Efficient Batch Scheduling Procedures for Shared HPC Resources

Fountanas Angelos

35354s05-38g

August 22, 2008

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2008
Abstract

Efficient batch scheduling is a need that occurs the moment that we have to run a number of parallel tasks on an architecture employing a respectable number of processors. The problem becomes more specific when we consider that we usually do not know how long is a task going to execute for, or if and when a task execution is going to break for any reason, thus forbidding a complete scheduling plan of a specific load from the beginning. Utilising the most of resources available while at the same time being fair, to users making use of the resources, has been the objective for developers since the early beginnings of large-scale architectures. The objective is to construct a simulator implementing basic scheduling policies and process historical real data, computing periods coming from the HPCx supercomputing service’s resources manager, Loadleveller.
## Contents

Contents ................................................................................................................................. 3  
List of Figures ......................................................................................................................... 5  
Chapter 1: Introduction .......................................................................................................... 7  
Chapter 2: Project Proposal ................................................................................................... 9  
Chapter 3: Background ......................................................................................................... 10  
  3.1. Space sharing ........................................................................................................... 10  
  3.2. Time sharing ........................................................................................................... 11  
  3.3. Backfilling .............................................................................................................. 11  
    3.3.1. Aggressive backfilling .................................................................................. 12  
    3.3.2. Conservative backfilling ............................................................................. 13  
  3.4. Checkpointing strategy ......................................................................................... 14  
  3.5. Migrating jobs strategy ......................................................................................... 14  
  3.6. Loadleveler - HPCx’s scheduling strategies ........................................................... 16  
Chapter 4: Design ............................................................................................................... 17  
  4.1. Code Architecture ............................................................................................... 17  
  4.2. Linked List ............................................................................................................ 18  
  4.3. System ................................................................................................................ 21  
  4.4. Scheduler ............................................................................................................ 21  
  4.5. Events ............................................................................................................... 22  
  4.6. Job .................................................................................................................... 23  
  4.7. Queue ................................................................................................................. 25  
  4.8. Main ................................................................................................................... 26  
Chapter 5: Implementation .................................................................................................. 28  
  5.1. Node ................................................................................................................... 30  
  5.2. Linked List ........................................................................................................... 30  
  5.3. System ............................................................................................................... 31  
  5.4. Scheduler ............................................................................................................ 31
List of Figures

Figure 1 – Space sharing policy..........................................................................................11
Figure 2 – ‘C’ can execute without delaying ‘B’ in aggressive backfilling ....................12
Figure 3 – ‘C’ would delay ‘B’ in aggressive backfilling.............................................13
Figure 4 – Conservative backfilling demonstration......................................................14
Figure 5 – Job migration ..................................................................................................15
Figure 6 – Overall system control flow diagram..........................................................18
Figure 7 – Doubly linked list .........................................................................................19
Figure 8 – add() function..............................................................................................20
Figure 9 – Remove() function ......................................................................................20
Figure 10 - Scheduler control flow diagram .................................................................22
Figure 11 – Job_Start/End events creation....................................................................25
Figure 12 – Main control flow diagram.........................................................................27
Figure 13 – Use case diagram.......................................................................................29
Figure 14 - Instance of scheduling activity.................................................................40
Acknowledgements

First and foremost, I would like to thank Dr Kenton D’ Mellow and Dr Stephen Booth for their time and effort supervising this project, as well as for their concise advising which proved very significant. I would also like to thank my close friends in Greece for the distance shouts of encouragement whenever needed. Last but not least, I thank my family for their invaluable empowering effort, as well as for making this year reality for me.
Chapter 1: Introduction

This dissertation addresses the matter of efficient task scheduling on modern shared resources architectures. The adopted approach was to design a batch scheduling simulator following an Object-Oriented fashion and implementation took place in C++.

The objective was to construct basic scheduling policies and simulate historical real data, computing periods coming from the HPCx supercomputing service’s resources manager, Loadleveller. Results should be able to mimic the real scheduling outcomes of HPCx leading to insight and a discussion about how to combine already existing policies, or applying new ideas on the sets already existing and working, could result in better queue time measurements for the sake of more advanced and efficient scheduling strategy.

In this work, it was necessary to understand how modern scheduling works, what techniques exist and/or are already being used for batch scheduling in shared resources. Additionally, solid comprehension on the problem of batch scheduling from both, users and machine administration, points of view and use, is required.

The rest of this report has the following structure:

Chapter 2 presents the issue of efficient batch scheduling on shared HPC resources. I discuss the nature of the problem and the approach adopted towards tackling it. It also gives some low-detail information on what the concept of the application is.

Chapter 3 will review the background work already taken place in this field, and presents key techniques developed in HPC. Here, we also discuss the main trends adopted by modern architectures, and consider the pros and cons of each one. Additionally I will briefly describe Loadleveller’s setup on HPCx.

Chapter 4 presents the design of the application developed. I describe the concrete parts of the system and will expose the concept under which all parts are going to co-operate for it to work as a whole. Additionally, the architecture of the code is outlined in detail.

Chapter 5 describes the parts of the system separately in full detail. Implementation details are carefully outlined in this chapter. Additionally, I present any weaknesses
created during implementation, and any compromises made at each point of the implementation.

Chapter 6 is the Data description section. In this, I describe the data that HPCx uses for scheduling, as well as the part of this data inserted as input for the simulation to take place. I also discuss the way some data is being used, its purpose in this application, and how it could be used in the simulator’s future upgraded forms.

Chapter 7 presents the overall throughput of the system implemented. I discuss the nature of the input and I make some scheduling performance discussion. I also discuss limitations, either known from the beginning or met during the development, existing in the overall system.

Chapter 8 is the conclusions section. I make a detailed discussion about the potential future upgrades, so that the application can reach certain goals not achieved due to limited time. Additionally there is a programming language discussion, in which there’s a brief application-specific presentation of the pros and cons of the two most important and widely used Object-Oriented programming languages, C++ and Java.
Chapter 2: Project Proposal

This project involves research in efficient batch scheduling on shared HPC resources. This translates to a study on how modern workload managers schedule jobs on large-scale architectures. The architecture model used, in this paper, for developing insights into batch scheduling and building knowledge on techniques used in modern supercomputers, is the HPCx national Supercomputing Service, and its resource manager, Loadlever.

The actual problem to be tackled responds to the inability of scheduling algorithms to make use of the most processors possible for the longest time possible. This takes place by efficiently mapping jobs as closer as possible to the whole number of processors for the longest time period, whilst at the same time trying to demonstrate fairness to individual users that constitute the ownership of the workload.

The approach taken for addressing this problem was to build a batch scheduling simulator making use of an Object-Oriented approach. The development model followed for this application is the evolutionary prototyping model. The actual objective was to build a simulator that initially implements basic functionality by employing the most primitive as well as standard technique of space sharing. In later steps it is to employ some more complicated and sophisticated scheduling methods such as aggressive backfilling or region migration, as long as environment improving techniques such as job check pointing, to be explained in background chapter.

One of the primary targets set for this development was that no matter when the development process of this simulator should stop, its design should make it able to exhibit ability to employ several updates; As updates we could consider a number of scheduling policies that will analytically be described in section 8.2, and that could enable this application to return efficient queuing throughput for combinations of modern scheduling techniques.

The final objective was that, again, no matter how many iterations were taken through our development model, it would produce some results that would exhibit its working scheduling ability, no matter if it is less efficient than the existing set up of HPCx or not.
Chapter 3: Background

In this chapter I will discuss the existing policies for scheduling jobs on shared resources architectures. Policies already developed begin from the simplest resource and time sharing, to later, more efficient and complicated ones, for addressing the problem of effectively utilising resources. Most of them are commonly used in HPC systems in several combinations and configurations for achieving the best scheduling performance according to machine size, architecture and workload quality.

In this section I present some basic terms such as machine, or system, for the HPC architecture, job representing the application that a user submits for execution and the queue, representing the list in which jobs are stored after submission and on which scheduler acts for efficiently keeping the machine busy. Additionally, some basic terminology of Loadleveller, IBM’s resources manager, as well as the techniques it employs, is mentioned as appropriate.

3.1. Space sharing

In a space sharing strategy, each job submitted gets to the front of the queue. Upon reaching the front, it is executed to completion. This seems to be the fairest scheduling policy. Every job will be executed at its time, while nothing is going to delay the start or the finishing of the execution, as shown in figure 1. Its main problem though, is that the factor of utilising the machine as much as possible is not taken into consideration at all, making it highly cost-ineffective.

Space sharing was one of the first ways to schedule jobs on HPC resources, but the need to exploit resources more effectively, especially filling up the idle time of CPU’s occurring by the arbitrary requirements of jobs respectively going towards scheduling, soon created the need for smarter scheduling algorithms. Space sharing means that the head of the queue is going to start executing even if this is going to leave a number of CPU’s unexploited, due to the fact that it doesn’t serve anybody’s exact requirements at that moment.
3.2. Time sharing

In this policy, executions may experience possible ‘checkpoints’ whose frequency and duration may get defined by various parameters such as job length or resources required. These ‘checkpoints’ save at the point the execution has reached, and pauses execution. Therefore, some programs exhibit short wait times, thus completing later than under dedicated CPU’s like in a space-sharing policy. Their purpose is quite different than the checking taking place periodically for saving results, promising low-cost recoveries in case something goes wrong, due to status saving. Trying to avoid misconceptions fed by this, from now on I shall refer to them as breaks. This policy is one of the first ones trying to address this problem but is not much used nowadays, since the rise of more advanced techniques like backfilling.

3.3. Backfilling

Backfilling is a policy where a narrower job ‘C’ is allowed to jump over a waiting wide job ‘B’, so long as the execution of ‘C’ does not delay the projected start of ‘B’ [1]. This is shown in figure 2. Backfilling is a later concept in the field of batch scheduling and came to partially address the problem of arbitrary sets of requirements referred to in
section 3.1. Backfilling has been divided into two main strategies, conservative, and aggressive. A significant assumption made for this policy is that the job lengths are known a priori.

3.3.1. Aggressive backfilling

In aggressive only the head of the queue has what is referred to as reservation number. A job is only allowed to backfill when the projected start of the head of the queue is not going to be delayed. As we can see in figure 2, in a rack consisted of 10 nodes, Job ‘A’ is already running using 8 nodes. Job ‘B’ is the head of the queue and it requires 10 nodes to run on. There is a Job ‘C’ which is behind ‘B’ in the queue, and will require 2 nodes for its execution. The scheduler will check if it can place job ‘C’ to run on the two free nodes. The main issue is that job ‘C’ has to finish before job ‘A’ does, so that it doesn’t delay the starting of job ‘B’. This is why the concept of wall_clock_time_limit was given birth in LoadLeveler. Wall_clock_time_limit is the time, the user anticipated, that the programme will run for. Therefore, the scheduler has to check this variable to verify that ‘B’ will not be delayed by anyone. If the anticipated time of ‘C’ is going to exceed the finishing of ‘A’, ‘C’ will have to wait for ‘B’ to finish and then execute, as shown in figure 3. In the opposite case, job ‘C’ will take advantage of the two free nodes, during the execution of ‘A’.

Figure 2 – ‘C’ can execute without delaying ‘B’ in aggressive backfilling
3.3.2. Conservative backfilling

In *conservative backfilling* every *job* is given a reservation number when entering the *system* [2]. A small *job* is allowed to leap forward and execute, only when the manoeuvre will cause no *job* to be delayed. Based on the previous example and figure 4, ‘C’ could execute after ‘B’ only if ‘B1’, ‘B2’ and ‘B3’, with higher priorities than ‘C’, concurrently did not have probability of being delayed. The highest priority, while ‘B’ is running, is ‘B1’ and its execution could not be violated by ‘C’, even if it needed more execution time. But ‘B3’ would be delayed if the case was that, therefore ‘C’ would execute after ‘B3’.

*Figure 3 – ‘C’ would delay ‘B’ in aggressive backfilling.*
3.4. Check pointing strategy

*Check pointing* does not have to do with how *jobs* schedule on a *machine*, therefore, it cannot be called a scheduling policy. Since experience has illustrated that during the execution of an application, many things can go wrong in various levels, the concept of *check pointing* illustrates periodic status saving of a running application. A checkpoint file contains the program’s data segment, stack, heap, register contents, signal states, and the states of the open files at the time of the checkpoint[3]. This can construct a fairly large file, which will cost some time to concentrate and write to disk. This can either be user initiated, where the user makes a call to the *ckpt()* routine, or system initiated, where loadleveler automatically *checkpoints* every specified time interval. During the time this procedure takes place, the state of the application is characterized as *check pointing* (CK). In case of any kind of failure, the *job* can be restarted from the last successful checkpoint.

3.5. Migrating *jobs* strategy

The resources of a High Performance Computer may be separated in several different areas of execution. On HPCx the main areas are ‘capacity’, ‘capability’, and ‘development’. Each of these areas is ready to run *jobs* within some specified limits of
resources asked. Therefore, one area may run jobs, from 1 (serial) to 192, the second one from 192 to 1024, and the third one, from 1024 to the upper bound that the region can have, according to the machine. If an area’s queue is getting too crowded, the system should take an action in order to relax the queues, and keep users from waiting too much for their job to run. This action is migrating jobs, that practically means taking a job from a region and transferring it to another one, thus speeding up the time an application in a crowded region, would wait for executing.

Migration strategy is not a currently implemented strategy on HPCx. Migrating takes place manually, when needed, by the administration team. This is considered to be one of the most interesting ideas and I, ideally, wish to head my investigation towards this. In the example following (Figure 5), the capability region is getting overloaded, thus the jobs will delay their start time way much. The scheduler takes the last two jobs X and Y, and places them to the capacity region that seems to be more relaxed. In this way, X and Y are going to start executing much faster, exploiting resources from another region. In this case, the job queued in one region gets to the end of the queue, or maybe at a specific position according to the scheduling policy, of the region which is finally going to execute.

![Job migration. X and Y migrate from Capability to Capacity region.](image)

Figure 5 – Job migration. X and Y migrate from Capability to Capacity region.
3.6. Loadleveler - HPCx’s scheduling strategies

LoadLeveler is the software, developed by IBM, that manages the scheduling policies and enables an efficient resources use in cluster environments. HPCx employs the *conservative backfilling* policy, being described in chapter 3.3.1, as well as the *checkpointing* policy, described in 3.4. Loadleveler offers various flexible ways for setting the *machine* up to schedule, towards any personalised approach, thus making it dependent on the system administrators and the approach they have. Loadleveler also offers the possibility to attach an external scheduling algorithm, through EASY. EASY stands for External Argonne Scheduling sYstem, it’s developed by Argonne National Laboratory and allows overriding all the present scheduling settings by attaching an alternative implementation as the scheduling policy.
Chapter 4: Design

The approach of this research was on building a batch scheduling simulator whose design will be thoroughly discussed in this chapter. The main concept is based on having a central event list recording all actions taken in the system and a scheduler running through it every some fixed, predefined amount of time.

4.1. Code Architecture

The system employs an event-driven architecture. In this, an event is triggering the action of another, or sets of other events. The core mechanism is a doubly linked list that stores all events taking place and also contains methods for supplying information to any subsystem in need of it.

As it is depicted in the control flow diagram in figure 6, a file input stores new jobs into a linked list, and an event referring to each of these jobs is created in the events list. Events types are discussed in detail in section 4.5. Upon file input end, the scheduler is called and it is the first event, after the file input stage, to go into the events list before the main iterative loop starts running. The iteration lasts for steps of number equal to the events list length and checks the event type of each event.

The four different event type cases are going to be analytically presented in section 4.5. If the event encountered is a job_submit, the corresponding job object from the job list is added in the queue list. If the event is a job_start, the system is informed about the CPU number occupied and the job_status of the corresponding job object in the queue list is updated. In the case of a job_end event, the system is informed about the CPUs freed, and the corresponding job is erased from the queue list. If a sched_run event is encountered, the scheduler is called.

When the scheduler is called, it checks on system availability and starts going through the queue list for retrieving the appropriate queued jobs qualified to run. While the machine has free CPUs, the scheduler calls job_startevent for a start event to be created in the events list for every job respectively. The start event will implicitly call the job_endevent function which is responsible for placing a job_end event for the job in the events list. The time of the start event is set to the time that the scheduler scheduled this event to run and the end event’s time is equal to the start plus the actual CPUTime that represents the
time that this job ran for. When system availability reaches zero, or when the queue list reaches the end, the scheduler reschedules a sched_run event in the events list.

Finally, when events list contains no more events, the procedure ends and Main returns value 0, representing the normal exit of the program, as expected.

4.2. Linked List

The linked list is considered to be the most important mechanism in this application. The Node class, which has linkedlist class as a friend, implements doubly linked nodes. This
means that a node consists of three parts, the data part, the next and the previous part that point to the next and the previous node respectively as shown in figure 7. The list consists of nodes connected to each other and has head, which represents the top node of the list, and a tail, representing the final one. It is implemented as a template class, meaning that its data will contain a template, as is discussed in chapter 5. It contains a small number of functions capable of handling all the operations needed in the simulation. More specifically, it employs methods add and remove, for adding and removing nodes in the list. A graphical representation on their design can be seen in figures 8 and 9 for add and remove respectively. These two functions are designed to handle all potential add/remove activity, irrespective of the contents of the list, and make sure that the list is always sorted, since insertion is always performed in the appropriate position. Additionally, the list counts the objects contained in it by having a length attribute. This increments every time an add call returns successful and decrements when a remove call takes place.

Further on there is a search function that handles the pairing of an event to the corresponding job object. This is used in the Main iterative loop where a submission event corresponds to a job object in the job list. This action is more thoroughly discussed in section 4.8, and a further analysis is presented in chapter 5.

In figure 7 we can see the format of the linked list implemented for serving our needs. Each node contains three pointers, *p for pointing to the previous node, *n for pointing to the next node and *data that points to the actual data contained. It was decided from quite early in the design stage that the linked list should be doubly linked, since in the case of a single linked list we are obliged to run through the entire list from top towards the bottom counting objects, in order to find the object needed. This happens due to the fact that a node is not aware of its previous one. This is considered as inefficiency for an application designed to make intensive use of list, such as this.
In figure 8 we can see three different scenarios of the *add* function. Taking as example the *linkedlist* presented above, we consider the *nodes* having time values as shown below. We can then see three new *nodes* with three different values, by which this list is ordered. The one with value 3, after going through the function *job::before()* which compares times and is described in the section 4.6, is set to go on top since its ordering value is the smallest of all. Therefore, the *head node* will point to it, and its *next* will point to the previous *head node*. The *previous* part of the now second *node* will point to the new one inserted, and the addition is considered complete.

In the second example, consider the *node* with value 13 that has to move in between two *nodes*. The connection as existing until now breaks. Previous *node’s* (10) *next* points to the new *node* and *previous* of the *node* after (15) now points to the new *node*. Respectively, the *next* pointer of the new *node* now points to its new next one (15) and the *previous* points to its new previous (10). In the last scenario, the *node* inserted is the biggest of all and has to go to the end of the list, the *tail* now points to the new *node* and the previous *tail’s* *next* points to the new *node*. Finally, the *previous* part of the new *node* points to the previous *tail node*.
In this figure we see an example of removing a node from the list using the remove function. The, towards removing, node’s previous (10) has to set its next pointing where the, towards removing, node’s next part (20) is. Additionally the next node (20) of the node to be removed (15) has to set its previous pointing where the previous of the, towards removing, node pointed (10). Having taken these actions we haven’t yet actually removed the node object, but since it has practically lost all connection with the list, it is considered as removed. Before function exits, the object is explicitly deleted from memory for avoiding garbage concentration and keeping memory clean.

4.3. System

The system is a mechanism that holds a certain number of processors. It contains a function for initialising a system with a CPU number, as well as a system_update function, for keeping the system CPU number aware of charge change occurring from job job_start and job_end events. The system_update function checks on the type of the event from which it was called. If the event is of type job_start, CPU requirements will be subtracted from the system. In any other case, which can only be job_end, CPU requirements will be added to the system, as result of CPUs freed from a job finishing execution. The scheduler accesses system availability, meaning the available CPUs at the moment, so that it can decide on whether jobs can move on towards execution. Main calls update function for charging and discharging CPUs according to system availability and work load requirements as is shown later.

4.4. Scheduler

As we can see in the scheduler flow diagram, in figure 10, the scheduler is responsible for checking the system availability and addressing jobs to run accordingly. If it is the first time that the scheduler is called, it reschedules an event for itself in the list. If the time called is not the first, it checks on the system availability. If the system availability is zero, it again reschedules an event for itself. If the system can take jobs, it checks on whether the queue has jobs or not. If no jobs exist in the list it reschedules itself in the events list. If the queue list is not empty, it checks if their job_status is “Q” and reads the CPU requirements of each job from the top of it, where job objects are stored. Jobs with zero CPU requirements will not be considered for scheduling. This action keeps being taken for as long as the system has free CPU able to take new jobs running.

For every job granted to execute there is a call made to a job_start event, belonging to events class. The job_start event automatically creates a job_end event for this job, as shown in the events entity design.

If the number of CPUs available becomes less or equal than zero (0), the process breaks, and the scheduler calls the rescheduling function in order to reschedule itself in the events list so that it runs again in a predetermined period of time. I should note here that for the
first time in the execution, the scheduler is being explicitly called in Main, as it will be shown in the description of the Main part or the application. After it has been called once, it will keep rescheduling itself every a predetermined period of time, until the end of the simulation. Finally, if the rescheduling function finds that a sched_run event lies in the tail of the events list, it no more creates new sched_run events and the function returns.

![Scheduler control flow diagram.](image)

4.5. Events

*Events* is the entity that controls the actions taken on *events* happening in the *system*. It inherits properties of a *job* object, so that it can use the appropriate data stored in it and create an *event* about it. This is happening through the inheritance mechanism of C++. Inheriting a class means that the object inheriting, the *child*, is able to use all of the data and functionality of the inheriting one, the *parent*. The object inheriting and making use of all functionality of the *parent* class can also have new data, as well as redefinitions of attributes and methods for itself.
Events are defined to be one of the following four cases:

job submit: An event of type job submit refers to a job that has been submitted and has not yet been considered for scheduling.

job start: An event of this type means that the job in which it refers to starts running.

job end: job end is the type of event that refers to a job that has been running since its start event was encountered, and is now about to finish execution.

sched run: An event sched run calls the scheduler again so that it keeps on scheduling jobs to the system.

All actions taken for each of the events are further described in the Main section.

The events class contains basic functions for controlling, adding and removing events to the events list. This includes using functions from the linkedlist to determine where an event should be added and where one should be removed from, as well as actually adding and removing on and from the events list. This class also contains a printing function, used for printing an event line out to screen. This has been used only for debugging purposes.

4.6. Job

Job represents a task towards execution in the system. It is meant to contain all data concerning a job and comes from input. Its functionality extends up to the point of constructing an object as well as adding and removing it from the job list. Some information contained in a job object, such as the job_status or the job_start and job_end time, is updated throughout the simulation.

A job has a job_status that shows at which state the job is at that moment. job_status can have four different values as shown below.

“NS”: NS stands for Not-Submitted. This status exists when a job is read from input. If the job is successfully inserted in the job list, its status changes to “S”, which is described below, and an event corresponding to this job is created, as is described in section 4.8.

“S”: S stands for Submitted. This status exists from the moment that a job object goes successfully in the job list. It means that a job has not yet been considered for scheduling, and it just declares that a non-queued job exists in the job list.
“Q”: Q stands for Queued. This status occurs when a job has been processed by the Main function and has been added in the queue list, for the scheduler to consider it for scheduling.

“R”: R stands for Running. This status occurs when a job has already started executing, meaning that the scheduler has already created a job_startevent for this job and this event has already been processed. job_status “R” lasts until the job_endevent is encountered, time at which the job object is going to be removed from the queue.

Further details on the way each job_status case occurs and what the application’s actions are when each is encountered, is given in the section 4.8.

Job contains the job_start and job_end functions that are responsible for creating a start and end event for the job that is permitted to execute by the scheduler. A flow diagram for these two events and the way they work is shown in figure 11. The scheduler is the only unit, within the whole system, calling the job_start function, which in turn is responsible for calling the job_end function, creating an end event corresponding to the job having called the job_start. This way it is assured in the simplest way that every job that a start event was created for, also has a corresponding end event.
When a job_start event is called, the submit time of this event is set to the time the scheduler has at the moment of the call. Then, the actual job_start event is created and added to the list. Before the function ends, it calls the job_end function which sets the submit time of this event, thus the end time of the job, equal to the job_start event time plus the CPU Time of this job -the time that this job actually ran for according to the input data.

4.7. Queue

Queue is the class that is responsible for keeping jobs that have been submitted and are to be considered for execution. The queue is a linkedlist of job objects that get stored upon submission. Queue can alter a job object with all potential job statuses. The first time this list goes in use is when Main runs on a job_submit event, it changes the job object’s status into “Q” and it inserts the job object in the list. The scheduler uses the queue to retrieve which jobs can run at the time given and calls start_event as discussed in section 4.6. When a job_start event is encountered, the job_status of the corresponding job in the queue list is set to “R”, status described in section 4.6.
There is the possibility that more than one queue can participate in the system. This application employs a simple list for implementing the queue, but a piece of code is escorting this report and further details on how it could work can be found in section 5.7.

4.8. Main

The Main part of the application is the part which coordinates the co-operation of all the other mechanisms. A graphical representation for this can be found in the control flow diagram in figure 12. At this point, the application is responsible for reading in the file input, the structure of which will be described in the chapter 6, and break it in separate strings on a per line basis. At the same time it builds the corresponding job object as well as the event object and stores them in the job and events lists respectively for every line of input, setting the job_status to “NS” standing for non-submitted.

Main then explicitly calls the scheduler for the first time, time at which no further action than a self rescheduling will be taken. Exiting the scheduler will have placed a sched_run event in the events list. Then the Main function goes into iterating through the events list. The iteration will last for steps equal to the list length and list will be searched from the head towards the tail.

If a job_submit event is encountered, the job_status of the corresponding event in the job list will change from “S” to “Q”, and the job object will be taken in the queue list, being prepared for the scheduler to read it upon call. Additionally, it goes in the next position of the job list so that when the next job_submit event is encountered, the job list is already in the position of the corresponding job object.

When a job_start event is encountered, the search function is called to retrieve the corresponding job object from the queue list. Then, a call to system_update is made, to charge the system with the CPU requirements of the starting job. Additionally, this event will cause the change of job_status to “R”, representative of the running state of the job.

If a job_end event is encountered, the search function is called to retrieve the corresponding job object from the queue list. Then, Main will free the CPUs from the system, thus increasing its availability. Additionally it will remove the corresponding job object from the queue list. The last event case is sched_run. In this case, Main will call the scheduler which will read the queue list and set a job_start and job_end events for every job found liable to execute.
Figure 12 – Main control flow diagram.
Chapter 5: Implementation

The system implements five main entities as presented in chapter 4. That is the *machine*, the *scheduler*, the *job*, the *queue* and the *events*. In this section I will provide some implementation details on the individual classes developed. Additionally there will be some discussion, escorting each section, on inefficiencies developed throughout the development procedure, as well as ways that could have led implementation in clearer and more efficient outcome.

All classes consist of a header and a source file. The header contains all methods and attributes declarations for its corresponding source file. Source file contains all method implementations. All headers start and finish with the pre-processor commands `#ifndef/#define/#endif`, also known as inclusion guards that help to avoid code repetition in the compilation. If the `#ifndef` macro has not been defined, the code following will be compiled and the macro will be set so that the next time it is encountered, the code following won’t be compiled for second time. All source code uses the standard namespace for avoiding confusion occurring from variables existing and applying changes to different namespaces than expected.

In figure 13 we can see a Class diagram containing all classes implemented, as well as relationships occurring from a simple Use case.
Figure 13 – Use case diagram.
5.1. Node

The Linked list consists of nodes properly equipped with previous and next pointing parts for participating in a doubly linked list, implemented in node.h. Class node is a template class and has the template class linkedlist as a friend. This header implements the node default constructor as long as a constructor that initialises the next and previous parts to NULL and takes a template as argument that initialises the data part.

5.2. Linked List

Linked list class is also a template class. It is implemented in solely a header file as in the case of node and includes node.h which is the core part of the linked list. It contains a pointer to the head and the tail of the list and the default constructor sets these two to NULL. The functions implemented here, besides a constructor, are add, remove, search and printlist which are structured as following:

Linkedlist(): The default constructor of the linked list class. This reserves space for a linkedlist object to be stored, and it also sets head and tail to NULL.

add(): Add() initially checks if the list is empty, at which case it will set the head and the tail pointing to it. If the list contains at least one node and the towards insertion node is smaller than the head or greater than the tail, then insert node will become head or tail respectively, since the list is intended to be sorted from the smallest towards the biggest submission time. In all other cases, add() will successfully rearrange connections so that the new node is successfully added. Every time add() function returns, the length property of the list has increased by one.

remove(): This function receives a template data as argument, and starts searching through the list in order to find the node matching the data given. When the object is found, remove() will rearrange connections concerning nodes related to this action, so that it practically disconnects the towards removal node. Removal of the node is considered successful exactly due to the reason that the linked list is no more aware of its existence. Eventually the object is explicitly deleted from memory. Remove() can handle removing nodes from head or tail just as add() does.

search(): search() is implemented for retrieving and returning an object according to its ID. This is used from Main, and especially from job_start and job_end events that need to find the corresponding to the event job object in the list. This function is unlikely to fail in retrieving the object needed, since it is sure that it exists in the queue list.
5.3. System

System is the class that represents the system on which jobs are to run on. System implements two functions. That is the machine(CPU) which is called in Main and takes an integer as an argument which will set the CPU number of the system as well as system availability which is also set during initialisation. The second function implemented is sys_update that takes as an argument a CPU number and the name of the event having called it. If the name of the event is job_start, the CPUs passed will be subtracted from the system availability. In any other case, which can only be the job_end, the CPUs are added to the system availability. sys_update can only be called from job_start and job_end events, while the initialisation of the system takes place only once in the beginning of the Main part of the application.

5.4. Scheduler

Scheduler is implemented in two basic functions. One is responsible for applying the actual scheduling activity and the other one for repositioning an event that will invoke the scheduler, in the events list.

The way these two work is presented here:

sched_run(): This function takes as argument the queue and the events list, as well as the time at which it was called. If it is the first time that the scheduler runs, then it just calls sched_reposition in order for it to place a sched_run event in the events list. If it’s not the first time, the scheduler checks on the system availability and if the job encountered has status “Q” and can execute, a job_start event will be called. Additionally, a variable will increase by one, eventually showing the number of jobs that have been scheduled. When a job is scheduled, the next job will be considered for scheduling. If system availability is too low for this, the scheduler will call sched_reposition. At this point the scheduler could be going on checking on whether any of the other jobs can use the given system availability, making use of a backfilling policy.

If, while iterating through the queue list, the scheduler reaches the bottom of the queue, the sched_reposition() function will be called. At this point I should mention that any job that has zero CPU requirements will not be considered for scheduling.

Sched_Reposition(): This function takes as an argument the time for which its being called, as long as the events list. If it’s called for first time, the function will create a sched_run event with time equal to this of the head of the events list. In any other case, it will create a sched_run event whose time, will be set to the time of call, increased by a predefined time, and it will place it in the events list. If the scheduler is the last one in the events list, the scheduler will not place any more sched_run events. Opposite behaviour at this point would lead to infinitely keep creating sched_run events.
5.5. Events

*Events* implement three functions and their functionality is as following:

**event_add()**: This is the function that accepts as argument a *linkedlist*, usually the *events* list, and adds the object to this list using *linkedlist::add()* function.

**event_remove()**: As above, it takes as argument a *linkedlist* and removes the object to this list using *linkedlist::remove()* function.

Additionally, *events* implement two constructors, one used for constructing any *event* other than *sched_run*, and one for constructing a *sched_run* object. Their difference appears in the data type of the id that the *sched_run event* takes. While the input data requests a string id for each job created, it has been chosen that the *scheduler* should implement its ID using an integer. This happens due to simplicity, since IDs need to increase every time the *scheduler* is called and in the case of strings, string appending operations would be needed.

The constructors are as follows:

**Events()**: This is the basic constructor for constructing an *event* object. Its target is to build an *event* object containing information about what kind of *event* this is, what is the job ID it refers to and what time this action takes place. The ID of the job is of string type, as instructed by HPCx data.

**Events()**: This constructor performs exactly the same operation as the above one, but it’s only called by the *scheduler*. Its difference is that the ID property in this case is of type integer. This happens due to the fact that the *scheduler* has to increment its ID value every time it creates an *event* of type *sched_run*.

5.6. Job

*Job* is one of the most complicated classes of this application. The functions implemented here are as following:

**job()**: This is the basic constructor for this class. *Job* is meant to contain all data inserted from the file, so it has eight arguments that represent all properties of a *job*, as it is read in. The constructor calls *cpu_def* for defining CPUs as appropriate, and additionally it calculates the *CPUtime*, which is the time this *job* ran for. Calculation takes place by subtracting the completion time from the start one.
**Cpu_def():** The main purpose of this function is that, since HPCx can only charge processor sets of 16, is to round the CPU requirement of a job to the closer upper bound of multiple of 16.

**Job_add():** This is the function that accepts as argument a linkedlist, in this application, the queue or the job list, and adds the object to this list using linkedlist::add() function.

**Job_remove():** As above, it takes as argument a linkedlist and removes the object from this list using linkedlist::remove () function.

**event_jobstart();** This function is meant to construct a start event for a job that has been allowed to execute. This will only be called through the scheduler, when it decides that an object can execute. It constructs an event with the time that the scheduler granted this job for executing. This way we can see the time that a job started running. In the end it will store this start event in the events list and will also call the event_jobend() function, described next.

**event_jobend();** This function has the purpose of creating the event that notifies the system that a job finishes running. The time that this event takes is the time that the start event had, plus the time this job ran for, information fed by the input. It eventually stores this end event in the events list.

**before();** before compares the submit time of the object that called it to the submit time of the object that it takes as an argument. It’s mainly used from the linkedlist class to determine where a node should be inserted, since sorting takes place according to the submission time.

**print();** print() takes care of printing the actual output of the simulation which is a specific amount of information for every job. Details on this information can be found in the results chapter, in section 6.3. Comparisons and understandings of this output are discussed in section 7.2.

### 5.7. Queue

The queue class implements three functions, one for creating a queue with specific CPU and time properties, one for retrieving the appropriate queue for a job to go in, and one to actually store the job in it.

Due to time shortage these functions have not been yet utilised, and the application uses a linkedlist for storing queued objects. Therefore, all job objects go in one queue. Nevertheless, since some code for achieving multiple queue generation and control has been developed, it is presented later. The functions implemented are as following:
**Queue_make()**: A function that takes as arguments a name, a CPU upper and lower limit
and a time limit that will be used for defining a queue. This function returns a linked list
which will be the actual queue.

**Queue_def()**: This function will take as argument the table of queues as long as the CPU
and time properties of the job towards queuing. It is meant to return the queue in which a
job belongs.

**Queue_object()**: This is meant to insert the object in the list.

Implemented functions are not ready to be included in the implementation and exhibit
functionality. Compilation has no objection including them, some logic problems exist
though, and additional effort is needed in order for them to work. Strict time constraints,
is the reason why full functionality was not achieved for this piece of code.

### 5.9. Main

*Main* is the part that coordinates the other parts. Here, the *system* as well as the *scheduler*
are initialised and the main iterative loop takes place. The whole input process and object
creation concerning this takes place here. *Main* runs through the *events* list and takes
actions according to the *event* encountered. This is the part of the application that notifies
the *system* for a job using or freeing CPUs. Throughout the whole processing loop, an
output file stream remains open and closes only when the loop exits. At every
*job_end* event, the line concerning this object in the *queue* is printed out to a file, forming
the output data. Additionally, this is where the overall time the application takes for
executing is measured.
Chapter 6: Data Description

6.1. HPCx Data Description

The HPCx Supercomputing Service keeps record of all jobs having run on it since its beginning of service. The amount of records stored in its database is approximately one million and is continuously increasing. There’s a large information set concerning every job having ran there, describing the job itself, the user and the discounts they received, as well as information on what part of the machine was used for executing the task and for how long.

Further down we will briefly analyze the data as provided by the database of HPCx:

**JobstepLogID:** It represents the unique ID of a task and it’s being incremented by one for every job that executes on the machine.

**MachineName:** This holds the name of the machine the job executed on.

**UserName:** It represents the UserName of the User who owns the job.

**JobstepNum:** The job “Step” inside a job submission script.

**JobName:** The actual name of the job submitted.

**BudgetCode:** BudgetCode is the CPU budget that each user has for executing jobs on the machine. Its containing is the same as the UserName, since the budget is owned by the user, and quantifies his ability to run jobs on the system.

**NumCPUs:** This is the number of CPUs that a task required and ran on.

**Class:** This represents the queue class that the job entered. It has to do with its CPU and time requirements, as well as its type.

**Type:** This is the type of the job. Type 2 represents a parallel task and Type 3 a serial one.

**SubmittedTimestamp:** This is the Timestamp (UNIX time) that a job was submitted.
StartedTimestamp: This is the time at which a job started running.

CompletedTimestamp: The time that a job completed running.

ResidencyTime: This is the time for which the job remained in the machine. This represents all the executing life cycle of a job in the system, calculated by the time it ran for, multiplied by the number of processors asked for execution.

CPUTime: CPUtime represents the actual time of CPUs that a task occupied for its execution.

CompletionCode: This code represents how a job completed. That is, if it finished normally, if it crashed or if it was killed due to excess of anticipated time (see later). 0 stands for normal finishing.

Discount: This is the amount of discount units that a user was granted for a specific job. Discount may have occurred due to delayed starts, or job sizes.

MachineID: This represents the ID of the machine that the task ran on.

ProjectID: This is the unique ID of the project a task belongs to.

UserID: Represents the ID of the user.

RawSUCost: This represents the cost of a job in terms of AU’s

DiscountRate: This is the rate of discount that applies on a user for a specific job.

ExecutableID: This represents the ID of the executable file.

ResourcePoolID: This is the identity of the resources pool that the job ran on.

The above data is missing one important piece of information. As it has been mentioned before, for the sake of efficiency in scheduling, every job submitted has to have an expected upper limit of execution time given by the user. This is a time for which the user expects his job to run for, and after which a job will get killed if not finished. This piece of data is going to be called wall_clock_time_limit from now on.

6.2. Input Description

The data that is used as input derives from the data analysed in the HPCx data analysis section. It is a set extracted from above and should be able to serve all needs of the simulator for its present shape, as long as for the most of the future additions expected to
take place. Further down we will analyse the data set taken from HPCx, as well as how each piece of information is used and where it should be needed.

**JobstepLogID**: As described above, JobstepLogID represents the unique ID of a task. This is the standard way for identifying a *job* in our application.

**JobName**: This represents the actual name of the *job* submitted. This data is used for input as an extra identification point of the *job* towards scheduling.

**UserName**: This information represents the username of the owner of the task.

**NumCPUs**: This is the number of CPUs that a task required and ran on. This is used for checking on whether a task can run or not, as well as for updating the system when a *job* starts or finishes execution.

**SubmittedTimestamp**: This is the Timestamp when a *job* was submitted. This is very important information for sorting the *jobs* in the correct order so that they are considered in the correct order.

**StartedTimestamp**: This is the time at which a *job* started running. This information changes during simulation since the application is expected to either decrease or increase the time that a task waits to be executed.

**CompletedTimestamp**: The time that a *job* completed running is expected to be the time that a *job* started running added to the CPUTime. This information changes during simulation since if a *job* starts earlier it is going to run for the same amount of time and it is respectively going to complete at an earlier time.

**ResidencyTime**: This is the time for which the *job* remained running in the *machine*. It is the time that a *job* ran for multiplied by the number of CPUs occupied for it.

**CPUTime**: CPUtime represents the actual time of CPUs that a task occupied for its execution. This means that CPUTime is, in most cases, the actual difference between the start and the complete timestamps. This piece of information is used for forming the end time of a *job* according to its simulation start time, which will probably already differ from the start time of the input file corresponding to this *job*. Due to the fact that this piece of information was not included in the data received, it is being calculated and stored in the *job* constructor.

**wall_clock_time_limit**: This piece of information is inserted, but, for the present version of the simulator, is not used. It is expected to be needed when *scheduler* will be able to calculate complex queuing orders, e.g. in the case of a backfilling policy, and move on to executing other tasks scheduled according to it. It is also considered as very valuable.
data, since it can reveal the trends users adopt for anticipating times and thus assisting in the overall work for achieving efficiency in batch scheduling in shared resources.

The data used in this simulation comes from the development region [2] of HPCx, and are from queues par16, par32, par64 and par128. Input used for the simulation made here can be found in Appendix B.

6.3. Output Description

The output of the simulation is stored in a file called output.dat. Every line in this file represents a job. The output contains the following information:

**ID:** This is the ID of the job.

**Submit:** Represents the submit time of the job.

**Start:** This is the time that the job started running in the simulation.

**End:** The time that the job finished running.

**CPUs:** The amount of CPUs that the job requested.

Submission time remains the same as the input, while start and end times change depicting the way the job was confronted by the scheduler. Before the iterative loop in main starts, an output stream buffer opens and stays open until the loop exits. Each line is recorded to file when a job finishes executing, therefore it is the last action taken when a job_end event is encountered. File is delivered sorted by completion time; since write happens upon job_end events. Output produced from the simulation can be found in Appendix C.
Chapter 7: Results

7.1. Simulation

Simulation took place with a small amount of data no more than 79 records that represent the computing activity of HPCx for 1st of June 2008. While running, the simulation produces a respectable size of screen output that has been used for debugging purposes and has remained there for a quick view of the application flow. Its execution time is roughly 0.03 seconds.

7.2. Results Discussion

The simulator implements basic functionality using a space sharing policy. In figure 14 we can see an instance of the simulation. The faded colours represent jobs that were running at the moment this snapshot was created. The beginning of time is the time the scheduler runs for first time in this instance. The scheduler runs every 1000 seconds, time that represents a time slot, and schedules whichever job is queued and can execute.

The dataset is quite small and, during simulation, queue is not always populated. Therefore, even if the dataset is appropriate for exploiting the system as much as a space sharing policy allows, important gaps occur. This could change if we redefine wider time than 1000s slots, since more jobs will submit while the scheduler is sleeping. This would offer a totally different image for this use case, though more jobs would have to wait longer before they queue. A solution well satisfying both job waiting time and periodicity of scheduler execution will bring the application in its top scheduling performance.

Results seen in figure 14 are as expected for a simple space sharing policy. Potential existence of a backfilling strategy would bring a noticeable performance boost, even with this dataset, where a big amount of small jobs could lean forward, thus minimising queuing time. Applying ideas concerning the scheduler’s upgrading, as analyzed in section 8.2, are considered to bring major scheduling performance impact.
Figure 14 - Instance of scheduling activity
Chapter 8: Conclusions

8.1. Conclusions

This dissertation made an attempt on addressing the issue of efficient batch scheduling on shared HPC resources. The approach in tackling the problems occurring from this was to build an event driven simulator and process historical data of the HPCx service in order to mimic and improve its existing scheduling performance. The metrics adopted in this focused on improving the scheduling throughput, compared to how the present configuration performs. Time available for succeeding these targets proved to be insufficient; nevertheless, a simulator implementing basic functionality, as well as plenty of ideas for future work, was developed. Simulation took place with a fairly low amount of data and, even if results restrict an extensive discussion, understandings developed lead to assertive conclusions for the prosperous future of batch scheduling procedures’ efficiency.

8.2. Future additions

The functionality level of the simulator implements the most basic form of scheduling, adopting the space sharing technique. The number of upgrades and the level of improvement it can take are quite considerable. This section discusses the potential additions to the application, as well as ways that this can be implemented according to the scheduler’s design.

Time sharing: As it has been described in the Background section, time sharing is the second most basic technique for scheduling, where tasks practically share execution time on a standard CPU mapping. In this, jobs experience short wait times so that others execute on the processes freed up by it. This way, tasks share time of processors.

This technique can cause difficulty while co-existing with a space sharing policy. It is not employed on HPCx and the reason is that queue processing can become even more complex when having jobs rescheduling parts of them. There’s no perspective, when employing a combination of these two techniques, for reaching any better performance than space sharing achieves when used solely. Thus, there’s is no reasoning in attempting to employ it. Time sharing can only be suggested as an alternative to a space sharing policy.
In this application this could be implemented as following: consider a *job* that, if it was to run on some *job*'s offered time, both this and the one offering time, can finish in a reasonable, according to submission and anticipation, time. This takes place by *checkpointing* the previous *job* (the one to offer time) for time equal to the other’s `wall_clock_limit`; only if the other is set to execute until finishing. *Checkpointing* is quite different from the occasional *checkpointing* technique to be described further down, and it suggests a full back up of data and status of a *job* running. A task, upon resuming, can read this file and bring itself to executing at exactly the same point as before the checkpoint occurred.

**Backfill:** *Backfilling* is very famous in modern resources managers and is implemented in two different versions: *aggressive* and *conservative backfilling*, whose exact logic has been described in the background chapter. Making this potential upgrade more application specific I should note that every *job* object according to *conservative*, or only the one in the *head* of the *queue*, according to *aggressive*, should have a reservation number.

The *scheduler* should be able to check every *job* owning a reservation number, on whether it can execute without delaying the projected start of the others, or the first one’s in the *queue* for aggressiveness. Considering that the scheduler, upon every execution, is fully aware of the *system* status and its available processors, it can search for a *job* that can be backfilled.

*Backfilling* was the first objective after basic functionality with *space sharing* policy. Upgrading the simulator this way, is expected to give an important rise of queuing measuring times.

**Migration:** In this policy, the *system* is divided in to several execution areas, each one of which has defined restrictions on what *jobs* can go in it. More details about this policy can be found in the background chapter. In our application, the *system* can be divided in to several areas by defining a number of processors that define each one. So, for example, on a *system* with 2048 CPUs that needs to implement three different areas, the first 256 CPUs can define the first area, the next 768 define the second, and the last 1024 can define the third one.

When a *job* is submitted it can be considered for queuing in a specific *region* according to its CPU and time requirements. More detailed information can be found in the background chapter. When an area is judged to be too crowded, a *job* can move to another, more available area, since this action is going to run this *job* sooner while no other *job* will be delayed.
In HPCx, if necessity for migration occurs it takes place manually, by the administration team. My suggestion about a potential implementation would be that a crowded area is considered to be the one that has at least double workload size than another one. This cannot be restricted to CPU load, since due to the difference of the areas, one may have a task asking for 1024 CPUs and another one may have 50 jobs asking for 4 processors each. In this case there might be a need for some of the small jobs to migrate in the capacity area, only according to when they, as well as the big one, are expected to finish; also depending on if the overall scheduling process benefits from this. For every scenario, it is considered pretty unlikely that a big task is going to migrate in a lower requirements area.

**Soft killing:** As it has been mentioned before, when a user submits a job he is asked to submit a `wall_clock_time_limit` that is the time he sets as an upper limit of execution. Users are being asked to submit reasonable time limits, so that their entire budget is not burn, in case their program crashes or hangs for any reason. If CPUs are freed earlier, they are going to stay idle for some time unless a successful backfill operation occurs. In the most successful cases, a job will finish shortly before the anticipation time is reached—a scenario in which the user as well as the scheduler are both satisfied.

If a job goes beyond its `wall_clock_time_limit`, it simply breaks and all data is lost. Soft killing suggests that the user is asked to submit a time value, a bit smaller than the `wall_clock_time_limit`, as well as the `wall_clock_time_limit` time itself. If this time value is reached and the application is still running, the system has to break the main iterative loop and let it exit normally before the `wall_clock_time_limit` is reached and all data is lost. From one point of view, this way the program is not finishing as the programmer expected, but the important aspect is that it terminates smoothly and delivers results, even if process has not completed.

In our simulator this seems not to be of any significance. In contrast though, it is an addition that can assist in evaluating users’ ability to judge when their application should start terminating and what number of tasks makes use of this “intervention”. Additionally, it can contribute to understanding the trend users follow in their time judgment, so that we can evolve schedulers according to the machine’s users.

**Checkpoint:** This technique does not constitute a scheduling technique, but its purpose is to prevent jobs from experiencing data loss due to any reason. Its logic is to save all data of a job occasionally. That is either because the job is going to be stopped, simulating a soft killing case, or because the job explicitly requests to be check pointed. The system may also perform occasional check pointing by default.

It is a very sensible thought that check pointing has no meaning in this application; since execution does not process the real amount of data and status saving makes no sense. On the other hand though, it can, at first, simulate the time a real system is going to spend on
check pointing for a specific workload, especially since check pointing is a pretty intensive procedure. Additionally, I consider that it offers the ability to statistically determine how often check points’ existence proves to be crucial for jobs, and whether it is worth employing it or not.

**Reshapable Jobs:** The basic idea says that a job should be able to alter its resources requirement, in case this is judged as appropriate by the scheduler. Technically, one job running on X number of processors, and now has to offer some of them to another job, should save status (check point), stop and reset the number of processors that it should run on from now on, taking into consideration user fed information. After that, it should start again, having spent a small amount of time on doing that, but now running on \( X / p\_factor \) processes, where \( p\_factor \) is the processor multiplying factor supplied by the user.

Reshaping should mainly happen in specific cases such as when job escalation (a job needs to run within specific time constraints thus having to lean forward in the queue) occurs. Additionally, consider a fairly big job that it’s executing will seriously alter other jobs’ start times. If reshaping applies, it will allow other jobs to execute, while running in a reduced number of processors at the same time. A job should ideally be able of recurring to any CPU size according to needs and demands.

Not all source codes can do this, and this is why users should confirm whether they want their job to be reshapable or not. Source codes, for which testing is not of importance and proved robustness exists, are considered as appropriate cases of codes capable of resisting in any inhibiting of the numerical reproducibility. If the user chooses to submit a reshapable job should also submit a number of information, such as lower/upper bound of processes that it should run on, and what the periodicity (p_factor) of increasing or decreasing CPU numbers should be.

**Alternative simulator triggering:** In this simulator, the simulator triggers every some fixed amount of times and schedules what is already queued. The scheduler could alternatively be woken up by a signal. My suggestion is that a signal can be an event notifying that the summed up CPU properties of the last jobs’ submission fits a completely idle system. This, though, ignores the fact that queue may already be well populated, thus diminishing the meaning of it. An alternative signal could be an event notifying for reaching a specific amount of CPUs freed from job_end events.

In this implementation triggering the scheduler is a very light process. In real systems scheduler consumes more noticeable amount of resources and simulating an alternative implementation for scheduler triggering could be of interest.
8.3. Programming Language Discussion

I discuss here the potential programming languages that this application could have been developed on, as well as C++, on which it was actually developed. I also consider some conceptual differences between languages, as well as their pros and cons from some points of views. The discussion made here takes place in terms of the simulator developed as part of this project.

The main reason that C++ was chosen for the development of this code was not my prior experience, but my will to develop my systems knowledge. C++ is a language that was developed for object oriented programming and implements high level support for abstract data types as well as hardware pointing techniques. It is considered as very challenging and it could be characterized as a widely used and highly controllable programming language.

The alternative language that this project could have been implemented on is Java. Java is a much safer language and has undoubtedly taken over several sections of software development in modern industry. It is also a language that promises compatibility, as well as an extensive wealth of high quality libraries.

C++ offers ways to control memory manually. We are being offered the “raw hardware pointers” mechanism which allows explicitly manipulating memory. This provides the programmer with a full understanding of how an application maps on memory, thus allowing the implementation of ideas in the most efficient way, since one can always be sure on where information is and which entity points at it. As it is normal though, if pointers fail for any reason, the execution of a program can become strange, usually leading to unexpected behaviors and catastrophic errors. Pointer chasing, which practically means multiple accessing in scattered memory locations for retrieving a pointee, as well as memory leaks, that states the loss of a pointee, are the two most ordinary cases of such bugs.

Java on the other hand does not offer hardware pointers thus minimizing the level of hardware control. This practically means that the language is always likely to exhibit the expected behavior of a system designed, in contrary to C++ that provides many more subtleties that may lead programming tripping up the unknown. The price Java applications pay for this good behavior has mostly to do with the performance of the outcome, which is considerably lower in comparison to programming languages such as C++ or C.

Given the fact that the differences between these two languages were clear from the beginning, language choice took place according to where the greatest educational profit could be gained. It was judged therefore that C++ should be employed for this implementation.
Here, I should note that judging from the time, and the quality of it, spent on debugging stage, choosing Java would have brought this application in a more advanced level. This means that we would have made a second iteration on the development model, thus implementing the aggressive policy, at the least of my expectations. Choosing C++ though led me to constructing a much better understanding of systems handling. I consider that I have gained invaluable knowledge and have expanded my skills in the point I could only imagine before this project.
Main.cpp

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include "linkedlist.h"
#include "system.h"
#include "job.h"
#include "events.h"
#include "queue.h"
#include "scheduler.h"
#include <ctime>
using namespace std;
int main(int)
{
    std::clock_t starttime=std::clock();
    machine sys(CPU);//initialise a system(CPU defined in system.h)
    int i=0;
    int j=0;
    int incr=0;
    scheduler *sched=new scheduler(&sys);//initialise a scheduler for this system
    linkedlist<events> *eventlist=new linkedlist<events>;
    linkedlist<job> *queuelist=new linkedlist<job>;
    linkedlist<job> *joblist=new linkedlist<job>;
    //set list lengths to zero/
    eventlist->length=0;
    queuelist->length=0;
    joblist->length=0;
    //variables to hold individual strings coming from line input*/
    string jobsteplogid;
    string username;
    unsigned int submit;
    unsigned int start;
    unsigned int end;
    int numcpu;
    float residency;
    int anticipation;
    events nodereturn;
    events eventremoved;
```
string line;
/*open an input file stream*/
fstream batchfile("input.txt",ios::in);
/*read in line, create job and event object, add event to events list, change Job_Status from "NS" to "S" and add job to the job list*/
while (!batchfile.eof() )
{
    getline(batchfile,line);
    istringstream is(line);
    is>>jobsteplogid;
    is>>username;
    is>>submit;
    is>>start;
    is>>end;
    is>>numcpu;
    is>>residency;
    is>>anticipation;

    job *j = new job(jobsteplogid,username,submit,start,end,numcpu,residency,anticipation,Job_Status0);
    events *e = new events(Event0,jobsteplogid,submit);
    e->event_add(eventlist);
    j->Job_Status=Job_Status1;
    j->job_add(joblist);
}
batchfile.close();

/*call scheduler for the first time before iteration starts so that a sched_run event is created*/
sched->sched_run(queuelist,eventlist,NULL,NULL);
job jobremoved;
node<events> *eventtmp;
eventtmp=eventlist->head;
node<job> *queuetmp;
queuetmp=queuelist->head;
node<job> *jobtmp;
jobtmp=joblist->head;
ofstream output("output.dat");
for (i=0;i<eventlist->length; i++)
{
    /*If event==Job_Submit change Job_Status into "Q" and add job to the queue list. Additionally, move to the next node of job list so that the next Job_Submit event corresponds to the correct object from there*/
    if (eventtmp->data->Events_Type==Event0)
    {
        cout<<"event Job Submit time<"<<eventtmp->data->Events_Job_Submit_time<<endl;
        cout<<"job submit corresponding Jobtmp:"<<jobtmp->data->Events_Job_Submit_time<<endl;
        jobtmp->data->Job_Status=Job_Status2;
jobtmp->data->job_add(queuelist);
if (jobtmp->next!=0)
{
    jobtmp=jobtmp->next;
}
else
{
    cout<<"no more jobs in joblist"<<endl;
    jobtmp=jobtmp;
}

//If event==Job_Start find the corresponding job in the queue list, update the system with the CPU number of the job and change the job status to "R"*/
else if (eventtmp->data->Events_Type==Event1)
{
    cout<<"event job_start":"<<eventtmp->data->Events_Job_Submit_time<<endl;
    queuetmp=queuelist->search(eventtmp->data->Events_Job_Step_Log_ID);
    sys.sys_update(queuetmp->data->Events_Job_CPUs,Event1);
    jobtmp->data->Job_Status=Job_Status3;
    cout<<"start of id":"<<eventtmp->data->Events_Job_Submit_time<<"machine":"<<sys.sys_cpu_no_available<<endl;
}
//If event==Job_End retrieve the corresponding object from the queue list, update the system and print the output*/
else if (eventtmp->data->Events_Type==Event2)
{
    cout<<"event job_end":"<<eventtmp->data->Events_Job_Step_Log_ID<<"
"<<eventtmp->data->Events_Job_Submit_time<<endl;
    queuetmp=queuelist->search(eventtmp->data->Events_Job_Step_Log_ID);
    sys.sys_update(queuetmp->data->Events_Job_CPUs,Event2);
    cout<<"endof:"<<eventtmp->data->Events_Job_Submit_time<<"machine:"<<sys.sys_cpu_no_available<<endl;
    queuetmp->data->print(output);
}
//If event=="Sched_Run" call scheduler*/
else if (eventtmp->data->Events_Type==Event3)
{
    cout<<"event sched_run":"<<eventtmp->data->Events_Job_Step_Log_ID<<"
"<<eventtmp->data->Events_Job_Submit_time<<endl;
    unsigned int &time=eventtmp->data->Events_Job_Submit_time;
    sched->sched_run(queuelist,eventlist,&eventlist->length,time);
}
if (eventtmp->next!=0)
    eventtmp=eventtmp->next;
else break;

output.close();
cout<<"main returned!"<<endl;
cout<<"execution time:"<<(std::clock() -starttime)/(double)CLOCKS_PER_SEC<<endl;
return 0;
# Node.h

```c++
#ifndef NODE_H
#define NODE_H

/* word template symbolises that the class following
   is parameterised and Symbol T plays the role of the type.
   This parameterised class gives one node of the list,
   consisted of the data and the pointer to the next & previous node parts */
template<class T>
class node
{
    public:
    template <class U> friend class linkedlist;

    /*methods*/
    /*class node implements a default constructor
       and a custom constructor that receives template
       variable and sets it as the data part*/
    node(){}
    node(T * target)
    {
        data=target;
        next=NULL;
        previous=NULL;
    }

    /*properties*/
    /*data is the variable of type T which is going to store
       the actual data of a node. next and previous are going to
       respectively store the pointers to next and previous nodes*/
    T *data;
    node<T> *next;
    node<T> *previous;
};
#endif
```

# LinkedList.h

```c++
#ifndef LINKEDLIST_H
#define LINKEDLIST_H

#include <iostream>
#include <string>
#include "node.h"
using namespace std;

/* linked list is the class to create, preserve and control
   the main data storage structure of this application */
template <class U>
class linkedlist
{
    public:

    private:
```

50
/*methods*/
/*The constructor is responsible for reserving memory
 for an instance of this object to be created*/
linkedlist();
/*function for adding a node in the linked list*/
void add(U *data);
/*remove is used for removing a node from the list*/
void remove(U *data);
/*search is used for retrieving the an event's
 corresponding job object in the queue list*/
node<U> * search(string id);
/*attributes*/
/*head and tail represent the top and the bottom node of a list*/
node<U> * head,*tail;
/*length keeps a count of the nodes in a list*/
int length;
private:
 int i;
};

/*implements the default constructor setting head and tail to NULL*/
template <class U>
linkedlist<U>::linkedlist()
{
 head=NULL;
tail=NULL;
};
template <class U>
void linkedlist<U>::add(U *dat)
{
 node<U> *addnode=new node<U>(dat);
 /*if list is empty, the node to be stored
 will become the head and the tail of it.
 list length will be increased by one*/
 if (head==NULL)
  {
   head=tail= addnode;
   length++;
   return;
  }
 /*if node to be inserted is smaller than head
 then make it a head and point its next to the
 next node, until previously head. list length
 will be increased by one*/
 if(addnode->data->before(head->data))
  {
   addnode->next=head;
   head = addnode;
   length++;
   return;
   

```c
node<U> *p;
/*for as long as next part is not NULL and
and addnode is not smaller, go to next node.
when this exits, p will point to the first node
found to have smaller submit time than addnode*/
for( p=head; p->next != NULL && ! addnode->data->before(p->next->data); p=p->next);
/*addnode after that will add itself after p*/
  addnode->next=p->next;
  addnode->previous=p;
p->next=addnode;
/*if next part is inexistent, addnode becomes tail*/
  if( addnode->next == NULL )
    { 
      tail=addnode;
    }
/*else the next part must be informed that
its previous is the addnode*/
else
  { 
    addnode->next->previous=addnode;
  }
/*increase list length*/
  length++;
return;
}
```
according to the data part given as an argument*/
for(p=head; p!=NULL && p->data != dat ; p=p->next);
/*if node found is not NULL let the until now previous
node know about its new next point. If node to remove was
tail, the tail is now its previous node. Else, the previous
part of the next node should point to its previous node, thus
successfully disconnecting the node from the list*/
if( p!= NULL )
{
    p->previous->next = p->next;
if( p->next == NULL )
{
    tail=p->previous;
}
else
{
    p->next->previous=p->previous;
}
delete p;
}
/*length is decreased by one*/
length--;
}

template <class U>
node<U>* linkedlist<U>::search(string id)
{
    node<U> *tmp;
    tmp=head;
    /*while node is not empty if the id is the same
as the one taken through argument, return node found
if the next node is not empty, move on checking on
the next.If no next part exists, last node encountered
returns. It is unlikely that no node will be found,since
a job start/end event that calls search, has a
corresponding node for sure*/
while (tmp!=NULL)
{
    if (tmp->data->Events_Job_STEP_Log_ID==id)
    {
        break;
    }
    if (tmp->next!=0)
    {
        tmp=tmp->next;
    }
    else if (tmp->data->Events_Job_STEP_Log_ID==id)
    {
        break;
    }
}
Events.h

#include <iostream>
/*Class events stores all events taking place in the system. For beginning these can be job_submit, job_start, job_end and Sched_Run. Events inherit from class job. This means that all properties and functionality can be exploited from this class.*/
using namespace std;
class events: public job {
public:
    /*methods*/
    events(); // default constructor
    /* events constructor is an alternative than the default
    and is used for importing job objects from input*/
    events(string type, string jobsteplogid, unsigned int time);
    /* this constructor is used for constructing a scheduler event.
    Its difference lies on the jobsteplogid, which is a simple integer,
    instead of input that is string*/
    events(string type, int jobsteplogid, unsigned int time);
    /*methods event_add/remove are used for adding
    and removing event objects to the events list*/
    void event_add(linkedlist<events> *list);
    void event_remove(linkedlist<events> *list);
    /*attributes*/
    /*holds the type of an event, such as submit, start, finish*/
    string Events_Type;
    int Events_Sched_Strip_Log_ID;
};
#endif

events.cpp

#include <iostream>
#include <string>
#include "linkedlist.h"
#include "job.h"
#include "events.h"
using std::string;

/*method implementations*/
events::events() {}
void events::events(string type, string jobsteplogid, unsigned int time) {
    Events_Type=type;
    Events_Job_Step_Log_ID=jobsteplogid;
    Events_Job_Submit_time=time;
}
cout << "event of type" << Events_Type << "with ID:" << Events_Job_Sched_Log_ID << " was created" << endl;
}

events::events(string type, int jobsteplogid, unsigned int time)
{
    Events_Type = type;
    Events_Sched_Log_ID = jobsteplogid;
    Events_Job_Submit_time = time;
}

void events::event_add(linkedlist<events> *list)
{
    list->add(this);
}

void events::event_remove(linkedlist<events> *list)
{
    list->remove(this);
}

Job.h

#ifndef JOB_H
#define JOB_H
#include <string>
/* definitions of job statuses */
#define Job_Status0 "NS"
#define Job_Status1 "S"
#define Job_Status2 "Q"
#define Job_Status3 "R"
/* definitions of Event cases */
#define Event0 "Job_submit"
#define Event1 "Job_start"
#define Event2 "Job_end"
#define Event3 "Sched_Run" // keeps the job_status
using namespace std;

/* job has Class events as friend, and is going to represent job objects as long as implement methods for taking several actions on them */
class job
{
    friend class events;
    public:
    /* methods */
    /* default constructor */
    job();
    /* alternative constructor for storing the objects read from file input */
    job(string id, string user, int submit, int start, int end, int cpurequired, float residency, int anticipation, string status);
/*function for defining a cpu number within
constraints such as machine or node size*/
int cpu_def(int cpu);

/*job_add/remove are the functions that
add/remove job objects to the list*/
void job_add(linkedlist<job> *list);
void job_remove(linkedlist<job> *list);

/*event_jobstart/jobend are the constructors that build
start and end events for a job when it gets scheduled
by the scheduler. These are meant to store in the events list*/
unsigned int event_jobstart(job start, linkedlist<events> *list, int time);
int event_jobend(job end, linkedlist<events> *list, unsigned int finish);

/*before is used for determining whether a job's
submit time property is smaller than the peer's one
It will be used for determining list insertion positions*/
int before(const job *peer);

/*prints some specific data from a job object*/
void print(ostream &s);

/*properties*/
string Events_Job_Step_Log_ID;//holds the step of the Log of HPCx
string Events_Job_Username;//holds the username of the user associated
unsigned int Events_Job_Submit_time;//holds the submit time
unsigned int Events_Job_Start_time;//holds the start time
unsigned int Events_Job_Complete_time;//holds the finish time
int Events_Job_CPUs;//holds the number of CPUs a job asks for
float Events_Resid_time;//holds the residence time
int Events_Anticip_time;//holds the wall_clock_limit
string Job_Status;//keeps the status of a job(Submitted, Running, Queued)
int Events_CPU_time;

#endif

Job.cpp
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include "linkedlist.h"
#include "job.h"
#include "events.h"
using std::cout;
using std::string;

/*method implementations*/
job::job(){}
/* Implementation of the constructor that reads in a record
and stores all data in the corresponding variables*/
job::job(string id, string user, int submit, int start, int end, int
cpurequired, float residency, int anticipation, string status)
{
    int cpu=cpu_def(cpurequired);
Events_Job.Step_Log_ID=id;
Events_Job.Username=user;
Events_Job.Submit_time=submit;
Events_Job.Start_time=start;
Events_Job.Complete_time=end;
Events_Job_CPUs=cpu;
Events_Resid_time=residency;
Events_Anticip_time=anticipation;
Job.Status=status;
if (start!=0)
    Events_CPU_time=end-start;
else
    Events_CPU_time=0;
cout<<"job object with ID:"<<Events_Job.Step_Log_ID<<"and cpu:"<<Events_Job_CPUs<<"cputime:"<<Events_CPU_time<<" was created"<<endl;

int job::cpu_def(int cpu)
{
    int i;
    int limit=192;
    int node=16;
    /*if cpu==0, return it, scheduler will deal with it*/
    if (cpu==0)
        return cpu;
    /*if cpu is more than 128, return 128*/
    else if(cpu>limit)
    {
        cpu=limit;
        return cpu;
    }
    /*for the 8 cases of CPU charge (16,32,48...128)
    check CPU number and round it up to its closest
    16 multiple*/
    for (i=1;i<9;i++)
    {
        if (cpu==node*i)
            break;
        else if (cpu>node*i)
            continue;
        else if (cpu<node*i)
        {
            cpu=node*i;
            break;
        }
    }
    return cpu;
}

/*Add the job to the list*/
void job::job_add(linkedlist<job> *list)//adds a job to the job list
{
list->add(this);
}

void job::job_remove(linkedlist<job> *list) //removes a job from the list
{
    list->remove(this);
}

unsigned int job::event_jobstart(job start, linkedlist<events> *list, int time) //creates a start event and adds it in the eventlist
{
    /*set event time equal to the time the scheduler scheduled the event, constructs a start event and inserts it in the list. Additionally, it sets the finish time and it passes it to the jobend event, which is called straight after. Function returns the finish time*/
    start.Events_Job_Start_time = time;
    events *startevent = new events(Event1, start.Events_Job_Start_time, time);
    startevent->event_add(list); //add the job start event on the head of the list
    int finish = start.Events_Job_Start_time + start.Events_CPU_time;
    cout << "finish time:" << finish << endl;
    event_jobend(start, list, finish);
    return finish;
}

int job::event_jobend(job end, linkedlist<events> *list, unsigned int finish)
{
    /*create the end event with the finish time given and store it in the list*/
    events *endevent = new events(Event2, end.Events_Job_Start_time, finish);
    endevent->event_add(list);
    return 0;
}

int job::before(const job *peer)
{
    /* return 1 if comparison is true
    this is used for the linked list to define here an object should be inserted*/
    return Events_Job_Submit_time < peer->Events_Job_Submit_time;
}

/*print the specified data in a file*/
void job::print(ostream &s)
{
    s << "ID:" << Events_Job_Step_Log_ID << " Submit:" << Events_Job_Submit_time << " Start:" << Events_Job_Start_time << " End:" << Events_Job_Complete_time << " CPUs:" << Events_Job_CPUs << endl;
}

Queue.h
#ifndef QUEUE_H
#define QUEUE_H
#define QMAX 3
#include <string>
#include "linkedlist.h"
#include "job.h"
using namespace std;

/*queue is the class that is going to define
 queue classes and functions*/
class queue
{
  public:
  /*methods*/
  queue(); //default constructor
  linkedlist<job>* queue_make(string name, int from, int upto, int time); //construct a queue class
  int queue_def(queue q[], int cpus, int time);
  void queue_object(linkedlist<job> *q, job object);
  string queue_name; //holds the name of the queue
  int queue_limit_from;
  int queue_limit_upto;
  int queue_time;

  private:
  int i;
};
#endif

Queue.cpp
#include "queue.h"
using namespace std;

/*method implementations*/
queue::queue(); //default constructor

linkedlist<job>* queue::queue_make(string name, int from, int upto int time) //construct a queue class
{
  linkedlist<job> *list=new linkedlist<job>;
  queue_name=name;
  queue_limit_from=from;
  queue_limit_upto=upto;
  queue_time=time;
  return list;
}

int queue::queue_def(queue q[QMAX], int cpus, int time)
{
  for (i=0; i<QMAX; i++)
  {
if (cpus>q[i]->queue_limit_from && cpus<q[i]->queue_limit_upto &&
time<=queue_time) {
    cout<<"returned q no:"<<i;
    return i;
} else i++;
}

void queue::queue_object(linkedlist<job> *q, job* object) {
    node<job> *tmp=new node<job>;
    tmp->data=object;
    q->add(object);
}

Scheduler.h
#include <string>
define forward 1000
using namespace std;

/*scheduler class is the one to schedule jobs on system.*/
class scheduler {
public:
    /*methods*/
    /*scheduler constructor takes as an argument the system already created. It counts on the notion of one scheduler per system*/
    scheduler(machine *m) { mach=m; }
    /*Sched_run is the mechanism that runs the scheduling procedure. It takes the events and the queue lists, as long as the time it was called and the time for which is called. This will be useful when running the scheduler for first time thus only wanting it to reschedule.*/
    void sched_run(linkedlist<job> *qlist, linkedlist<events> *elist,int *count_jobs, int time);
    /*This is the function that will take care of inserting a new scheduler_run event in the events list every some fixed time*/
    void sched_reposition(int timecalled, linkedlist<events> *list);
    /*public properties*/
    int incr;
private:
    /*private properties*/
    int i;
    int j;
/* holds the integer id of a scheduler
   (instead of the string id of events) */
int sched_steplogid;

/* holds the considered available processors for scheduling.
   Since machine does not change through scheduler, but through
   start/end events, it helps for the scheduler to count the
   decreasing availability while scheduling */
int availability;
node<job>*qnode; /* will hold the node read towards scheduling
job *qdata; /* holds the data part of qnode
events *sched_event;
unsigned int time;
unsigned int finish;
machine *mach;

Scheduler.cpp
#include <string>
#include "linkedlist.h"
#include "system.h"
#include "job.h"
#include "events.h"
#include "scheduler.h"
using namespace std;

/* method implementations */
void scheduler::sched_run(linkedlist<job> *qlist, linkedlist<events> *elist, int *countjobs, int time) {
    /* if its the first job that the scheduler is called, call
       reposition for placing first scheduler event in the events list */
    if (count jobs == NULL) {
        sched_reposition(0, elist);
        cout << "positioned for 1st time";
        return;
    }

    qnode = qlist->head;
    cout << incr << incr;
    /* set local variable availability equal to the system available
       This happens because we don't want to decrease the actual machine
       availability, since jobs are not actually starting but just scheduled */
    availability = mach->sys_cpu_no_available;
    /* While availability stays above 0, add queued jobs from the point
       started and on. Call a job start event for each one and change queue
       status for this job. Increase the number of jobs having gone through
       scheduling and set the considered for scheduling availability */
    while (availability > 0) {
        qdata = qnode->data;

if (qdata->Job_Status==Job_Status2 && qdata->Events_Job_CPUs!=0 && qdata->Events_Job_CPUs<=availability)
    {
        cout<<"adding...machine before:"<mach->sys_cpu_no_available<<endl;
        qdata->Job_Status=Job_Status3;
        qdata->Events_Job_Start_time=time;
        finish=qdata->event_jobstart(*qdata,elist,time);
        qdata->Events_Job_Complete_time=finish;
        incr++;
        cout<<"increased:"<<incr<<endl;
        availability=availability - qdata->Events_Job_CPUs;
        cout<<"machine after:"<mach->sys_cpu_no_available<<endl;
        cout<<"considered availability:"<availability<<endl;
    }
else if (qnode->next!=0)
    {
        qnode=qnode->next;
    }
else if (qnode->next==0)
    {
        cout<<"qnode->next=0, repositioning self"<<endl;
        break;
    }
    /*if next node is not NULL, move on to the next node,
    else call reposition function*/
    sched_reposition(1,elist);//call repositioning for every other
time than the first with value 1
    return;
}

void scheduler::sched_reposition(int timecalled, linkedlist<events> *list)
{
    /*if its the first call, set id=0 and time equal
    the head of the list. Create the new Sched_Run event
    and increase the id number for the next Sched_Run
    event to take it*/
    if (timecalled==0)
    {
        sched_steplogid=0;
        time=list->head->data->Events_Job_Submit_time;
        sched_event=new events(Event3,sched_steplogid,time);
        sched_steplogid++;
    }
    /*if scheduler is the last event, return function,
    so that we avoid chain repositioning without meaning*/
    else if (list->tail->data->Events_Type=="Sched_Run")
    {
        cout<<"scheduler is last"<<endl;
        return;
/*else increase time by a fixed amount of seconds, 
create a sched event and increase the id by 1*/
else
{
    cout<<"in else of sched_reposition"<<endl;
    time=time+forward;
    sched_event=new events(Event3,sched_steplogid,time);
    sched_steplogid++;
}
/*add event created to the events list and return*/
sched_event->event_add(list);
return;
}

system.h
#include <iostream.h>
#include <string>
#define CPU 192//Define CPU for system to initialise
using namespace std;

/*system is the class that represents the system and contains 
   methods for retrieving status and applying load changes*/
class machine
{
public:
    /*methods*/
    /*machine method is used for initialising a system. 
   It takes an integer, standing for number of CPUs, as an argument*/
    machine(int sys_cpu)
    {
        sys_cpu_no=sys_cpu;
        sys_cpu_no_available=sys_cpu;
        cout<<"system initialised"<<endl;
    }
    /*sys_update is used for updating the system with any changes 
of load occuring from Job_Start and Job_End events*/
    void sys_update(int proc_num, string event);

    /*properties*/
    int sys_cpu_no;//keeps number of total system processors
    int sys_cpu_no_available; //keeps number of free processors of the whole system
};
#endif

system.cpp
#include <iostream.h>
#include "system.h"

/*method implementations*/
/*updates the system CPU property*/
void machine::sys_update(int proc_num, string event)
{
    if (event=="Job_start")
    {
        sys_cpu_no_available=sys_cpu_no_available - proc_num;
    }
    else  //if event=finish
    {
        sys_cpu_no_available=sys_cpu_no_available + proc_num;
    }
}

Makefile
MF= Makefile
CC= pgCC
CFLAGS= -g
EXE= scheduler
SRC= 
    main.cpp 
    system.cpp 
    job.cpp 
    events.cpp 
    scheduler.cpp 
    queue.cpp
### NO NEED TO EDIT BELOW THIS LINE ###
.SUFFIXES: .cpp .o
OBJ= $(SRC:.cpp=.o)
.cpp.o:
    $(CC) $(CFLAGS) -c $<
all: $(EXE)
$(EXE): $(OBJ)
    $(CC) $(CFLAGS) -o $0 $(OBJ) $(LFLAGS)
$(OBJ): $(MF)
clean:
    rm -f $(OBJ) $(EXE) core
clean: clean all
Appendix B: Input data

Here we can see the input data of the simulation, and is contained in input.txt.

<table>
<thead>
<tr>
<th>Time</th>
<th>User</th>
<th>Start Time</th>
<th>End Time</th>
<th>Duration</th>
<th>ID</th>
<th>Count</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>user-1416</td>
<td>1212269983</td>
<td>1212269992</td>
<td>32</td>
<td>1215552</td>
<td>43200</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>user-1297</td>
<td>1212319840</td>
<td>1212319849</td>
<td>64</td>
<td>640</td>
<td>20400</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>user-905</td>
<td>1212315647</td>
<td>1212315665</td>
<td>64</td>
<td>303808</td>
<td>10800</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>user-240</td>
<td>1212329800</td>
<td>1212329802</td>
<td>16</td>
<td>288</td>
<td>2130</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>user-1430</td>
<td>1212296520</td>
<td>1212296524</td>
<td>128</td>
<td>4336384</td>
<td>43200</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>user-1297</td>
<td>1212320322</td>
<td>1212320335</td>
<td>64</td>
<td>702528</td>
<td>20400</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>user-240</td>
<td>1212331781</td>
<td>1212331796</td>
<td>16</td>
<td>288</td>
<td>2130</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>user-240</td>
<td>1212332194</td>
<td>1212332216</td>
<td>16</td>
<td>256</td>
<td>2130</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>user-240</td>
<td>1212332751</td>
<td>1212332770</td>
<td>16</td>
<td>272</td>
<td>2130</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>user-240</td>
<td>1212333659</td>
<td>1212333680</td>
<td>16</td>
<td>288</td>
<td>2130</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>user-240</td>
<td>1212334400</td>
<td>1212334404</td>
<td>16</td>
<td>272</td>
<td>2130</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>user-240</td>
<td>1212335561</td>
<td>1212335584</td>
<td>16</td>
<td>272</td>
<td>2130</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>user-1316</td>
<td>1212338850</td>
<td>1212338878</td>
<td>128</td>
<td>1408</td>
<td>7200</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>user-240</td>
<td>1212344072</td>
<td>1212344126</td>
<td>16</td>
<td>672</td>
<td>2130</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>user-240</td>
<td>1212346418</td>
<td>1212346461</td>
<td>16</td>
<td>672</td>
<td>2130</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>user-240</td>
<td>1212346770</td>
<td>1212346816</td>
<td>16</td>
<td>672</td>
<td>2130</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>user-1357</td>
<td>121238429</td>
<td>121238499</td>
<td>32</td>
<td>690176</td>
<td>57540</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>user-19</td>
<td>1212350028</td>
<td>1212350161</td>
<td>128</td>
<td>16896</td>
<td>900</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>user-19</td>
<td>1212351502</td>
<td>1212352138</td>
<td>128</td>
<td>81024</td>
<td>3600</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>user-19</td>
<td>1212352419</td>
<td>1212352734</td>
<td>128</td>
<td>39808</td>
<td>3600</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>user-19</td>
<td>1212352619</td>
<td>1212352910</td>
<td>128</td>
<td>36992</td>
<td>3600</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>user-19</td>
<td>1212348088</td>
<td>1212354539</td>
<td>0</td>
<td>0</td>
<td>900</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>user-19</td>
<td>1212348558</td>
<td>1212354555</td>
<td>0</td>
<td>0</td>
<td>3600</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>user-19</td>
<td>1212355771</td>
<td>1212356251</td>
<td>128</td>
<td>59776</td>
<td>3600</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>user-240</td>
<td>1212357034</td>
<td>1212357063</td>
<td>16</td>
<td>288</td>
<td>2130</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>user-240</td>
<td>1212357267</td>
<td>1212357319</td>
<td>16</td>
<td>608</td>
<td>2130</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>user-240</td>
<td>1212357830</td>
<td>1212357857</td>
<td>16</td>
<td>224</td>
<td>2130</td>
<td></td>
</tr>
<tr>
<td>l1f401.351324.0</td>
<td>user-240</td>
<td>1212358159</td>
<td>1212358170</td>
<td>1212358212</td>
<td>16</td>
<td>672</td>
<td>2130</td>
</tr>
<tr>
<td>l1f401.351326.0</td>
<td>user-240</td>
<td>1212358392</td>
<td>1212358401</td>
<td>1212358443</td>
<td>16</td>
<td>672</td>
<td>2130</td>
</tr>
<tr>
<td>l1f401.351327.0</td>
<td>user-240</td>
<td>1212358910</td>
<td>1212358911</td>
<td>1212358952</td>
<td>16</td>
<td>656</td>
<td>2130</td>
</tr>
<tr>
<td>l1f401.351330.0</td>
<td>user-240</td>
<td>1212359661</td>
<td>1212359662</td>
<td>1212359679</td>
<td>16</td>
<td>272</td>
<td>2130</td>
</tr>
<tr>
<td>l1f401.351332.0</td>
<td>user-240</td>
<td>1212360098</td>
<td>1212360100</td>
<td>1212360136</td>
<td>16</td>
<td>272</td>
<td>2130</td>
</tr>
<tr>
<td>l1f401.351339.0</td>
<td>user-1316</td>
<td>1212362548</td>
<td>1212362565</td>
<td>1212362575</td>
<td>128</td>
<td>1280</td>
<td>7200</td>
</tr>
<tr>
<td>l1f401.351342.0</td>
<td>user-1316</td>
<td>1212363058</td>
<td>1212363065</td>
<td>1212363076</td>
<td>128</td>
<td>1408</td>
<td>43200</td>
</tr>
<tr>
<td>l1f401.351361.0</td>
<td>user-1316</td>
<td>1212367335</td>
<td>1212367336</td>
<td>1212367344</td>
<td>16</td>
<td>128</td>
<td>3600</td>
</tr>
<tr>
<td>l1f401.351362.0</td>
<td>user-1316</td>
<td>1212367616</td>
<td>1212367622</td>
<td>1212367630</td>
<td>16</td>
<td>128</td>
<td>3600</td>
</tr>
<tr>
<td>l1f401.351363.0</td>
<td>user-1316</td>
<td>1212367910</td>
<td>1212367920</td>
<td>1212367927</td>
<td>16</td>
<td>112</td>
<td>3600</td>
</tr>
<tr>
<td>l1f401.351373.0</td>
<td>user-1316</td>
<td>1212371065</td>
<td>1212371067</td>
<td>1212371624</td>
<td>16</td>
<td>8912</td>
<td>7200</td>
</tr>
<tr>
<td>l1f401.351296.0</td>
<td>user-1416</td>
<td>1212345760</td>
<td>1212345762</td>
<td>1212383722</td>
<td>32</td>
<td>1214720</td>
<td>43200</td>
</tr>
<tr>
<td>l1f402.352225.0</td>
<td>user-1357</td>
<td>1212240623</td>
<td>1212240636</td>
<td>1212317089</td>
<td>16</td>
<td>1223248</td>
<td>111540</td>
</tr>
<tr>
<td>l1f402.352371.0</td>
<td>user-1297</td>
<td>1212243805</td>
<td>1212243811</td>
<td>1212330219</td>
<td>32</td>
<td>2765056</td>
<td>86400</td>
</tr>
<tr>
<td>l1f402.352387.0</td>
<td>user-240</td>
<td>1212332008</td>
<td>1212332021</td>
<td>1212332037</td>
<td>16</td>
<td>256</td>
<td>2130</td>
</tr>
<tr>
<td>l1f402.352399.0</td>
<td>user-240</td>
<td>1212335284</td>
<td>1212335285</td>
<td>1212335302</td>
<td>16</td>
<td>272</td>
<td>2130</td>
</tr>
<tr>
<td>l1f402.352406.0</td>
<td>user-1316</td>
<td>1212338951</td>
<td>1212338963</td>
<td>1212338974</td>
<td>128</td>
<td>1408</td>
<td>7200</td>
</tr>
<tr>
<td>l1f402.352409.0</td>
<td>user-1316</td>
<td>1212340318</td>
<td>1212340320</td>
<td>1212340331</td>
<td>128</td>
<td>1408</td>
<td>7200</td>
</tr>
<tr>
<td>l1f402.352416.0</td>
<td>user-240</td>
<td>1212343684</td>
<td>1212343689</td>
<td>1212343731</td>
<td>16</td>
<td>672</td>
<td>2130</td>
</tr>
<tr>
<td>l1f402.352435.0</td>
<td>user-1416</td>
<td>1212346569</td>
<td>1212346579</td>
<td>1212346621</td>
<td>16</td>
<td>672</td>
<td>2130</td>
</tr>
<tr>
<td>l1f402.352439.0</td>
<td>user-240</td>
<td>1212347037</td>
<td>1212347038</td>
<td>1212347080</td>
<td>16</td>
<td>672</td>
<td>2130</td>
</tr>
<tr>
<td>l1f402.352443.0</td>
<td>user-240</td>
<td>1212347922</td>
<td>1212347930</td>
<td>1212347950</td>
<td>16</td>
<td>672</td>
<td>2130</td>
</tr>
<tr>
<td>l1f402.352446.0</td>
<td>user-240</td>
<td>1212347754</td>
<td>1212347764</td>
<td>1212347806</td>
<td>16</td>
<td>672</td>
<td>2130</td>
</tr>
<tr>
<td>l1f402.352432.0</td>
<td>user-19</td>
<td>1212350199</td>
<td>1212350201</td>
<td>1212350640</td>
<td>128</td>
<td>56192</td>
<td>3600</td>
</tr>
<tr>
<td>l1f402.352433.0</td>
<td>user-19</td>
<td>1212351349</td>
<td>1212351359</td>
<td>1212351674</td>
<td>128</td>
<td>40320</td>
<td>3600</td>
</tr>
<tr>
<td>l1f402.352439.0</td>
<td>user-1412</td>
<td>1212356641</td>
<td>1212356691</td>
<td>1212356740</td>
<td>16</td>
<td>704</td>
<td>2130</td>
</tr>
<tr>
<td>l1f402.352444.0</td>
<td>user-240</td>
<td>1212356815</td>
<td>1212356828</td>
<td>1212356872</td>
<td>16</td>
<td>704</td>
<td>2130</td>
</tr>
<tr>
<td>l1f402.352446.0</td>
<td>user-240</td>
<td>1212357134</td>
<td>1212357137</td>
<td>1212357156</td>
<td>16</td>
<td>304</td>
<td>2130</td>
</tr>
<tr>
<td>l1f402.352447.0</td>
<td>user-240</td>
<td>1212357434</td>
<td>1212357439</td>
<td>1212357456</td>
<td>16</td>
<td>272</td>
<td>2130</td>
</tr>
<tr>
<td>l1f402.352448.0</td>
<td>user-240</td>
<td>1212357872</td>
<td>1212357875</td>
<td>1212357917</td>
<td>16</td>
<td>672</td>
<td>2130</td>
</tr>
<tr>
<td>l1f402.352451.0</td>
<td>user-240</td>
<td>1212358560</td>
<td>1212358561</td>
<td>1212358603</td>
<td>16</td>
<td>672</td>
<td>2130</td>
</tr>
<tr>
<td>l1f402.352452.0</td>
<td>user-240</td>
<td>1212359137</td>
<td>1212359147</td>
<td>1212359189</td>
<td>16</td>
<td>672</td>
<td>2130</td>
</tr>
<tr>
<td>l1f402.352453.0</td>
<td>user-240</td>
<td>1212359296</td>
<td>1212359306</td>
<td>1212359348</td>
<td>16</td>
<td>672</td>
<td>2130</td>
</tr>
<tr>
<td>l1f402.352454.0</td>
<td>user-240</td>
<td>1212359436</td>
<td>1212359450</td>
<td>1212359467</td>
<td>16</td>
<td>272</td>
<td>2130</td>
</tr>
<tr>
<td>l1f402.352457.0</td>
<td>user-240</td>
<td>1212360289</td>
<td>1212360306</td>
<td>1212360343</td>
<td>16</td>
<td>592</td>
<td>2130</td>
</tr>
<tr>
<td>User</td>
<td>Start Time</td>
<td>End Time</td>
<td>Bytes</td>
<td>Duration</td>
<td>Size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>-----------</td>
<td>---------</td>
<td>----------</td>
<td>-------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1f402.352465.0</td>
<td>user-1316</td>
<td>1212362679</td>
<td>1212362689</td>
<td>1212362700</td>
<td>128</td>
<td>1408</td>
<td>7200</td>
</tr>
<tr>
<td>1f402.352467.0</td>
<td>user-1316</td>
<td>1212363127</td>
<td>1212363130</td>
<td>1212363141</td>
<td>128</td>
<td>1408</td>
<td>43200</td>
</tr>
<tr>
<td>1f402.352468.0</td>
<td>user-1316</td>
<td>1212363209</td>
<td>1212363226</td>
<td>1212363236</td>
<td>128</td>
<td>1280</td>
<td>43200</td>
</tr>
<tr>
<td>1f402.352481.0</td>
<td>user-1316</td>
<td>1212366299</td>
<td>1212366301</td>
<td>1212366318</td>
<td>32</td>
<td>544</td>
<td>7200</td>
</tr>
<tr>
<td>1f402.352486.0</td>
<td>user-1316</td>
<td>1212367389</td>
<td>1212367400</td>
<td>1212367408</td>
<td>16</td>
<td>128</td>
<td>3600</td>
</tr>
<tr>
<td>1f402.352394.0</td>
<td>user-1430</td>
<td>1212333439</td>
<td>1212333440</td>
<td>1212367710</td>
<td>128</td>
<td>4386560</td>
<td>43200</td>
</tr>
<tr>
<td>1f402.352487.0</td>
<td>user-1316</td>
<td>1212367743</td>
<td>1212367758</td>
<td>1212367764</td>
<td>16</td>
<td>96</td>
<td>3600</td>
</tr>
<tr>
<td>1f402.352495.0</td>
<td>user-1316</td>
<td>1212369699</td>
<td>1212369704</td>
<td>1212370275</td>
<td>32</td>
<td>18272</td>
<td>7200</td>
</tr>
<tr>
<td>1f402.352498.0</td>
<td>user-1316</td>
<td>1212371142</td>
<td>1212371150</td>
<td>1212371158</td>
<td>16</td>
<td>128</td>
<td>7200</td>
</tr>
<tr>
<td>1f402.352500.0</td>
<td>user-1316</td>
<td>1212371353</td>
<td>1212371369</td>
<td>1212371377</td>
<td>16</td>
<td>128</td>
<td>7200</td>
</tr>
<tr>
<td>1f402.352286.0</td>
<td>user-1253</td>
<td>1212267392</td>
<td>1212267397</td>
<td>1212386078</td>
<td>16</td>
<td>1898896</td>
<td>172800</td>
</tr>
</tbody>
</table>
Appendix C: Output Data

Here we can see the output data of the simulation, and is contained in output.dat.

<table>
<thead>
<tr>
<th>ID</th>
<th>Submit</th>
<th>Start</th>
<th>End</th>
<th>CPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>l1f401.350952.0</td>
<td>1212161035</td>
<td>121233843</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>l1f402.352225.0</td>
<td>1212240623</td>
<td>1212217488</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>l1f402.352230.0</td>
<td>1212243805</td>
<td>121230443</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>l1f402.352286.0</td>
<td>1212267392</td>
<td>121228616</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>l1f402.351170.0</td>
<td>1212269983</td>
<td>1212308021</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>l1f402.352350.0</td>
<td>1212307973</td>
<td>1212345816</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>l1f402.352359.0</td>
<td>1212312037</td>
<td>1212356356</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>l1f401.351170.0</td>
<td>1212269983</td>
<td>1212308021</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>l1f402.352350.0</td>
<td>1212307973</td>
<td>1212345816</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>l1f402.352359.0</td>
<td>1212312037</td>
<td>1212356356</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>l1f401.351240.0</td>
<td>1212315647</td>
<td>1212320782</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>l1f401.351245.0</td>
<td>1212319840</td>
<td>1212321045</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>l1f401.351246.0</td>
<td>1212320322</td>
<td>1212333012</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>l1f401.351250.0</td>
<td>1212329800</td>
<td>1212330053</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>l1f401.351256.0</td>
<td>1212328429</td>
<td>1212352603</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>l1f402.352386.0</td>
<td>1212331239</td>
<td>1212332053</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>l1f401.351262.0</td>
<td>1212331781</td>
<td>1212333053</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>l1f402.352371.0</td>
<td>1212320603</td>
<td>1212334769</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>l1f402.352387.0</td>
<td>1212332008</td>
<td>1212334051</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>l1f401.351263.0</td>
<td>1212332194</td>
<td>1212334051</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>l1f401.351265.0</td>
<td>1212332751</td>
<td>1212335052</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>l1f401.351272.0</td>
<td>1212334400</td>
<td>1212335052</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>l1f401.351270.0</td>
<td>1212333699</td>
<td>1212335053</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>l1f402.352399.0</td>
<td>1212335284</td>
<td>1212336052</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>l1f401.351276.0</td>
<td>1212335561</td>
<td>1212336052</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>l1f402.352416.0</td>
<td>1212343684</td>
<td>1212344077</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>l1f401.351292.0</td>
<td>1212344072</td>
<td>1212345077</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>l1f401.351216.0</td>
<td>1212296520</td>
<td>1212379913</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>l1f401.351296.0</td>
<td>1212345760</td>
<td>1212390995</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>l1f401.351297.0</td>
<td>1212346418</td>
<td>1212357077</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>l1f402.352422.0</td>
<td>1212346565</td>
<td>1212358077</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>ID: l1f401.351298.0</td>
<td>Submit: 1212346770</td>
<td>Start: 1212359035</td>
<td>End: 1212359077</td>
<td>CPUs: 16</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------</td>
<td>------------------</td>
<td>-----------------</td>
<td>--------</td>
</tr>
<tr>
<td>ID: l1f402.352423.0</td>
<td>Submit: 1212347037</td>
<td>Start: 1212360035</td>
<td>End: 1212360077</td>
<td>CPUs: 16</td>
</tr>
<tr>
<td>ID: l1f402.352424.0</td>
<td>Submit: 1212347292</td>
<td>Start: 1212361035</td>
<td>End: 1212361077</td>
<td>CPUs: 16</td>
</tr>
<tr>
<td>ID: l1f402.352426.0</td>
<td>Submit: 1212347754</td>
<td>Start: 1212362035</td>
<td>End: 1212362077</td>
<td>CPUs: 16</td>
</tr>
<tr>
<td>ID: l1f402.352443.0</td>
<td>Submit: 1212356691</td>
<td>Start: 1212363035</td>
<td>End: 1212363079</td>
<td>CPUs: 16</td>
</tr>
<tr>
<td>ID: l1f402.352444.0</td>
<td>Submit: 1212356815</td>
<td>Start: 1212364035</td>
<td>End: 1212364079</td>
<td>CPUs: 16</td>
</tr>
<tr>
<td>ID: l1f401.351321.0</td>
<td>Submit: 1212357034</td>
<td>Start: 1212365035</td>
<td>End: 1212365053</td>
<td>CPUs: 16</td>
</tr>
<tr>
<td>ID: l1f402.352446.0</td>
<td>Submit: 1212357134</td>
<td>Start: 1212366035</td>
<td>End: 1212366054</td>
<td>CPUs: 16</td>
</tr>
<tr>
<td>ID: l1f401.351322.0</td>
<td>Submit: 1212357267</td>
<td>Start: 1212367035</td>
<td>End: 1212367073</td>
<td>CPUs: 16</td>
</tr>
<tr>
<td>ID: l1f402.352447.0</td>
<td>Submit: 1212357434</td>
<td>Start: 1212368035</td>
<td>End: 1212368052</td>
<td>CPUs: 16</td>
</tr>
<tr>
<td>ID: l1f401.351323.0</td>
<td>Submit: 1212357830</td>
<td>Start: 1212369035</td>
<td>End: 1212369049</td>
<td>CPUs: 16</td>
</tr>
<tr>
<td>ID: l1f402.352448.0</td>
<td>Submit: 1212357872</td>
<td>Start: 1212370035</td>
<td>End: 1212370077</td>
<td>CPUs: 16</td>
</tr>
<tr>
<td>ID: l1f401.351324.0</td>
<td>Submit: 1212358159</td>
<td>Start: 1212371035</td>
<td>End: 1212371077</td>
<td>CPUs: 16</td>
</tr>
<tr>
<td>ID: l1f401.351326.0</td>
<td>Submit: 1212358392</td>
<td>Start: 1212372035</td>
<td>End: 1212372077</td>
<td>CPUs: 16</td>
</tr>
<tr>
<td>ID: l1f402.352451.0</td>
<td>Submit: 1212358560</td>
<td>Start: 1212373035</td>
<td>End: 1212373077</td>
<td>CPUs: 16</td>
</tr>
<tr>
<td>ID: l1f401.351327.0</td>
<td>Submit: 1212358910</td>
<td>Start: 1212374035</td>
<td>End: 1212374076</td>
<td>CPUs: 16</td>
</tr>
<tr>
<td>ID: l1f402.352452.0</td>
<td>Submit: 1212359137</td>
<td>Start: 1212375035</td>
<td>End: 1212375077</td>
<td>CPUs: 16</td>
</tr>
<tr>
<td>ID: l1f402.352453.0</td>
<td>Submit: 1212359296</td>
<td>Start: 1212376035</td>
<td>End: 1212376077</td>
<td>CPUs: 16</td>
</tr>
<tr>
<td>ID: l1f402.352454.0</td>
<td>Submit: 1212359438</td>
<td>Start: 1212377035</td>
<td>End: 1212377052</td>
<td>CPUs: 16</td>
</tr>
<tr>
<td>ID: l1f401.351330.0</td>
<td>Submit: 1212359661</td>
<td>Start: 1212378035</td>
<td>End: 1212378052</td>
<td>CPUs: 16</td>
</tr>
<tr>
<td>ID: l1f401.351332.0</td>
<td>Submit: 1212360098</td>
<td>Start: 1212379035</td>
<td>End: 1212379071</td>
<td>CPUs: 16</td>
</tr>
<tr>
<td>ID: l1f402.352457.0</td>
<td>Submit: 1212360289</td>
<td>Start: 1212380035</td>
<td>End: 1212380072</td>
<td>CPUs: 16</td>
</tr>
<tr>
<td>ID: l1f402.352394.0</td>
<td>Submit: 1212365945</td>
<td>Start: 1212380035</td>
<td>End: 1212381005</td>
<td>CPUs: 128</td>
</tr>
<tr>
<td>ID: l1f401.351361.0</td>
<td>Submit: 1212367335</td>
<td>Start: 1212381035</td>
<td>End: 1212381043</td>
<td>CPUs: 16</td>
</tr>
<tr>
<td>ID: l1f402.352486.0</td>
<td>Submit: 1212367389</td>
<td>Start: 1212382035</td>
<td>End: 1212382043</td>
<td>CPUs: 16</td>
</tr>
<tr>
<td>ID: l1f401.351362.0</td>
<td>Submit: 1212367616</td>
<td>Start: 1212383035</td>
<td>End: 1212383043</td>
<td>CPUs: 16</td>
</tr>
<tr>
<td>ID: l1f402.352487.0</td>
<td>Submit: 1212367743</td>
<td>Start: 1212384035</td>
<td>End: 1212384041</td>
<td>CPUs: 16</td>
</tr>
<tr>
<td>ID: l1f401.351363.0</td>
<td>Submit: 1212367910</td>
<td>Start: 1212385035</td>
<td>End: 1212385042</td>
<td>CPUs: 16</td>
</tr>
<tr>
<td>ID: l1f401.351373.0</td>
<td>Submit: 1212371065</td>
<td>Start: 1212386035</td>
<td>End: 1212386592</td>
<td>CPUs: 16</td>
</tr>
<tr>
<td>ID: l1f402.352481.0</td>
<td>Submit: 1212366299</td>
<td>Start: 1212387035</td>
<td>End: 1212387052</td>
<td>CPUs: 32</td>
</tr>
<tr>
<td>ID: l1f402.352482.0</td>
<td>Submit: 1212366422</td>
<td>Start: 1212388035</td>
<td>End: 1212390920</td>
<td>CPUs: 32</td>
</tr>
<tr>
<td>ID: l1f402.352498.0</td>
<td>Submit: 1212371142</td>
<td>Start: 1212391035</td>
<td>End: 1212391043</td>
<td>CPUs: 16</td>
</tr>
<tr>
<td>ID: l1f402.352495.0</td>
<td>Submit: 1212369699</td>
<td>Start: 1212391035</td>
<td>End: 1212391606</td>
<td>CPUs: 32</td>
</tr>
<tr>
<td>ID: l1f401.351281.0</td>
<td>Submit: 1212338850</td>
<td>Start: 1212415035</td>
<td>End: 1212415046</td>
<td>CPUs: 128</td>
</tr>
<tr>
<td>ID: l1f402.352406.0</td>
<td>Submit: 1212338951</td>
<td>Start: 1212416035</td>
<td>End: 1212416046</td>
<td>CPUs: 128</td>
</tr>
<tr>
<td>ID: l1f402.352409.0</td>
<td>Submit: 1212340318</td>
<td>Start: 1212417035</td>
<td>End: 1212417046</td>
<td>CPUs: 128</td>
</tr>
<tr>
<td>ID: l1f401.351307.0</td>
<td>Submit: 1212350028</td>
<td>Start: 1212418035</td>
<td>End: 1212418167</td>
<td>CPUs: 128</td>
</tr>
<tr>
<td>ID: l1f402.352432.0</td>
<td>Submit: 1212350199</td>
<td>Start: 1212419035</td>
<td>End: 1212419474</td>
<td>CPUs: 128</td>
</tr>
<tr>
<td>ID: l1f402.352433.0</td>
<td>Submit: 1212351343</td>
<td>Start: 1212420035</td>
<td>End: 1212420350</td>
<td>CPUs: 128</td>
</tr>
<tr>
<td>ID: l1f401.351310.0</td>
<td>Submit: 1212351502</td>
<td>Start: 1212421035</td>
<td>End: 1212421668</td>
<td>CPUs: 128</td>
</tr>
<tr>
<td>ID: l1f401.351312.0</td>
<td>Submit: 1212352419</td>
<td>Start: 1212422035</td>
<td>End: 1212422346</td>
<td>CPUs: 128</td>
</tr>
<tr>
<td>ID: l1f401.351313.0</td>
<td>Submit: 1212352619</td>
<td>Start: 1212423035</td>
<td>End: 1212423324</td>
<td>CPUs: 128</td>
</tr>
<tr>
<td>ID: l1f401.351315.0</td>
<td>Submit: 1212355771</td>
<td>Start: 1212424035</td>
<td>End: 1212424502</td>
<td>CPUs: 128</td>
</tr>
<tr>
<td>ID: l1f401.351339.0</td>
<td>Submit: 1212362548</td>
<td>Start: 1212425035</td>
<td>End: 1212425045</td>
<td>CPUs: 128</td>
</tr>
<tr>
<td>ID: l1f402.351346.0</td>
<td>Submit: 1212362679</td>
<td>Start: 1212426035</td>
<td>End: 1212426046</td>
<td>CPUs: 128</td>
</tr>
<tr>
<td>ID: l1f401.351342.0</td>
<td>Submit: 1212363058</td>
<td>Start: 1212427035</td>
<td>End: 1212427046</td>
<td>CPUs: 128</td>
</tr>
<tr>
<td>ID: l1f402.352467.0</td>
<td>Submit: 1212363127</td>
<td>Start: 1212428035</td>
<td>End: 1212428046</td>
<td>CPUs: 128</td>
</tr>
<tr>
<td>ID: l1f402.352468.0</td>
<td>Submit: 1212363209</td>
<td>Start: 1212429035</td>
<td>End: 1212429045</td>
<td>CPUs: 128</td>
</tr>
</tbody>
</table>
Appendix D: HPCX Setup

The main HPCx system consists of 40 frames, each with 4 LPARs of 8 processors in them: that's 160 LPARs, 1280 processors.

The processors of the system are divided into a number of Regions as following:

<table>
<thead>
<tr>
<th>Region</th>
<th>LPARs</th>
<th>Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Production</td>
<td>128</td>
<td>1024</td>
</tr>
<tr>
<td>Development</td>
<td>24</td>
<td>192</td>
</tr>
<tr>
<td>Interactive Parallel</td>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td>Batch Serial</td>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td>Login</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>
References

[1]. S.Srinivasan, R. Kettimuthu, V. Subramani, P. Sadayappan,.Characterization of Backfilling Strategies for Parallel Job Scheduling, International Conference on Parallel Processing Workshops (ICPPW'02), 2002, Department of Computer and Information Science, The Ohio State University

[2]. The HPCx User Guide, UoE HPCx Ltd
   http://www.hpcx.ac.uk/support/introduction/index.html


[8]. IBM Corporation, LoadLeveler Using and Administering, Ver.3 Rel. 3

Online at: ibm.com/redbooks