millions of complex instructions within seconds made our application programs run much faster than what they would have taken a decade ago. But as the complexity of applications increases even the fastest single processor on earth will not be able to match its requirements. Let’s consider for an example a weather modeling **expert systems** that predict the weather and climatic changes that will take place in the future, predict Tsunamis, earthquakes and volcanic eruptions. In order to compute these measures large amount of input data in the form of geographic information’s, images and graphs are fed to the computers. These applications may take months or even years to complete their execution.

With the rise of parallel computing environment such as clusters, grids, and supercomputers message passing parallel programs are designed that run on high performance distributed memory parallel computers that would take too long to run on conventional sequential machines. The weather modeling expert system we discussed could be finished within a few hours or days. Thus, the need for parallelizing applications has arisen to reduce the execution time of complex applications drastically.

1.1.2 NEED FOR LOAD BALANCING

Once we have parallelized the application and it is submitted to the parallel computing environment such as a cluster, the task assigned to each node must be evenly balanced. Any imbalance in the distribution of load leads to decreased throughput, wastage of computational CPU cycles in many nodes of the cluster, additional power consumption by a heavily loaded machine, increased fault rate and so on. An efficient scheduling mechanism and a load balancing strategy must be followed in order to allocate resources properly and utilize the parallel environment in a meticulous manner.

1.1.3 NEED FOR DYNAMIC LOAD BALANCING

The bottleneck of a static scheduler plays a major role in cluster as it schedules the task to each node by breaking the application into chunks and this is called fixed scheduling. If a node has completed its task of executing the submitted chunk of application, the node becomes idle as no further tasks are submitted by the scheduler. On the other hand, dynamic scheduling can avoid this by assigning a job to the idle node immediately thus improving the throughput and decreasing the wastage of resources. This explains the need of a efficient dynamic scheduler with load balancing capabilities

1.1.4 NEED FOR FAULT TOLERANCE

Let us assume a parallel computing environment set up as a cluster of interconnected machines/nodes. A parallelized critical scientific application such as N-body simulation, Grid Search, Ant Colony Optimization or a Data with huge inputs of data so that the application may run for a week or a month or so is submitted to the cluster. After few days of executing the application successfully, a **node failure** occurs in the cluster environment consequently all the power and computational CPU cycles used by that node becomes a waste of computational energy in the end and that particular process running in that node has to be restarted again in some other node from the beginning adding to the cost of additional power and resources being used.

1.2 PROBLEM STATEMENT

The aim of this project is to develop a dynamically load balancing scheduler. The users should be able to submit a task/job from any node of a cluster which will be scheduled to least loaded node. The scheduler should also monitor the loads of the individual nodes of the cluster and dynamically reschedule the jobs in them in case of load imbalance. The scheduler should also detect any node failure and redistribute the jobs from the failed node to other active nodes.

1.3 ORGANISATION OF THE THESIS

Chapter 1 - A compact summary of the report is given in this section. This section serves to state the objectives and present a brief background that explains the impetus behind the work.

Chapter 2 – ‘Literature Review’, this describes in general about the existing work on dynamic load balancing schedulers and checkpointing.

Chapter 3 – “Proposed Methodology” describes the methodology we propose for dynamically load balancing a cluster and the implementation of the described methodology.

Chapter 4 – ‘Evaluation and Results’, this provides a clear picture of the results obtained.

Chapter 5 – ‘Conclusion and Future Work’, provides a complete picture of the success of the enhanced system over the existing system and future advancements that could be made.

CHAPTER 2
LITERATURE REVIEW

This chapter describes the details of the existing work on job schedulers and load balancing that inspired us to take up this project and the difference between our proposed model and the existing ones.

2.1 CLUSTER SCHEDULER

Most of the applications executed on a parallel environment using clusters are critical applications that require high computational capabilities from the system's end and rely heavily on resources that are responsible for the computations involved. In order to efficiently allocate the resources and schedule the tasks on the nodes of a cluster for high throughput, we need a scheduler. Some of the basic schedulers and the methodologies they follow are discussed below.

2.1.1 Maui/Maob

The *Moab Workload Manager* [1] is based on the Maui batch scheduler [8], with all its flexibility and added features – backfilling, service factors, resource constraints and weights, fair-share options, direct user/group/account prioritization, target wait times, etc. However, based on the Maui Scheduler Administrator's Guide [4] its default behavior out of the box is a simple FCFS batch scheduler, with a backfilling policy that maintains a time reservation for the first job in the queue –EASY backfilling. In the Maui scheduler, the priority of each job is a weighted sum of several factors, where the weights are set by the administrator. Each factor itself is a weighted sum of sub-factors, whose weights, again, are governed by the administrator. After looking at the source code we found that even though all the factors' weights are set to 1 (in an array called *CWeight*), all the weights of the subfactors are set to 0, except for that of the job's queue time which is set to 1 (all the sub-factors weights are saved in the *SWeight* array). The result is that the job's queue time is the only factor that is not zero, and even though the factor weights are set to 1, the queue time is the priority function— resulting in a FCFS scheduler.

2.1.2 IBM LoadLeveler

IBM's LoadLeveler [9] supports several schedulers, such as FCFS, FCFS with backfilling, gang scheduling, and can also interface with external schedulers. The system also supports checkpointing [ ] and restarting of running jobs, and specific *IBM SP* hardware. Within its own set of schedulers many of the features are tunable: first and foremost, an administrator can rewrite the priority function *SYSPRIO* and use current system data. Examples for such data are a user's class, how many jobs the user/group has in the system, etc. Other parameters can also be used to establish a fair-share priority function. The administrator can also set specific privileged user/group/class accounts. This, coupled with support for job preemption, allows for high priority jobs to preempt low priority ones. At the user level, each user can change the running order (or explicitly specify one) of his own jobs. LoadLeveler supports backfilling, and can even be tuned to use either the best-fit or first-fit metrics to choose jobs for backfilling. The default scheduling of LoadLeveler is FCFS: the default priority function is FCFS, as the *SYSPRIO* function is simply the job's queue time. Backfilling is *not* set by default, but when enabled, its policy is first-fit, with time reservation set only for the first job in the queue When using backfilling, users are obligated to specify a runtime estimate for their jobs. When a job exceeds its time estimate, it is killed (sent a SIGKILL signal). This is similar to the EASY backfilling policy

2.1.3 Portable Batch Scheduler(PBS)

The Portable Batch System (PBS) comes in two flavors: OpenPBS [16] is intended for small clusters, and *PBS-Pro* [3] is the full fledged, industrial strength version. For simplicity, we will focus on PBS-Pro. The suite includes a very versatile scheduler support. Schedulers included with the suite are FCFS, SJF, user/group priorities and fair-share. Also, site specific schedulers can be implemented natively in the C and TCL programming languages, or in a special language called BaSL. Other features include checkpoint support, re-pack and rerun support for failed or stopped jobs, and failed nodes recovery. An administrator can distribute shares among groups, whose shares can, in turn, be divided to subgroups. This creates a tree structure in which each node is given shares, which are distributed by administrator assigned ratios to its child nodes, all the way down to the tree leaves. The leaves themselves can be either groups or specific users. As with other software suites, the administrator can define work queues with various features. Queues can have certain resource limits that are enforced on the jobs they hold. A job can even be queued according to its specified resource requirements, the administrator can define a queue for short jobs, and the queuing mechanism can automatically direct a job with small CPU requirements to the short jobs queue. Of course the administrator can define a priority for each queue, thus setting the dispatch order between queues, or can be selected for dispatch in a round robin fashion. Queues can also be set inactive for certain times, which allow using desktops as part of the cluster at night or holidays. The *PBS-Pro* system support preemption between different priority jobs. An administrator can define a preemption order between queues, by which jobs from higher priority queues can preempt jobs from lower priority queues if not enough resources are available. Inter-queue preemption is enabled by default, but there is only one default queue. Being the exception that makes the rule, the default scheduler in both PBS systems is SJF. To prevent starvation, the system can declare a job as starving after some time it has been queued (with the default time set to 24 hours). A starving job has a special status no job will begin to run until it does. The result begin is that declaring a job as starving causes the system to enter a draining mode, in which it lets running jobs finish until enough resources are available to run the starving job. The starvation prevention mechanism can be enabled only for specific queues. Backfilling is supported, but only in context of scheduling jobs around a starving job waiting to run, and only if users specify a wall time CPU limit. Like the starvation prevention mechanism, backfilling can also be enabled for specific queues. As mentioned before, the default scheduler is SJF, and both the starvation prevention mechanism and backfilling enabled for all queues.

2.1.4 Sun Grid Engine (SGE)

The Sun Grid Engine (SGE) [15, 13, 14] is much simpler then its contenders. SGE has two scheduling policies: FCFS, and an optional administrator set function of Equal-Share scheduler. The latter is a simple fair-share scheduler that tries to distribute resources equally among all users and groups. For example, to overcome a case where a user submits many jobs over a short period of time, its latter jobs will be queued until other users had a chance to run their jobs. An administrator can also define new job queues, with specific dispatch order among the queues themselves. Currently, the system does not support backfilling. The default behavior is still FCFS, since the default priority function is again the job’s queue time.

2.1.5 Condor

Condor is a specialized workload management system for compute-intensive jobs. Like other full-featured batch systems, Condor provides a job queuing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their jobs to Condor, and

Condor places them into a queue, chooses when and where to run them based upon a policy, monitors their progress, and ultimately informs the user upon completion.

While providing functionality similar to that of a more traditional batch queuing system, Condor’s novel architecture allows it to succeed in areas where traditional scheduling systems fail. Condor can be used to manage a cluster of dedicated nodes. In addition, several unique mechanisms enable Condor to effectively harness wasted CPU power from otherwise idle desktop workstations. Condor can be used to seemlessly combine all of your organization’s computational power into one resource.

The Condor features are extensive. They provide great flexibility ofr both the user submitting jobs and the administrator who provides the necessary resources and the CPU time. Some of its noteworthy features are Distributed submission, Job priorities, User priorities, Job dependency, Support for multiple job models, support for checkpoint and migration, periodic checkpoint, Job suspend and resume, and authorization and authentication.

2.2 LOAD BALANCING

The process of scheduling the tasks submitted to a cluster so that it maximizes the utilization of resources which eventually increases the efficiency of the cluster and increases the throughput is termed as load balancing. The two types of load balancing techniques include static load balancing and dynamic load balancing.

2.2.1 STATIC LOAD BALANCING

Assume that in a parallel application, a loop with *N* independent iterates is to be executed by *P* processors. The iterates are stored in a central ready work queue from which idle processors obtain chunks. The sizes of the chunks are determined according to a scheduling method which attempts to minimize the overall loop execution time. The method is classified as *nonadaptive*, when the chunk sizes are predictable from information that is available or assumed before loop runtime, or *adaptive*, when the chunk sizes depend on information available only during loop execution.

2.2.1.1 Non-adaptive task scheduling

Non-adaptive task scheduling methods generate equal size chunks or predictable decreasing size chunks. Equal size chunks are generated by *static scheduling* (STATIC), where all the chunks are of size *N/P*, *self-scheduling*, where all the chunks are unit size, and *fixed size chunking* [28] . STATIC has very low overhead, but it provides good load balancing only if the iterate times are constant and the processors are homogeneous and equally loaded. SS is suitable only when communication overhead is negligible as it requires sending *N* control messages.

2.2.2 DYNAMIC LOAD BALANCING

Dynamic load balancing strategies are applied when the tasks arrival time is not known a priori and therefore the system needs to schedule tasks as they arrive. The aim of dynamic scheduling may be different from minimizing the schedule length as in [17] where the aim is to obtain the short completion time of a batch of tasks, but to minimize some global performance measure related to the quality of service (QoS). In [6] it is defined a value *V* for each completed task and the aim is to maximize the sum of the values of completed tasks in a given interval of time. The value *V* is based on task priority, task deadline and a task performance metric.

The algorithms proposed for scheduling DAGs in heterogeneous systems consider only one DAG (or job) to schedule and therefore they are all static approaches [2, 18]. The dynamic scheduling on the other hand considers independent jobs (or tasks) and assigns one job to one processor, since it is not considered that a job may have several tasks [6, 17]. The dynamic scheduling methods consist in a scheduling strategy and a scheduling algorithm [6], [17]. The scheduling strategy defines the instants when the scheduling algorithm is called to produce a schedule based on the machine information and tasks waiting in the queue at the time they are called. The dynamic behavior is achieved by calling the static scheduling algorithm along time. As described in [12], the dynamic scheduling strategies can be divided in two categories: immediate mode and batch mode. In immediate mode only the new task arrived is considered by the scheduler; in batch mode the scheduler considers the new task and the tasks schedule before but that are waiting to be processed. It was shown that the batch mode allows to achieve higher performance.

2.3 CHECKPOINT/RESTART

Clusters of commodity computers running Linux are becoming an increasingly popular platform for high performance computing, as they provide the best price/performance ratio in the marketplace. But while the size and raw power of Linux clusters continues to increase, many aspects of their software environments continue to lag behind those provided by proprietary supercomputing systems. One feature missing from Linux clusters is a robust, kernel-level checkpoint/restart implementation that can support a wide variety of parallel scientific codes. The ability to checkpoint and restore applications (i.e. save the entire state of a job to disk, then later restore it) provides many useful features in a cluster environment. Some of them are,

2.3.1 Gang scheduling

Being able to checkpoint and restart a set of parallel processes that are part of a single application (gang scheduling) allows for both more flexible scheduling, and higher total system utilization. System administrators can allow different types of jobs to run at different times of day (favoring large, long jobs at night, for instance, and short interactive ones by day), simply by checkpointing any jobs that are still running when the scheduling policy changes, then restarting them when the policy changes back. A subset of nodes or even the entire cluster may be brought down for system maintenance without interfering with user job completion. Without checkpoint/restart, these types of scheduling actions must be implemented either by simply killing user applications (thus wasting the resources they have consumed so far), or by ‘queue draining’, in which no new jobs are accepted after a certain time, and the system waits for users jobs to finish (during which, by definition, the system runs at less than full capacity). Checkpoint/restart also allows extremely large jobs that may consume the entire system for a long duration (these are often the motivating applications for building a large cluster) to be intermittently scheduled, avoiding locking out all other users for what would be unacceptably long periods of time. Finally, checkpoint/restart can provide higher total system utilization by allowing the running of whatever configuration of available processes will most fully saturate the number of CPUs (or other resources) provided by the cluster.

2.3.2 Process Migration

A well-designed checkpoint/restart can allow checkpointed processes to be restarted not only on their original node, but also on other nodes in the system (or possibly even nodes on a different system that are part of a computing grid or other distributed system). Such migration can allow a job to continue if the imminent failure of one of its nodes is detected (such as a failing CPU fan or local disk). Certain partial failures can also be worked around. For instance, most Linux clusters that use custom high-performance networks also provide a standard Ethernet interface for administrative traffic. If a node’s high-performance network interface becomes unusable, it may still be possible to migrate its processes onto a fully performing node via the Ethernet interface, allowing a job that would otherwise be terminated to continue. Process migration has also proved extremely valuable for systems whose network topology constrains the placement of processes in order to achieve optimal performance. The Cray T3E's interconnect, for instance, uses a three-dimensional torus that requires processes that are part of the same parallel application to be placed in contiguous locations on the torus. This naturally results in fragmentation as jobs of different sizes enter and exit the system. With process migration, jobs can be packed together to eliminate fragmentation, resulting in significantly higher utilization: the appearance of checkpoint/restart on the T3E has been credited as one of the major factors that allowed NERSC's T3E installation to go from 55% to 95% utilization [WONG99]. While networks with such constraining topologies have recently been out of style,

2.3.3 Periodic Backup

Finally, checkpoint/restart can provide an increased level of reliability in the face of node crashes and/or nondeterministic application failures (such as infrequently occurring synchronization bugs). If intermittent checkpoints are taken, a job can be restarted from the most recent checkpoint, and should continue successfully, assuming its actions are idempotent. Some special handling for files may be used by a checkpoint system to 'rollback' changes to an application's open files, making many otherwise non-idempotent applications idempotent. This is particularly true for scientific applications, which tend to write their output files in a serial, log-like fashion: a simple truncation of files to their length at checkpoint time suffices for rollback in such cases.

2.4 Berkeley Lab's Linux Checkpoint/Restart (BLCR)

BLCR can be used either as a standalone system for checkpointing applications on a single machine, or as a component by a scheduling system or parallel communication library for checkpointing and restoring parallel jobs running on multiple machines. BLCR has established a collaboration with the LAM MPI developers, who have modified LAM to work with BLCR's user level hooks. As a result, it is already possible to checkpoint and restart simple parallel scientific codes written in MPI (such as the NAS Parallel Benchmarks

[NPB02]) with BLCR. While more work needs to be done to get BLCR ready for use in a production environment, this work is in progress, and the modular design of BLCR should make it the logical choice for a wide range of cluster management and communication projects that are underway for Linux clusters.

2.4.1 Checkpointing using BLCR tool

In order to use the BLCR tool to checkpoint and restart a application. We must make the executable checkpointable. BLCR provides certain commands to take care of this, while running a application we must use the command *cr_run* to make the execute the application. This command makes the running application checkpointable, then we can use the *cr_checkpoint PID (process ID)* command to checkpoint the application. BLCR provides a variety of arguments that can be passed along with the cr_checkpoint command. *Save-all* is used to to save the state of the executable along with the checkpoint file which is named context.PID by the BLCR. The context file can be saved in any directory as specified by the *DIR* command line argument. By default, it is stored in the current directory.

2.4.2 Restarting using BLCR tool

To restart a checkpointed application. The context file should meet some conditions. The PID with which the context file got saved should not be occupied by some other process and the executable should be available with its contents unchanged. But, we can avoid these bottlenecks by using command line arguments namely *save-all* and *–no-restore-pid.* You may restart a program on a different machine than the one it was checkpointed on if all of these conditions are met (they often are on cluster systems, especially if you are using a shared network file system), and the kernels are the same. The application can be restarted by using the *cr_restart contxt.PID_filename* command. Once the application is started it resumes the process from the point where it was checkpointed.

2.5 GRID SEARCH

Grid Search is a critical computational intensive scientific application often employed in determining desirable points in an n-dimensional space filled with values in the form of Grid. This is similar to searching a person in a lost space assuming the space as a grid and the lost person as some point in the grid. The method adopted to determine the desirable points is computed by varying different set of points taken in a continuous, discrete or random from the n-dimensional space of the grid. A Grid search is usually started with some initial point taken from the grid space and the values are iterated till we obtain some desirable points in the grid space.

CHAPTER 3
PROPOSED METHODOLOGY

In this chapter, we present the method adopted by our scheduler in order to schedule a job submitted to the nodes of a cluster, dynamically balance the load of the cluster and achieve fault tolerance without wasting the computing time. Also, implementation of the proposed method is described in detail.

3.1. Solution Design

The solution to the proposed problem is discussed below:

3.1.1. Scheduling job to a node

In order to utilize the resources of a cluster effectively and uniformly, we assign jobs to nodes based on the computational load of the nodes. For this purpose, a resource monitor (client) program is run on all the nodes of a cluster. This client gets information of the load of the node on which it runs. This information is sent to the job scheduler periodically. The scheduler thus knows the information regarding the load of all the nodes of the cluster. Whenever a new job is started, the job is sent to the job scheduler which assigns the job to the least loaded node based on the information it receives from the client (resource monitor) processes.

3.1.2 Dynamic load balancing

The job scheduler has the load information of all the nodes of the cluster. If the job scheduler (server) detects that the load across the cluster varies widely, it finds the heavily loaded nodes of the cluster based on the history of loads it has and issues a checkpoint command to the heavily loaded nodes. Upon receiving a checkpoint command from the job scheduler, the resource monitor selects some processes running on it and checkpoints them, i.e., saves the state of these process in a file. The job scheduler issues restart commands to the less loaded nodes which takes these checkpoint files as input and resumes the stopped processes from the point it was stopped. In this way jobs are moved from the heavily loaded nodes to the lightly loaded nodes, thereby, dynamically balancing the load across the cluster.

3.1.3 Node failure detection and fault tolerance

The resource monitor program runs on all the nodes of the cluster. The resource monitor frequently communicates with the job scheduler to update the load information of the nodes and failure to update the same for a long time would mean that the node has failed. In order to achieve fault tolerance, all the processes running in a node are periodically checkpointed. When the job scheduler detects a node failure, it sends the checkpoint files of the processes that ran on it to the other active nodes of the cluster and resume them from their most recent checkpoints.

3.2. IMPLEMENTATION OF SOLUTION DESIGN

The implementation of the proposed solution is discussed in detail below:

3.2.1 Scheduling job to a node

A resource manager program is run on every node of the cluster and a job scheduler/resource monitor program is run on one of the nodes in the cluster which decides on which node a submitted job as to be scheduled. The client program which is run on all the nodes of the cluster finds the load of the node on which it is run using the ***uptime system call***. The ***uptime*** system call gives the average load of the system for the past 1 minute, 5 minutes and 15 minutes. The system call also gives information about the time for which the node has been used and the number of users using the node. The information pertaining to the average load of the system is sent to the job scheduler periodically. For this purpose of sending and receiving the load information of a node, a TCP socket is created and all communications between the job scheduler and the resource monitors take place through this socket. A separate port is allocated for the transmission of load information.

The server receives the load information for all the nodes of the cluster periodically. Using the history of load averages (over the past 1,5,15 minutes) it calculates the effective load of the nodes giving more weightage to the most recent activity history. Using this information the server calculates the average load of the cluster.

When a new job is to be started, the executable for the job is passed as an argument to the start process. The start process opens a TCP socket and sends the executable to the job scheduler. The job scheduler which has the load information of all the nodes of the cluster finds the node with the least load and sends a start signal and the executable to that node. The resource monitor upon receiving this start signal begins the execution of the job submitted on the node which hosts the resource monitor. This way, a newly submitted job is scheduled to the least loaded node.

3.2.2 Dynamic load balancing

The job scheduler calculates the average load of the cluster. A node whose load is greater than this average load of the cluster by an amount greater than a threshold, then that node is said to be overloaded. The value of the threshold can be changed by the user in accordance with the nature of the applications being run on the cluster.

A checkpoint command is issued to the client of the overloaded node by the server. On receiving a checkpoint command, the client picks up a process from the list of processes running on that node using the **ps** command. The selected process is checkpointed it in a shared folder and is killed. The

Berkeley Linux Checkpoint Restart (BLCR) tool is used for checkpointing purposes. The checkpointing is carried out using the command **cr_checkpoint –save-all PID** where the PID is selected by using the **ps** command. The **–save-all** option is used so that the process can be restarted in a new node without the exectable residing in that node. Here, the process to be checkpointed must be linked with the BLCR library while compilation and run using the **cr_run command** of the BLCR tool. The checkpoint files of the processes are stored in a shared folder common to all the nodes of the cluster. This is done to ensure that the node where the process has to be restarted have access to the checkpointed process's file.

The job scheduler then sends a restart signal along with the path of the checkpoint file of the killed process to a node whose load is lesser than that of the average load of the cluster. The client process (resource monitor) upon receiving a restart signal moves the checkpoint file from the path it receives to its own local folder. It then restarts the process from the point where it was stopped using the **cr_restart –no-restore-pid context.PID command.** The –no-restore-pid option is used to ensure that the restarted process is submitted as a new process so that there is no clash in process ID with an already existing process. In this way, the processes are rescheduled from over-loaded nodes of a cluster to under-utilized nodes automataically.

3.3.3. Detection of Node failure and fault tolerance

As already stated, the client processes periodically sends information about the load of a node to the server process. This communication is used to detect failure of node in a cluster. A count of how many times the client process has communicated with the server is tracked by the server process. Since all clients communicate with the server periodically and the time period of communication is exactly the same for all clients, the number of times a client communicates with the server is exactly the same for all the clients. If a node has failed due to some reason, then the client process running on it cannot communicate with the server. If the number of times a client has communicated with the server is significantly lesser (here 2) than that of the other client processes, then it can be inferred that the node that runs that client has failed.

In normal course of action, if a node fails, then the entire set of process that was running on the node as well as the parent process's tree of parallel processes has to be restarted from the beginning which results in wastage of computational time. The scheduler developed by us periodically checkpoints all the processes running on the system in a shared folder so that in case a node fails or fault occurs, then the processes can be restarted from the most recent checkpoint. This process has also been automated in the scheduler/resource manager.

As already stated, failure of a node can be detected using the communication history between the server and the client. Also, all the processes running on a node is checkpointed periodically. If the server has to restart the process in the failed node it must be informed about the processes running in the failed node. For this purpose, along with the average load of a node, information regarding the number of process running in the node and the process IDs/path of the checkpoint file of all the process running on the node is sent to the server. Now, the server knows the path/location of all the checkpoint files created by the node's client. If a node failure is detected, then the checkpointed files of all the processes that were being executed is taken and distributed to the less-loaded nodes of the cluster where these processes can be resumed.

3.3.4 Parallelization of grid search

In order to evaluate the performance of the scheduler developed by us, we have chosen a scientific application for parallelization. We parallelize the grid search algorithm to optimize the kernel parameters and hyper-parameters of an SVM (Support Vector Machine) to improve the training accuracy. We use the openMPI flavor of Message Passing Interface in order to parallelize the grid search. Each of the parallel processes evaluates the training accuracy for a different point in the solution space. The value of the parameters that gives the best accuracy is chosen as the optimal solution.

CHAPTER 4
EVALUATION AND RESULTS

The dynamically load balancing scheduler was implemented according to the design presented earlier and was tested using the application developed, namely, parallelized grid search along with several sequential programs. The following results were obtained.

4.1 Performance Measure

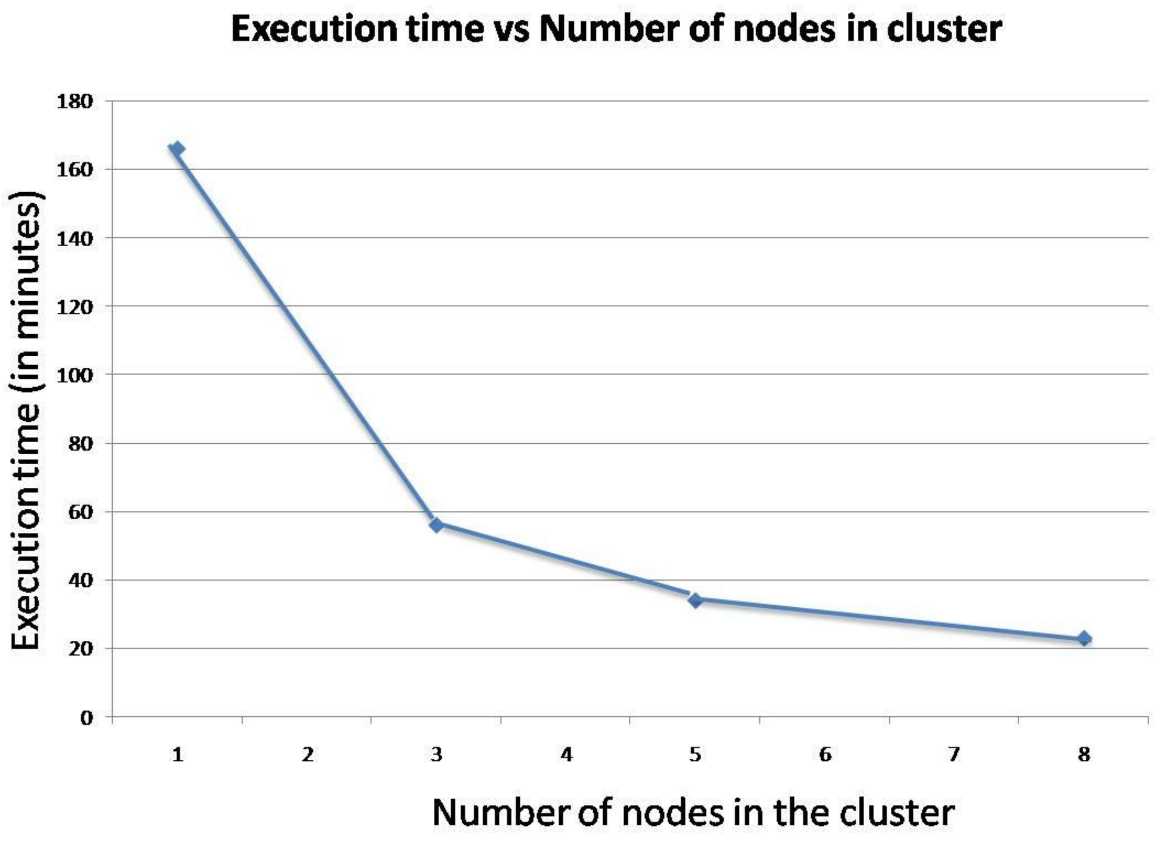

Fig 4.1 Comparison of execution time with increase in number of nodes

As shown in figure 1, the execution time of the application came down with increase in parallelism. It can be observed that the execution time decreases sub-linearly with the increase in number of nodes (thereby number of parallel processes) which is in accordance with the Amdahl's law.

In order to test the scheduler, we submitted a job to an overloaded node using the start process and it was observed that the process executed on some other less loaded node. Also, load imbalance across the cluster was detected by the scheduler and dynamic rescheduling of the processes took place.

We ran the application we developed on 5 nodes of the cluster. We simulated node failure by shutting a node down which was automatically detected by the scheduler and the processes running on it was rescheduled to other nodes of the cluster thereby enabling the application to finish its execution. The execution time for this case was nearly the same as that of the case of no node failure when we shut down the node towards the completion stage of the application.

This is significantly better than other approaches followed in the case of node failure to achieve fault tolerance such as running the applications multiple times in parallel in several nodes or restarting the application from the beginning.

CHAPTER 5
CONCLUSIONS AND FUTURE DIRECTIONS

This chapter concludes the report and presents the future directions to enhance the functionalities of the designed scheduler.

5.1 CONCLUSION

We have implemented at present a scheduler capable of scheduling jobs and dynamically rescheduling them in a cluster in a load-balanced manner in order to use the resources of a cluster effectively and uniformly. Also, the scheduler is capable of detecting failure of a node in a cluster and automatically reschedules the jobs being executed in the failed node to other nodes so that the entire application need not be executed from the beginning. In order to evaluate the efficiency of the scheduler designed by us and demonstrate its capabilities, we have also developed a parallel application which increases the accuracy of a classifier tool, libsvm 3.0, by performing a grid search over the hyperparameters of the libsvm. All the functionalities of the scheduler has been tested with the scientific application we have developed.

5.1 FUTURE DIRECTIONS

The BLCR tool for checkpointing and restarting applications works only if the applications do not use sockets. This limits the parallel programs for which the scheduler developed can be used as the MPI_SEND and MPI_RECV commands of the openMPI use TCP sockets for communication. Also, the BLCR tool does not work well with any user interactive programs thereby limiting the applications for which the scheduler can be used. Works in future should focus on improving the BLCR tool to include socket programs and user interactive programs.

Our project developed assumes that the cluster uses a shared file system and we save the periodically checkpointed files in a shared folder from which the processes are restarted. However, in clusters not using the shared file system, the storage of checkpoint files becomes a critical issue. The number of copies of the checkpoint files that has to be saved in the nodes across the cluster has to be optimally fixed up. Saving the checkpoint files in multiple nodes is carried out so that in the case of a node failure, these files can still be read from other nodes.

APPENDIX 1

USING THE BLCR TOOL

The following are the commonly BLCR commands.

cr_run <executable>	Used in order to make the executable checkpointable
cr_checkpoint *PID*	Checkpoints the process with the process ID PID in a file context.PID
cr_checkpoint –save-all PID	Checkpoints the process along with information about the executable
cr_restart *context.PID*	Resumes the process from the state saved in the context file with the old process ID
cr_restart –no-restore-pid *context.PID*	Resumes the process from the state saved in the context file with a new process ID

APPENDIX 2

SOURCE CODE

The CD that has been submitted along with this report consists of folders that comprise the following:

1. Document stating the abstract of the project.
2. The project report.
3. The research paper on which this project is based on.
4. Source code and instructions to execute the project.

REFERENCES

[1] Barbosa. J, Belmiro Moreira (2009). ‘Dynamic job scheduling on heterogeneous clusters’. Eighth International Symposium on Parallel and Distributed Computing.

[2] Barbosa. J, C. Morais, R. Nobrega, and A.P. Monteiro (2005). ‘Static scheduling of dependent parallel tasks on heterogeneous clusters’. In *Heteropar’05,* pages 1–8. IEEE Computer Society.

[3] Duell J, Hargrove P, Roman E (2002). ‘Requirements for Linux Checkpoint/Restart’. Published on *Lawrence Berkeley National Laboratory Technical Report LBNL-49659.*

[4] Jackson. D, Snell. Q, and Clement. M. (2001) ‘Core algorithms of the Maui scheduler’. In *Job Scheduling Strategies for Parallel Processing,* Lecture Notes in Computer Science. Springer-Verlag.

[5] Jong-Kook Kim *et al* (2007). ‘Dynamically mapping tasks with priorities and multiple deadlines in a heterogeneous environment’. *Journal of Parallel and Distributed Computing,* 67:154–169.

[6] M. Maheswaran, S. Ali, H.J. Siegel, D. Hensgen, and R.F. Freund (1999). ‘Dynamically mapping of a class of independent tasks onto heterogeneous computing systems’. *Journal of Parallel and Distributed Computing,* 59:107–131.

[7] Paul H. Hargrove and Jason C. Duell (2006). ‘Berkeley Lab Checkpoint/Restart (BLCR) for Linux Cluster’, published in10th International Software Metrics Symposium (METRICS'2004), Proceedings of SciDAC.

[8] Ricolindo L. Cariño, · Ioana Banicescu (2008). ‘Dynamic load balancing with adaptive factoring methods in scientific applications’ , The Journal of Supercomputing, Springer.

[9] Topcuoglu. H, Hariri. S, M.Y Wu (2002). ‘Performance-effective and low-complexity task scheduling for heterogeneous computing’. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274.

[10] Wei Sun, Yuanyuan Zhang, and Yasushi Inoguchi (2007) . ‘Dynamic task flow scheduling for heterogeneous distributed computing: Algorithm and strategy’. *IEICE Trans. INF. & SYST.*, E90-D(4):736–744.

[11] Yoav Etsion, Dan Tsafrir (2005). ‘A short survey of commercial cluster batch schedulers’. *Technical Report 2005-13, School of Computer Science and Engineering, the Hebrew University*.

[12] Kannan. S, Roberts. M, Mayes. P, Brelsford. D, Skovira. ‘*Workload Management with LoadLeveler’*. IBM, first edition, Nov 2001.
ibm.com/redbooks.

[13] Thomas Sterling, ‘Beowulf cluster computing with linux’. The MIT press, 2002.

[14] BLCR User Guide. http://upcbugs.lbl.gov/blcr,2010.

[15] Cluster Resources, Inc. *Maui Scheduler Administrator’s Guide*. http://www.adaptivecomputing.com/resources/docs/maui/mauiadmin.php

[16] MOAB workload manager. http://www.supercluster.org/moab/.

[17] Sun Microsystems, Inc. *Sun ONE Grid Engine Enterprise Edition Administration and User’s Guide*, 2002. version 5.3. http://download.oracle.com/docs/cd/E19080-01/grid.eng53ee/index.html

[18] Sun Microsystems, Inc. *N1 Grid Engine 6 Administration Guide*, 2004.
http://download.oracle.com/docs/cd/E19080-01/n1.grid.eng6/817-5677/index.html

[19] Sun Microsystems, Inc. Sun grid engine. http://gridengine.sunsource.net/, 2004.