Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Self adjusted auto provision system at resource level

Author: You Hu 2631052

1st daily supervisor:	Prof. Adam Belloum	UvA, Netherlands eScience Center
2nd supervisor:	Dr. Jason Maassen	Netherlands eScience Center
2nd reader:	Prof. Chrysa Papagianni	UvA

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

April 15, 2021

"History, as Hegel said, moves upward in a spiral of negations"

Abstract

Resource management is no doubt one of the key problems that all clusters have to face. The LOFAR telescopes observe the sky and archive the records. Though there are computation facilities to process the observation data, the quantity of data is far beyond their capability. Therefore, the data is fetched and processed through a multiple-step pipeline when it is needed. Each step may take a long time and consume a significant amount of computation power. Currently, we have horizontally scalable implementations in MPI and Spark The computation power is positively correlated to the numbers of involved computing units. However, both of them have an intrinsic drawback on resource utilizing. To promote the utilization of the resources, we propose an auto-provisioning distributed computing system. The auto-scaling mechanism enables the applications to dynamically fetch and release resources, and as the consequence, the resources of the cluster are used to the maximum extent. The results show that the nominal resource utilization of a cluster can be improved up to 99.9%. With idle resources being used, users may take less time to wait. In our busy cluster scenario test case, users take 10% less time on average to wait for the job to be finished at the submission of the jobs.

Acknowledgements

Finally, I have finished my masters thesis, and prepared myself to graduate from VU Amsterdam. The two-year study journey is a fantastic memory for me. I met a lot of new friends, teachers, and strangers. I think I am fortunate: got a scholarship one day before the flight took off, got an internship at 510 of the Netherlands Red Cross, being offered an exciting topic for master thesis, got an offer from Flow before the pandemic, and the most important, I met my girlfriend at Decathlon after my third bike was stolen. Here, I would like to thank my friend Robbie Luo, my supervisor Prof. Adam Belloum and Dr. Jason Maassen, my parents, and my girlfriend Minlan Cai. Thank you all for the support to the master project work and the accomplishment in this two-year study. In this special year, it is you who supports me to step forward.

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

1

Introduction

1.1 Context

The Low Frequency Array(LOFAR) telescope[1] consists of 51 stations across Europe, and a typical LOFAR observation has the size of 100 TB which can be reduced to 16 TB after frequency averaging(1). Collectively, there are over 5 PB of data to be stored each year(2). In the case that the data-collecting speed exceeds the processing capability, the data will be stored and archived first, and then processed at the request of the researchers on astronomy, physics, and computer science. The pipeline is divided into multiple steps, and in this thesis, we focus on the calibration which is available to reduce the noise of observation. The Netherlands eScience Center[2] has developed solutions for calibrating imaged observation collected by LOFAR. As one of the implementations for image calibration, SAGECaL is designed and implemented to calibrate the observation by a given sky map(3). Based on the sky map, SAGECaL is able to process data in parallel.

Using the given pre-processed observation data, sky model, and parameters for computation i.e. number of iterations, the input data can be processed in parallel, which is, however still a computationally intensive application. Currently, GPU-MPI[3], and Spark[4] versions of SAGECaL are developed by eScience Center to speed up the data processing, which all have achieved high acceleration compared with the naive version. The GPU provides great acceleration which expands the computation capability of single nodes. However, it only scales by adding more power (CPU, RAM, GPU) to an existing machine (vertical scaling) where the growth of computation capability can not catch up with the

[1] http://www.lofar.org/
[2] https://www.esciencecenter.nl/
[3] https://github.com/nlesc-dirac/sagecal
[4] https://github.com/nlesc-dirac/sagecal-on-spark

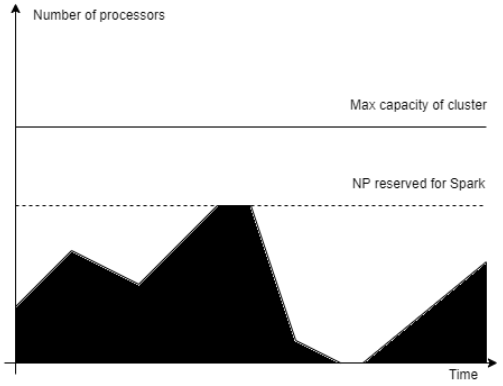

Figure 1.1: Resource utilization by Spark - Spark occupies fixed resources, but it would not release the idle resources

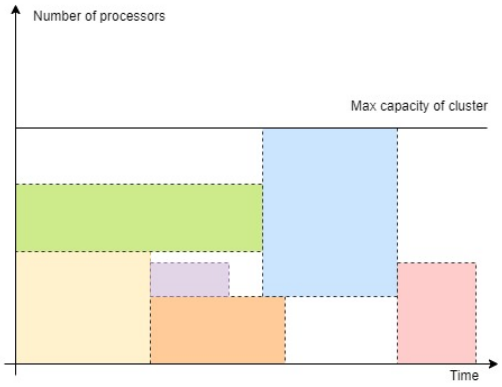

Figure 1.2: Resource utilization by MPI - Too large jobs make resource waste

growth of data. On the other hand, MPI and Spark implementations can scale by adding more machines to the resource pool (horizontal scaling), and the entire computational power increases via adding more resources. Of course, we can also enable GPU features to MPI and Spark implementations to make worker nodes more powerful. However, all these three solutions show their own limitations, in the following few paragraphs, we will outline these limitations.

LOFAR is one of the applications of ASTRON[1]and will likely run on ASTRON cluster which is designed to meet the demands of data processing applications. Therefore, the resource utilization of its infrastructures is also essential. The current implementations focus on the efficiency and utilization of assigned resources. In theory, MPI and Spark solutions may lead to a potential waste of resources of the whole cluster. Fig. 1.1 shows an example of computing resource waste of the Spark implementation when the required computation resources decrease, Spark does not release idle resources(compute nodes). On the other hand, a pure batch-job system may get easily in a situation that a big job is waiting for the resource while idle resources cannot fulfill the requirement, an example is shown in Fig. 1.2. The figure shows the possible resource waste represented by the blank area.

In this work, we will develop a dynamic resource management system that is able to adapt the resource used by an application to the backfill scheduling mechanism. To achieve this goal we will use Xenon[2] a middleware which aims to provide a uniformed and simple interface to enable application program to access both computation and storage resources.

[1] https://www.astron.nl/
[2] https://xenon-middleware.github.io/xenon/

Using Xenon interfaces, it is possible to submit jobs and access the status of jobs and the cluster without parsing the output of control commands from the resource manager. We will also use Ibis[1] a platform for distributed computing developed by the computer system group of VU Amsterdam. The Ibis Portability Layer, a sub-module of Ibis, enables computing entities to communicate with each other in a reliable cluster environment.

1.2 Objective

The main objective of this project is to achieve higher resource utilization of the cluster. At the same time, as a secondary objective, we aim to accelerate the calibration processing. In theory, higher resource utilization may lead to more active computation resources involved. To achieve these goals, we develop a system that reduces the waiting time of large distributed jobs.

1.3 Research Question

The overall research question is how to design and implement a distributed resource management system which can reduce the waste of resources.

1.4 Research Method

In this research, firstly, we briefly describe the commonly used resource management techniques in HPC clusters and analyse the MPI and Spark implementations to identify bottlenecks or limitations. Secondly, we will implement a resource management system that is able to scale up and down the computing resources dynamically according to the application workloads, so that the whole cluster can achieve overall higher resource utilization. Finally, we will test the performance of the system by comparing the resource utilization before and after the employment of this system. To validate our results, we will use the LOFAR calibration pipeline(SAGECaL). Calibration jobs independent and thus well-suited help validating our approach to application-centric resource management.

[1] https://www.cs.vu.nl/ibis/

2

Technical backgrounds

In this chapter, we firstly describe the detail for two implementations of LOFAR pipeline. And then, we briefly describe resource management techniques in HPC clusters, focusing on high resource utilization for cluster computing environment.

2.1 MPI and Spark implementation of SAGECal

MPI version

SAGECaL native supports multi-threads parallelization and GPU. The MPI version of SAGECaL[1] employs a master/worker architecture to manage the task distribution among nodes. The task division is based on data partitioning. Each worker node process a file at a time. The master tries to equally distribute tasks, but the workload can not be adjusted during the runtime. Besides, this MPI version does not support fault tolerance in case the worker nodes fail during the processing.

Spark version

To make better use of resources, eScience Center also developed a spark version of SAGECaL[2]. The SAGECaL is compiled as a dynamic library and utilized by Java native interface. The tasks are divided by file as well and managed by Spark. Compared to the MPI version, Spark provides better resource management and fault tolerance. Besides, with the help of container technology and container orchestras like Kubernetes, we can scale the running Spark environment manually.

[1] https://github.com/nlesc-dirac/sagecal/tree/master/src/MPI
[2] https://github.com/nlesc-dirac/sagecal-on-spark/tree/master/excon/JAVA

2.2 System dependency

2.2.1 SLURM

SLURM, formerly known as Simple Linux Utility for Resource Management, is a cluster management and job scheduling system for large and small Linux clusters which are open-source, fault-tolerant, and highly scalable. There are three critical functions, as it is stated, that is, allocating resources to users for a duration of time, providing a job management framework over-allocated node, and arbitrating contention for resources by a queue. SLURM can be configured with multiple kinds of queuing strategies, by default backfilling set up to maximize resource utilization in universal cases.

In our project, scaling relies on the submission and cancellation of jobs. To make a decision, the status of the cluster will also be periodically collected. The status includes the resource occupation and information of jobs in the queue. According to these statuses, the resource manager makes decisions to scale the calibration jobs. The SLURM receives instructions from the Resource manager of our system and allocates/retrieves resources by commands. And in the same time, it provides the status of the cluster to the Resource manager.

2.2.2 Xenon

Xenon[1], a middleware abstraction library, is utilized to manage the information and resources in an organized way, which enables our resource managers to communicate with the cluster in a more robust way, instead of parsing the output of command lines. It is designed for simple access to distributed computing and storage resources, which provides a single programming interface to many types of remote resources.

2.2.3 Shared file system

One of the fundamental requirements for our proposed system lies in the shared file system which allows us to achieve fault tolerance in a simple way. The shared file system can be accessed by all nodes, including head nodes and work nodes. It stores the container images of modules and processing environments for different kinds of jobs. The executors will read the raw data obtained from it, and generate a result which will be sent back.

[1] https://xenon-middleware.github.io/xenon/

2.3 Traditional resource management strategies

Batch scheduling has a long history covering the entire computer systems field from the mainframe age, up to today's computing systems. Batch scheduling is still the default scheduling method for modern computer systems. The simple FIFO batch scheduling systems turned to be quite inefficient and a number of optimization were proposed like preemption, backfill, and heuristics. In the following sections, we will explore these techniques in more details.

2.3.1 Preemption based resource management systems

Preemption is usually used to avoid job delays and resource starvation. Furthermore, apparently, resuming the execution of preempted jobs is the most time-consuming part. At the resources level, the preemption strategy is not common to be directly used on job scheduling along, instead, it is combined with other additional techniques. Sajjapongse et al. (4) proposed a run-time system based on a preemption strategy to increase GPU utilization on heterogeneous clusters. The paper describes the performance of hybrid MPI-CUDA applications showing the efficiency of preemption-based mechanisms. To overcome the drawbacks related to preemption, including the waste of resources, many adaptions are proposed. Lu Cheng et al. (5) proposed a solution inspired by MapReduce. They introduced a component Global Preemption to trade short-term fairness for better efficiency. Another approach is the checkpoint/restart mechanisms used by Berkely lab(6) in their Linux cluster. However, in the real environment, people use preemption strategies very carefully. Unless all jobs are equipped with a caching mechanism, otherwise, the cost of canceling running jobs will be unaffordable.

2.3.2 Backfill based resource management systems

The backfill algorithm is currently the default schedule algorithm to achieve high resource utilization in the production environment, which gives small jobs a higher priority. In Section 3.1, a backfill algorithm will be addressed in detail. Suresh et al. used a balanced spiral method for cloud metaschedules(7), which improves the performance and, at the same time, meets the requirement of QoS requirement of cloud systems. While Nayak et al. proposed a novel backfilling-based task scheduling algorithm to schedule deadline-based tasks(8). It aims to break the performance limit of the default backfilling algorithm of OpenNebula. This VM-based solution achieved a minor improvement in

resource utilization. Backfilling scheduling shows great generic and the ability for using resource utilization.

Several variations of the backfill technique have been proposed for different system configurations. EASY-backfill and conservative backfill hold the restriction not delay the job ahead(9). EASY-backfill is more aggressive, that is, for any job pending in the queue, backfill happens only when a small job does not delay the job at the head of the queue. However, in a conservative setting, a jobs backfill requires that the filling does not delay any job before it. Additionally, Flexible(10) and Multi-queue backfilling(11) are proposed to meet the requirements of more dynamic scenarios and reduce the response time for some jobs. Flexible tries to introduce slack factor to raise the priorities of big jobs in the queue. For multi-queue backfill, the partition will adapt as the workload change.

However, in terms of resource utilization, this algorithm still has some performance limitations, if there is no job that requires fewer processors than free processors, the free processors will remain idle. In (12), Hafshejani et al. turned to schedule jobs on the thread instead of on the processor. They tried to improve resource utilization via finer-granularity allocation. The results show that less response time on average is achieved compared with FCFS and traditional Backfilling.

2.4 Resource management strategies in research

2.4.1 Heuristics based resource management systems

Heuristics algorithms are usually more efficient, which takes less time to decide as the scheduling problem is NP-Hard. Xhafa and Abraham did a survey(13) and explored the application of heuristics algorithms in job scheduling. The most common and straightforward approach is local search, and the methods in this family include Hill Climbing (HC), Simulated Annealing (SA), and Tabu Search (TS), etc. In (14), local search facilitates the shortening of schedule on benchmark problems. Though the population-based approaches are more efficient, they require a longer time to convergence. In (15), the Genetic Algorithm approach allows the sufficient utilization of the resources. Moreover, of course, in this work, the above two approaches show that they can be combined to achieve better performance.

2.4.2 Machine learning based resource management systems

Machine/deep learning was greatly improved during the last few years, a couple of recent studies applied ML/DL approaches to resource management. Research made by Mao

et al.(16) shows that (deep)reinforcement learning is able to outperform the traditional state-of-art approaches. It translates the problem of packing tasks with multiple resources (herein referred to as CPU and memory) demands into a learning problem. Another similar study(17) also shows that the RL-based approach has great potential for resource management. However, the approach was only tested in simulation with synthetic load generated using well-known probability distribution like Bernoulli process, Uniform distribution, and Beta distribution.

ML/DL techniques have also been used to improve more traditional resource management algorithms. For instance, Gaussier et al.(18) used machine learning to improve backfilling. Backfill strategy relies on the estimated execution time which is normally assigned by users. Through predicting the execution time, better by ML model, backfill mechanism is available to make better decisions.

3

Backfilling and scaling policy

In this chapter, we first explain the backfill mechanism in more detail. Then we will describe a specialized scaling policy to maximize resource utilization .

3.1 Backfill mechanism

The backfill scheduling plug-in is loaded by default in the SLURM cluster. In the previous chapter, we have listed a few works related to backfill policy. Therefore, currently, we only consider the dynamic provision of CPU resources via SLURM job submission/canceling. By the setting of job submission and canceling, we designed the scaling policy of the resource manager to adapt to the backfill mechanism.

As an optimization for the basic priority queue, the backfilling scheduling starts the lower priority jobs provided it does not delay the expected start time of any higher priority jobs. In other words, the backfilling mechanism under discussion refers to the conservative backfill because standard configurations of clusters using in research aim to achieve a fair share of the computing resource among the users. An intuitive interpretation is shown in Fig. 3.1 and Fig. 3.2.

According to the configuration of SLURM, the backfilling scheduling is triggered when jobs are submitted/finished. Besides, the scheduler periodically checks whether a job in the queue is available to run. The decisions for backfilling depend on the number of resources, and the time limits of the jobs.

As shown in Fig. 3.1, Job 4 is pending in the queue due to the lack of resources. When Job 5 is appended to the queue, the scheduler estimates that Job 5 can be finished before Job 4 gets adequate resources. Therefore, the resource is allocated to Job 5.

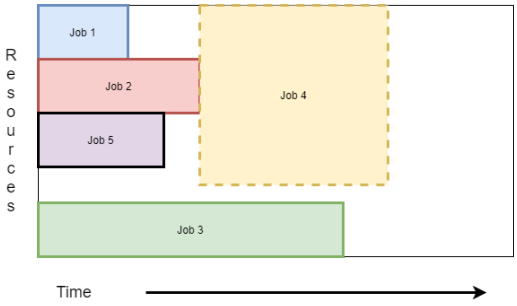

Figure 3.1: Job 5 is backfilled - Job 4 is estimated to start after the finishing of Job 2

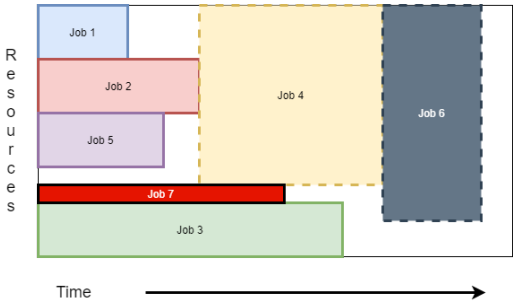

Figure 3.2: Job 7 is backfilled - Running Job 7 will not impact Job 4 and Job 6

Another scenario is shown in Fig. 3.2. Job 7 is added to the queue while it requires minimal resources, which will last when Job 4 is on the run by the estimation. Besides, it still gets the assigned resource as it does not affect the jobs ahead of it(by the estimation of starting time and priority).

The typical cases above show how the backfill mechanism works. In practice, the scheduler considers pending jobs in priority orders, that is, once a pending job fulfills the requirements of the backfill condition, it can start immediately. The resource manager of our system employs an adaptive algorithm that utilizes the backfilling mechanism to achieve high resource utilization.

3.2 An approach to maximize resource utilization

In the previous section, we discussed how backfilling mechanism enlarges resource utilization. In the default setting, users may submit any kind of job. However, in some cases like LOFAR use case, the users may execute the same program for different datasets. Therefore, we propose a resource management system that reorganizes one or more kinds of job execution and manages resources in a dynamic way. The system can retrieve and release resources according to scaling policy so that the overall resource can be fully used.

3.3 Scaling policy

The scaling policy aims to harvest every idle resource, which requires continuous monitoring of the status of the cluster and the running or pending jobs. The resource manager

periodically fetches status information, and makes decisions based on a scaling algorithm which acts according to the following figures:

- I - The number of idle nodes in a (partition of) cluster
- T - The total number of nodes in a (partition of) cluster
- R - The number of nodes reserved for our system
- J_i - The pending or running job with ID i
- N_i - The number of required nodes of J_i
- TL_i - The time limit of J_i
- RT_i - The running time of J_i
- $MiniNode$ - The minimum number of nodes reserved for our system

In the following, we demonstrate three cases that explain what will happen under the given conditions. Note that we describe background jobs as *Jobi* which are submitted by other users. At the same time, we will name The resources reserved for our system as *Calibration* as we will use the LOFAR calibration pipeline as the test use case.

3.3.1 Case 1: RM harvest idle resources

First, considering that sometimes there is no job pending in the queue, there are I nodes remaining idle. To increase the overall resource utilization, the resource manager will submit I one-node jobs, thereby sharing the calibration application workload, which is the basic strategy for any auto-scaling system. The distributed jobs will be accelerated, benefiting from more resources allocated.

3.3.2 Case 2: RM give free resources

To be friendly with other users, the system release resources when it gets sufficient resources($R >= MiniNode$), and other jobs are pending. In the case that the extra part of resources exceeds the requirement of the first pending job, resources will be released in our case from the set of resources needed for the LOFAR Calibration application. In other words, the system is trying to let as many jobs as possible run, provided that the giving out of resources will not slow down the calibration application($R < MiniNode$).
Example:

At a time, the resource manager collected the information from the cluster and jobs. Let $T = 21$,$MiniNode = 10$. And there two jobs J_1 and J_2 are running, where $N_1 = 5$, $N_2 = 5$, $TL_1 = 20min$ and $TL_2 = 15min$. Assume that both RT_1 and RT_2 are equal to $1min$. And there are two pending jobs J_3 and J_4 with $N_3 = 10$, $N_4 = 6$, $TL_3 = 25min$ and $TL_4 = 10min$. Now, the system has taken the rest 11 nodes in the cluster, which means $R = 11$. If J_2 is canceled somehow, then $I = 5$. It is easy to find that if the resource manager shares one more node, plus five idle nodes, J_6 can start according to the backfilling policy. After that, R is still not less than $MiniNode$. A graphic illustration is displayed in 3.3, where the dotted line represents the number of $MiniNode$. Please be

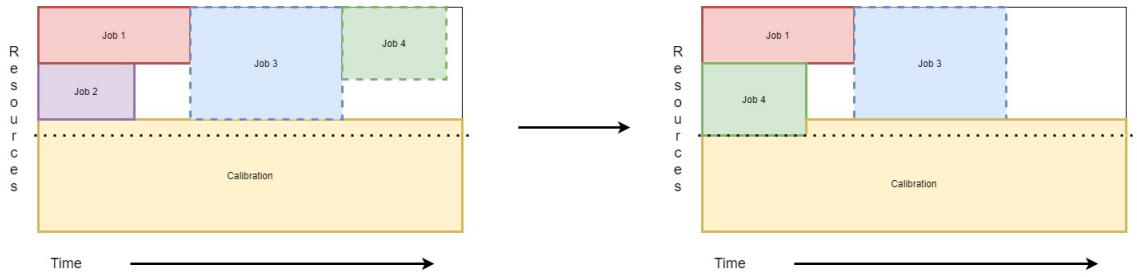

Figure 3.3: Scaling policy Case 2 - J_2 canceled, the system gives way, J_4 is backfilled

noted that if the job on top of the queue, herein refers to J_3, is able to start once getting sufficient resources, the resource manager will give way for it. And in the implementation, the job on the head of the queue will be considered first, which is followed by the jobs behind.

3.3.3 Case 3: RM does not free resources

The prerequisite of giving way for other jobs is that giving out resources would not break down the $MiniNode$, and the time limit is appropriate. In the case that there are no suitable jobs available to be backfilled, the resource manager takes those idle resources. It first calculates the maximum time necessary to ensure that a job can be backfilled. If no job can be backfilled, the resource manager submits I one-node jobs with $TL = maxTime - 2mins$. The reason to subtract 2 minutes is to ensure that the jobs can be backfilled correctly so that we make them redundant. The backfilling scheduling takes a long time, especially when there are many jobs on the cluster(running and pending).

Example:

The setting and jobs are the same as the previous example, however, we changed the time limit of J4 to 25 minutes. Thus, the resource manager submits jobs with $TL = 17mins$ to take 5 idle nodes. This scenario is illustrated in Fig. 3.4.

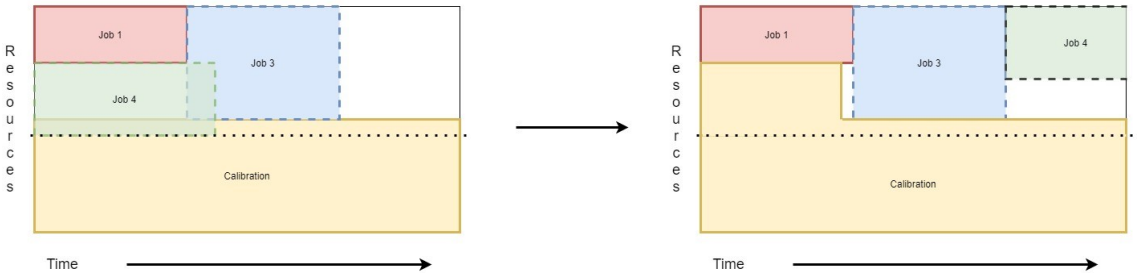

Figure 3.4: Scaling policy Case 3 - J_2 canceled, calibration application takes the idle resources because backfilling J_4 will delay J_3. Then calibration application takes them

4

Architecture and implementation

4.1 Overview design

Based on the review of the previous works and the scaling policy described in previous chapter, a system is proposed for the research questions: a user-side solution for overall resource utilization of batch job clusters. The system consists of two layers, i.e., the management layer and the computation layer. The overview design is illustrated in Fig.4.1. At the management layer, the resource layer is responsible for deciding resource allocation at runtime; therefore, the computation layer is enabled to scale on demand, which is responsible for parallel job execution. The computation layer is composed of multiple executors on arbitrary working nodes by the demand. All nodes can access the shared file system which is provided by the cluster. In the following sections, we first explain the functionality of each component and how these components interact with each other. And then, we explore the detailed implementations of this system.

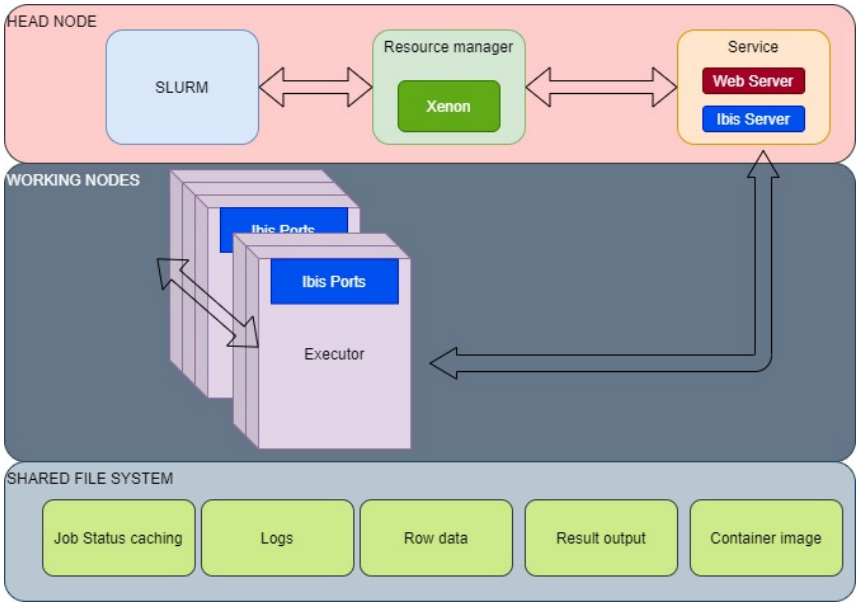

Figure 4.1: Three components are placed in two layers with a shared file system at button

4.2 Components

4.2.1 Resource manager

The Resource Manager(RM) is mainly responsible for deciding to change the number of resources allocated to this system. The decision-making is based on the information obtained from SLURM and WebService. The RM continuously queries the status of the available resources via the Xenon interface.

Besides, the RM also fetches information about the status of users jobs from Web-Service(Web server in Fig.4.1) through Restful API. These statistics help the resource manager to make decisions.

4.2.2 Service module

Consisted of two sub-components, the service module is a container instance hosting a web server and an Ibis server. In this project, we assume that the head node never crashes, and the processes are not terminated by external action.

The web server is based on Flask Restful framework. The end-users can submit jobs via Restful API, and the webserver temporally stores the configuration of those jobs. Besides, this webserver also allows the master of executors to update the recommended minimal working nodes. This number can be used for RM to make scaling decisions. Note that

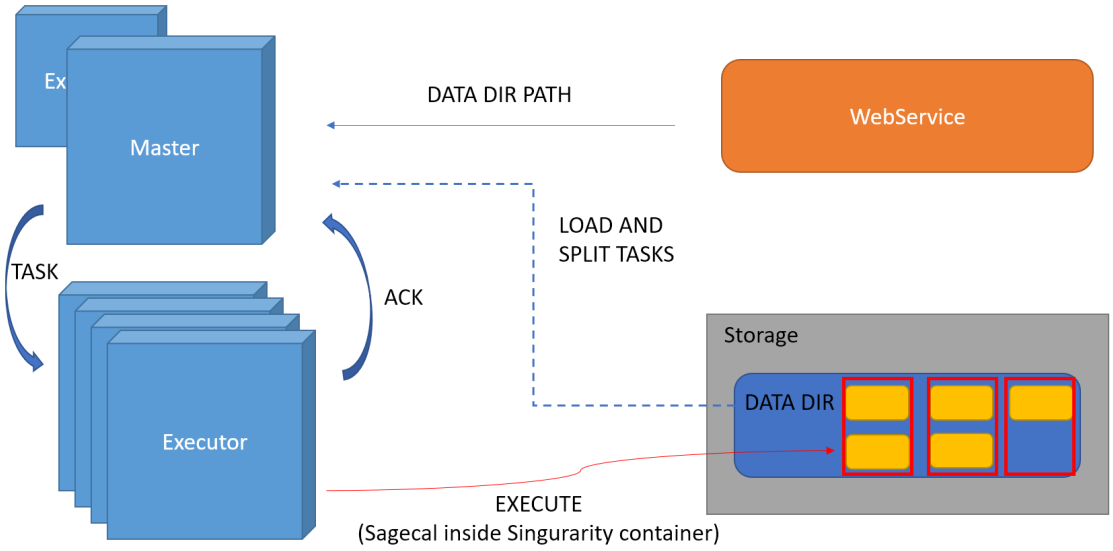

Figure 4.2: Master-worker architecture, red boxes indicate the batch size

the term job herein refers to application-related jobs. The other server is the Ibis server. In the computation layer, Ibis Portability Layer(IPL)(19) is employed for communication. The IPL requires an Ibis server as a centralized hub for managing the communication and events among Ibis instances. Considering the requirement for stabilization, we choose to run the Ibis server on the head node to mitigate the risk of crashing.

4.2.3 Executors

The executors are the main power for data processing. In this project, every time RM decides to scale up the computation ability of the system, it will submit a new pre-defined job to SLURM. Once this job is executed, a new executor is added to the pool of application executors.

By exploiting IPL interfaces, the computation layer is designed as shown in Fig. 4.2. Every executor creates an Ibis instance for communication, and all the instances, after initialization, will poll an election to select the master. As a result, one executor is acting as the master and the rest of the executors are tasked with processing the data. In our project, the executors process data based on containers, which enables our system to handle multiple types of jobs for different kinds of data set.

The master periodically fetches jobs from the WebService (a simple Flask Restful API service). The information returned by WebService includes data directory, user, job id,

and parameter list. In this system, the objective is to process a data set that can be divided into multiple sub-data sets. Therefore, as shown in Fig, 4.2, a job is represented by a data folder that consists of subfolders(as shown by the yellow blocks in the figure). The master reads the information of the data folder and creates a job object. *Job* object is defined as an abstraction of jobs submitted by end-users, which carries information, including the data directory, batch size, and parameters for processing. It also maintains a queue storing the tasks to be delivered to workers. The numbers of running and waiting tasks are recorded for task-redoing and job-finishing checks. In the case that a *Job* object is initialized, it lists the sub-directories under the directory where job data is stored. According to the given batch size, the Task objects are created and loaded to the queue. *Task* Task object stores the paths of sub-dataset, job id, and parameters. Moreover, executors send an acknowledgment to master every time they enter the idle state and wait for a new task. After that, the master delivers tasks to idle executors when there are unfinished tasks/jobs.

4.3 Implementation in detail

4.3.1 Actions of executors

Taking the advantages of Ibis, all executors run the same Java code, and take different actions according to the result of election. The first action of executors is to join the election for master. The Ibis service ensures that there is only one master in a pool. For both master and workers, the Upcall mechanism is utilized to receive incoming messages, which allows asynchronous message communication.

The master maintains three variables, i.e.

- (BTreeMap <Integer, Job>) $runningJobMap$;
- (Queue <IbisIdentifier>) $idleWorkerQueue$;
- (BTreeMap <IbisIdentifier, Task>) $runningNodes$.

$runningJobMap$ stores the jobs with job ID as the key. And $idleWorkerQueue$ is filled by IbisIdentifiers of executors, which send acknowledgment reporting that they are idle. When entering a master code block, the master firstly initializes an HTTPClient, the variables for statuses caching, a job fetcher, and the sending/receiving ports. After initialization, the master keep waits for notification from idle executors to assign the next task for execution.

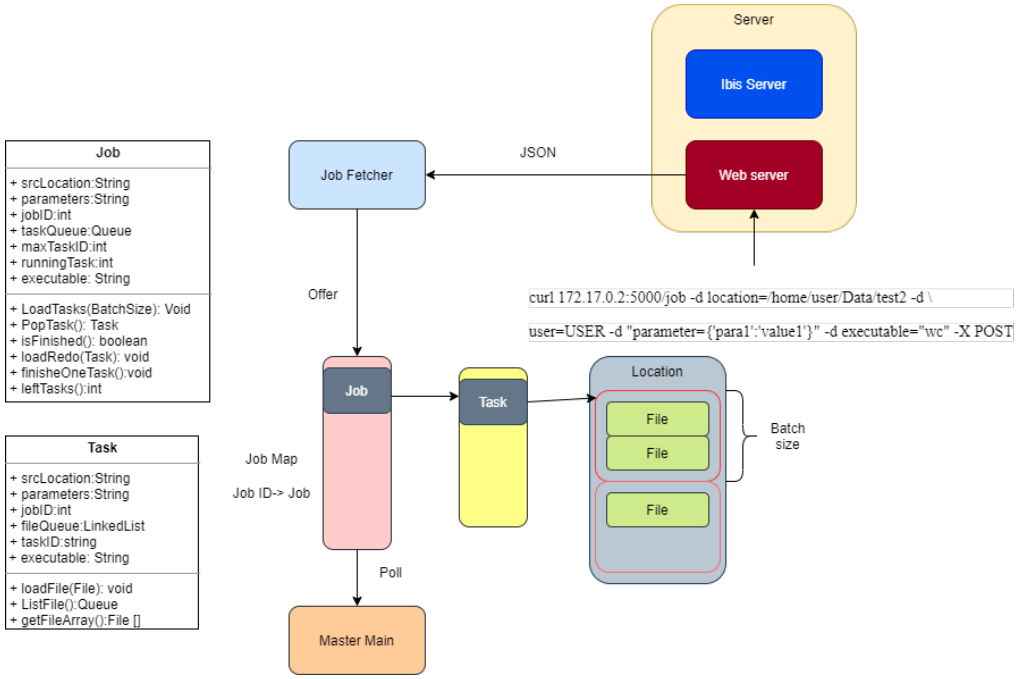

Figure 4.3: Users submit jobs to web service, Job fetcher parses JSON data pack and push *Job* objects to *runningJobMap*

The job fetcher runs asynchronously on another thread, communicating with the master main thread via managing *runningJobMap* and *runningNodes* which are accessible to two threads. It gets the jobs from web service, creates and initializes *Job* objects, and then pushes them to the *runningJobMap*. The submission process can be visualized as shown in Fig. 4.3.

The *runningJobMap* is locked when either main thread or job fetcher try to access and modify it. Besides, *runningJobMap* is a treemap, which is automatically sorted every time the elements inside are changed. Since the key is job ID, jobs are ordered by the job ID. In this manner, the order is kept, and jobs are processed in FIFO.

To assign tasks to executors, the master fetches a Task and an IbisIdentifier from *runningJobMap* and *idleWorkerQueue*. In general, If both Task and IbisIdentifier objects are not null, the Task object is sent to a node by the ID. Moreover, the Task and ID are stored in *runningNodes*. The detailed procedure is specified in Algo. 3. The actions when one of them is null are shown in this Pseudocode. Note that if the master instance only takes the role of management, the computation resource is wasted because of the fact that the management does not require too much computation. Therefore, in the initialization state, the master creates a process to launch a new instance aside. This

side executor will be a worker which also processing tasks.

At last, the master handles acknowledgment from workers. The upcall method is employed to process the incoming messages for both master and workers. In Algo. 4, the instances should take different actions to handle the incoming messages according to their roles. The master receives a control message which indicates whether this is the first time to join in the pool. Regardless of whether this is the first message of the worker, worker $IbisIdentifier$ is filled into the $idleWorkerQueue$, indicating that this executor is waiting for the task. However, if this worker is in $runningNodes$ while the control message shows it is a new worker, the task in $runningNodes$ will be fetched and redone. According to the job ID, the master updates the job status to indicate that a task is done. When all tasks of this job are accomplished, the job will be popped out from $runningJobMap$ and logged. In the previous section, we mentioned that the $MiniNode$ should be dynamically changed based on the workload. Therefore, a recommended $MiniNode$ is sent to the Web-Service at the end of each round by the master, which is defined at the computation layer and based on the resource manager scaling policy. Now, the $MiniNode$ of the resource manager is strictly the same as the recommended $MiniNode$ uploaded by the master.

The workers are simpler than the master, a worker first sends an acknowledgment to the master, and waits for a task to be processed. The main thread of the worker constantly tries to fetch the $Task$ object from (BlockingQueue<Task>) $workerTaskQueue$ and processes it. The $Task$ Task object contains paths to sub-datasets, and the worker processes sub-datasets referred to in task objects. Once all data are processed, the worker sends a control message to the master as acknowledgment. Workers also are enabled with Upcall function as it is shown in Algo. 4. The $Task$ objects are read and loaded to $workerTaskQueue$.

4.3.2 Decision flow of scaling policy

In the previous section, we described what the system should do in different scenarios. There is still a lot of detailed implementations to make it more robust to the environment. The pseudo-code showed in Algo. 1 describe the scaling policy in a formal way.

To ensure the stabilization of the system, in lines 14-16, the scaling is delayed because, within a particular time, some jobs will be finished. Besides, in practice, the scheduler will take much time(few seconds to 1 min) for backfilling. To avoid incorrect action, when a pending job has the possibility to be backfilled, the scaling procedure waits for the scheduler to handle it. This checking mechanism is specified in the lines from 20-22.

Note that, according to the limit of Xenon, the interface which is provided for querying the status of jobs does not contain the information about the starting time (for reservation

in advance), and preference on GPU nodes. Therefore, this system is not configured to deal with the GPU requirement, job array, and reservation in advance (start at a certain time).

4.3.3 Scaling up and down

Besides, when we should scale our system, it is also important to define how to scale via communicating with SLURM. In the implementation, scaling up relies on the submission of one-node SLURM jobs, and scaling down is done by canceling those jobs. In these actions, the jobs time-related features are significant; here is RT(real time) and TL(time limit).

A job that can be backfilled needs an appropriate time limit configuration. The $maxTime$, mentioned in Algo. 1, refers to the time duration to the estimated time when J_{Top} starts, that is, the maximum time limit within which those one-node jobs should be configured. A visual interpretation of $maxTime$ is displayed in Fig. 4.4.

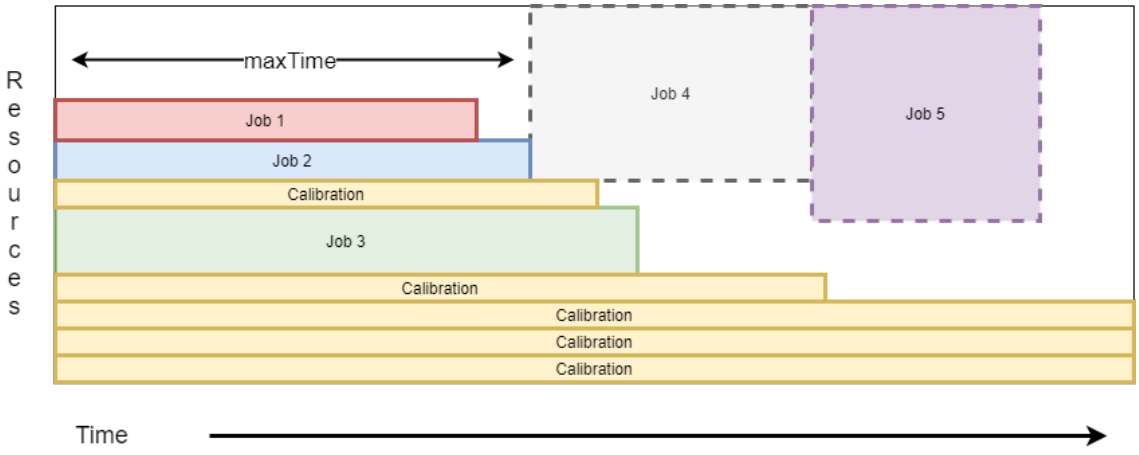

Figure 4.4: Max time for backfilling - J_4 is on top of the queue, according to the requirement for resources, it will start when J_2 finishes.

By applying Algo. 2, the resource manager can ensure the jobs expected to be backfilled start immediately.

Note that in SLURM, jobs can be configured with a $UNLIMITED$ time limit[1]. The sorting of $runningJobs$ is based on the left time of each job, which is calculated as $TL_i - RT_i$. Therefore, when a job has a $UNLIMITED$ time limit, this algorithm returns $UNLIMITED$.

[1] https://slurm.schedmd.com/scontrol.html

Besides of calculation of $maxTime$, time-out is added to scaling up, thereby preventing status changes on the cluster during the scaling. If other jobs take the resources, the time-out will be triggered as it cannot run at this particular time. However, in practice, sometimes submission still suffers from time-out even if there are idle resources, and the time limit is set correctly, due to the time taken by the backfill algorithm to complete its task. We introduce an initial time-out value of 5 seconds. Once a job submission triggers time-out, the scaling algorithm exits, and a time-out increment of 5 seconds is achieved. Then in the next round, the resource manager can still submit jobs to take idle resources while time-out is extended by 5 more seconds. In the case that the resources are taken by others, in the next round, it will not ask for scaling up again. Also, once a job is submitted successfully, the time-out is reset to 5 seconds again.

Scaling down requires sorting of $calibrationJobs$. Given a number N of resources to release, the top N jobs that are estimated to be almost close to finish will be canceled. In practice, the calibration jobs depend not only on the left time but also on the job id assigned by the SLURM because their left time is all infinite for jobs with a $UMLIMITED$ time limit. It is possible to cancel any jobs with the same left time randomly, and in principle, this will not cause any harm to the system. However, taking into consideration the master-worker of the structure computation layer, among the jobs with the same left time, jobs with larger job-ids will be killed first. This will make the old jobs last longer, and the node functioning as master will not be canceled frequently.

4.3.4 Fault tolerance

For the computation layer, it is very important to achieve fault tolerance. The dynamic scaling relies on the continuous creation and cancellation of jobs. Besides, the program does not have information about the time limit of SLURM jobs, therefore, when the time is out, those jobs will be terminated. In other words, the executors may be terminated by themselves without notice.

In the SLURM documentation, the job cancellation will result in the signals sent to the programs for cleaning up. According to the document[1], by default, a job will first send a SIGCONT to wake all steps up when it is canceled, which is followed by a SIGTERM sent to terminate programs. In the case that, after a duration, some steps are not terminated, a SIGKILL will be sent. This will also happen when the time limit is reached. Therefore, at the beginning of the setup of executor, and after the registration of the Ibis instance,

[1] https://slurm.schedmd.com/scancel.html

a signal handler is created to handle the SIGTERM and SIGKILL. Once a SIGTERM or SIGKILL is received, the Ibis instance is terminated, and the Ibis server is notified. Then, other instances will notice that the node was left or terminated, thereby taking actions according to their roles.

To handle the termination of instances better, the event handler provided by IPL is utilized. Once the Ibis server notices that an instance is left or was terminated, it will forward an event to all alive instances. An instance is able to handle the events which indicate the termination of other instances by the implementation of the member functions $died(IbisIdentifier\ corpse)$ and $left(IbisIdentifier\ leftIbis)$. Therefore, instances react to the events implicitly apart from the main logic.

For the master, in the case that a work node fails, it is important to ensure the running task of which this executor is in charge will not be lost. Here, the tasks should be processed at least once. As shown in Fig. 4.5, the master first fetches and removes the key-value

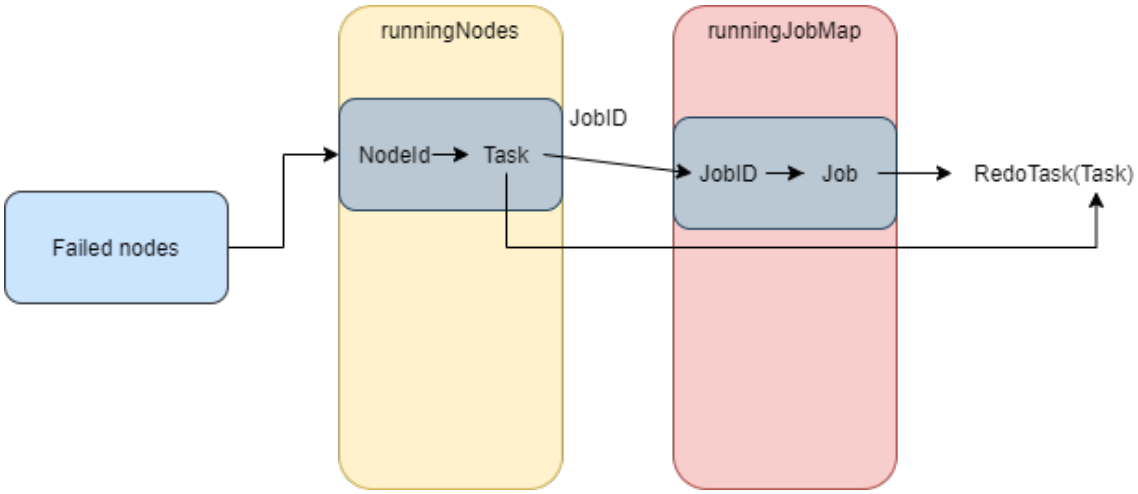

Figure 4.5: Master redoes task by failed node - Fetch task and load it back for redo

pair which belongs to *runningNodes*, and reloads the failed task. Thus, the tasks assigned to failed nodes can be re-computed, and the fault tolerance for workers is guaranteed.

The cases that the master fails are more complicated. On the one hand, the system requires a new master among executors, which may lead to a new round of the election. On the other hand, the new master should restore the jobs statuses, and continue with the unfinished jobs. To fulfill the requirements, the MapDB[1] is introduced into our system for lightweight persistence, which is an embedded Java database engine and collection

[1] https://jankotek.gitbooks.io/mapdb/content/

framework. The $runningJobMap$ and $runningNodes$ are constructed as BTreeMap; every time the change is committed, this update will be stored to off-heap storage. Therefore, a new master can read the caching file to restore these two variables, and then continue to process the unfinished jobs. Besides, to make the system simpler, every time a new master restores $runningNodes$, all running tasks will be reloaded to be processed again. This means, theoretically, the failed master leads to the redoing of all running tasks.

Another fault tolerance concept lies in the failure of the Ibis server. In this system, the Ibis server is assumed not to fail in any case. And in the future, it will be extended with fault tolerance at this level.

5

Experiments, results and analysis

In this chapter, we describe and analyze the results of a few experiments we have done to test the performance of the proposed system. The performance of this system includes

- The efficiency of scheduling algorithm - measured by resource utilization of the experiment cluster.
- The efficiency of parallel computation among executors - measured by drawing its speedup line and comparing it with the theoretical speedup.

In the following sections, first, we describe the test scenarios we have used to evaluate the performance of the proposed system. After that, we show how data processing jobs can be accelerated by adding more computation resources. In the last part, we test the overall performance of the proposed system by comparing the resource utilization of the cluster and the user waiting time before and after we introduced the proposed user-level resource scheduling. All experiments are performed on the DAS-5 Leiden site which contains 24 computation nodes, and each has dual 8-core CPUs with 2.4GHz speed and 64 GB memory.

5.1 SAGECal calibration use case

To test the proposed user-level resource management system, we use the calibration data pipeline of the LOFAR to process and calibrate the data collected by the LOFAR telescope. The latest radio astronomical calibration package in use is SAGECal (Space Alternating Generalized Expectation Maximization Calibration)[1]. It is fast and enabled with distribution and GPU acceleration features. In the test set, all worker nodes exploit SAGECal to process data in parallel by given configuration.

[1] http://sagecal.sourceforge.net/

To run SAGECal we need to provide a number of parameters, we discuss here only the one that is relevant to our tests, namely the number of threads. The number of threads is equal to the physical cores, it is based on the logic core number collected from the JVM runtime. However, as mentioned before in Section 4.3.1, there is a side executor alongside the master to maximize the resource utilization. To prevent the side executor from exhausting all the computation ability of the physical machine shared with the master executor, the number of threads for this worker executor is configured by subtracting 2 from the physical core number.

The SAGECal is well-encapsulated, and a typical use on a single node can be done by the command line below.

```
$ sagecal -d myData.MS -s mySkymodel -c myClustering -n no.of.threads \
          -t 60 -p mySolutions -e 3 -g 2 -l 10 -m 7 -w 1 -b 1
```

SAGECal is deployed as singularity container, and can be run as follows:

```
$ singularity exec Sagecal.simg /opt/sagecal/bin/sagecal PARAMETERS
```

The way worker executors exploit SAGECal to process data is to create a new process inside the Java program and use system call to launch data processing.

5.2 Distributed parallel computation

A test data set is created by duplicating a sample data set 150 times and each of them is a sub-data set. For each sub-data set, the size is 67 MB, and it takes about 20 seconds for computation.

We performed four experiments and recorded the execution time of each test in the function of the number of computing nodes. The results are shown in Table. 5.1 and the speed-up is visualized in Fig.5.1.

Num of Nodes	1	2	4	8
Time consumption(ms)	3,584,484	1,772,311	890,327	458,169
Acceleration	1	2.022	4.026	7.823
Theoretical speed-up	1	2.333	5.000	10.333
Efficiency rate	100%	86.7%	80.5%	75.7%

Table 5.1: Execution time by the different number of node

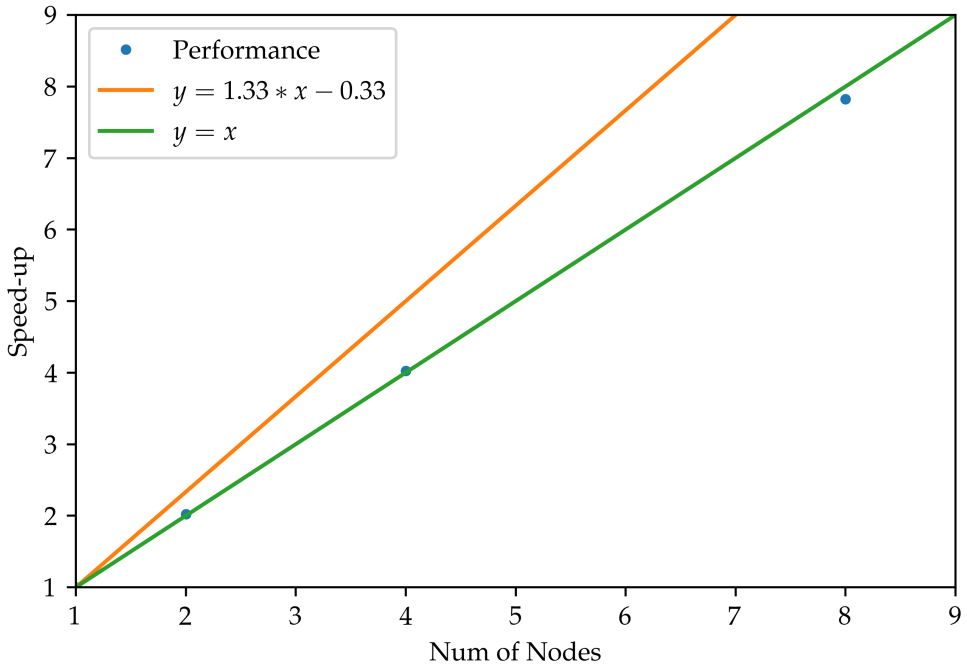

Figure 5.1: Performance of computation layer with different number of nodes - The ideal speed-up should follow $y = \frac{8}{6}x - \frac{1}{3}$ instead of $y = x$

In most cases, the curve of theoretical speed-up should be lower than that of linear speedup. However, as it can be seen from Fig. 5.1, the speed-up is super-linear. This super-linear speed-up originates from the inequality of computation speed between the side executor and the following additional worker executors. When there is only one physical node assigned to the system, it only uses full minus 2 cores to process data. In an 8-core cluster, every new involved physical node contributes $8/6 \approx 1.33$ times of computation power to the system in theory. Therefore, an adjusted theoretical linear speed-up should follow the line $y = \frac{8}{6}x - \frac{1}{3}$.

After the adjustment of the theoretical linear speed-up, the problem of inefficiency is revealed. It can be seen from Table. 5.1 that, by calculating the ratio between the real acceleration and theoretical speed-up, the efficiency keeps dropping with the increase of the resource introduced. According to the detailed performance metrics, the performance loss comes from connection build-up. In the implementation, the connection between ports is configured as an exclusive one-to-one connection. This means that every time the master sends a task to an idle worker, it has to disconnect to the previous workers port and connect to a new worker. This overhead cost is around one second per connection. In the experiment setting, with 150 tasks, connection takes 150 seconds for constant connecting, sending, and disconnecting. Of course, For long executions (hours) this overhead can be negligible Another optimization is to increase the batch size, which reduces the number of tasks and connections.

Besides the computation performance, we also tested the fault tolerance feature. The crash of either workers or the master will not result in the crash of the entire system. The jobs will be finished eventually as long as not all the resources are released. The dynamic scaling of the system depends on the fault tolerance mechanism. The completion of jobs in a dynamic scaling setting can support the conclusion that the fault tolerance feature is enabled.

5.3 Resource utilization optimization

For the user of this system, resource utilization is the key metric. In this section, the overall performance of the system is tested. We observe and consider two key metrics for performance.

- Nominal utilization: A/T; for cluster, it measures the resource utilization.

- User waiting time: $FinishTime - SubmitTime$; for calibration users, it represents the time they should wait.

The nominal utilization is called *Nominal* because it does not reflect the real utilization. For example, when there is no calibration job running, the system still tries to fully use the resource of the system. Besides the nominal utilization, the average resource usages of each kind of job are also monitored. One of the key features of the proposed use level resource management is not to be greedy and lead to being harmful to other users jobs, which means the average resource usage of other jobs should not drop too much.

5.3.1 Simulation settings

To test the systems performance, and measure its efficiency, a program to simulate various loads on the cluster has been implemented. We wrote an application that simulates multi-user jobs submission.

The application submits jobs including SAGECal jobs at various times to simulate the different load of the clusters. Each job submission specifies the time limit, real running time, submission time (from the start of the simulation), and other job-specific parameters. The application has two modes: SLURM-only mode where all jobs are considered as a batch job submitted by SLURM interface; and the scale mode where the resource manager is on, and the calibration jobs will be submitted to the web service instead of SLURM. The application will end 30 minutes after the last submission of the job.

The calibration jobs in SLURM-only mode are not real processing, instead, they are the same as normal jobs, i.e., executing the *sleep* command. With the configuration, the calibration jobs in SLURM-only mode will take five nodes for 240 minutes, which is close to the real case where the test data is processed using five nodes.

Besides, there is a monitor process that records the status of the cluster. The monitor records the resource occupation of different jobs (e.g., calibration, normal, others, and pending). The simulation program logs the submission time of the calibration jobs, and the calibration application itself logs the finishing of calibration jobs. Thus, the waiting time can be calculated. Additionally, the simulation application also records two internal values: the $miniNodes$ and $leftTasks$, which is the number of pending and running tasks. Monitoring these two values helps us to validate whether the scaling policy acts as expected. The mapping between $miniNodes$ and $leftTasks$ can be formulized as Eq.

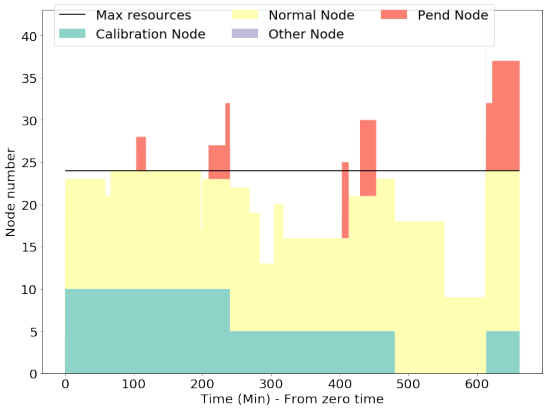

Figure 5.2: Resource utilization on SLURM-only mode ,non busy case - The overall resource utilization is 82.47%

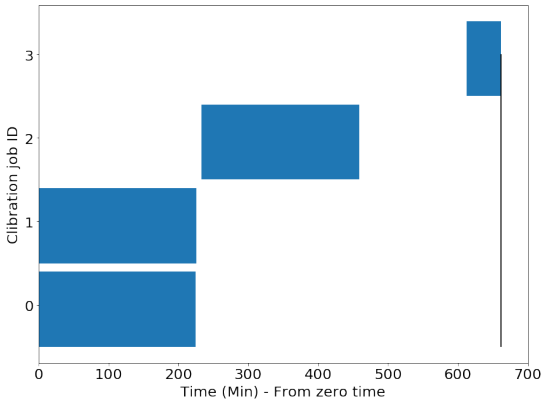

Figure 5.3: Gantt chart of calibration jobs - Benefit from adequate resources jobs start once they are submitted, the vertical line is the time simulation ends

(5.1).

$$miniNodes = \begin{cases} 2 & leftTasks < 5 \\ 4 & 5 < leftTasks < 20 \\ leftTasks/20 + 3 & 20 < leftTasks < 100 \\ 10 & else \end{cases} \tag{5.1}$$

The mapping function is also part of the performance parameters where we can configure the proposed system how greedy on resources for the calibration use case.

5.3.2 Scenario 1: Not heavily loaded cluster

First, we consider the scenario of the Non-busy cluster. In this case, most of the time, the cluster is not fully utilized due to a lack of jobs. The synthetic workload is described in Submit list 1, which consists of 19 jobs, including four calibration jobs and 15 other jobs. With this workload, the test takes about 11 hours, and during the test, in the SLURM-only mode, the resource usage is not optimal in Fig. 5.2 this idle resource is shown by the white area under the horizontal line. The cluster has 24 working modes, and during the experiments, the average resource occupation is 19.79(nodes), which means the overall resource utilization rate is 82.47%. The calibration jobs take 5.81 nodes on average, and other users take 13.99 nodes. According to Fig. 5.3 and Submit list 1, each job almost starts immediately after submitted due to the enough resources. In the second part of the test, we switch to scale mode. Fig. 5.4 comparing to Fig. 5.2, the white area under the

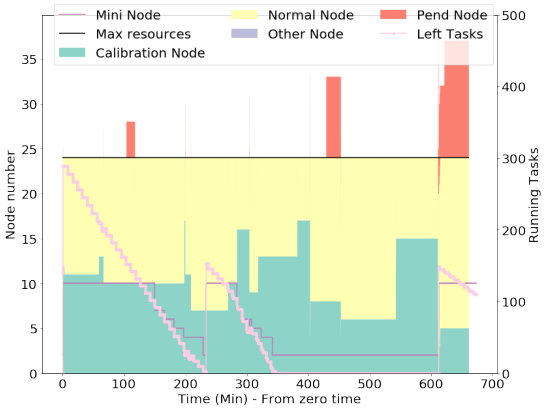

Figure 5.4: Resource utilization after introducing this system ,non busy case - The overall resource utilization is 99.83%

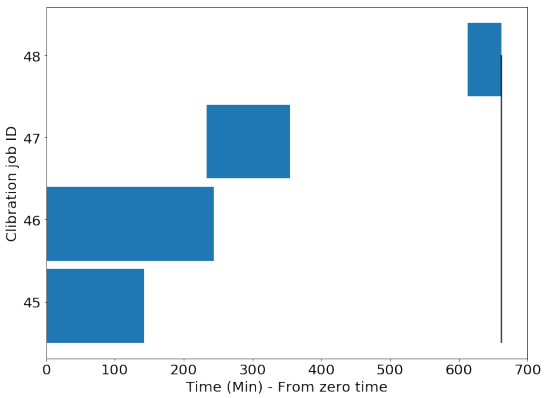

Figure 5.5: Gantt chart of calibration jobs - Job 47 is accelerated due to extra resources

horizontal line shows the idle resources has completely disappeared. The nominal overall resource utilization is 99.83%. The average resource occupation of normal jobs is 13.97 nodes, and for calibration applications, it increases to 9.99 nodes. In terms of waiting time, Fig. 5.5 shows that jobs cost less time to finish the same task. The waiting time is reduced from 240 minutes to 121 minutes.

Note that the waiting time reduction of Job 45 is not completely due to the auto-scaling. As within the proposed system, the execution order follows First come, first served(FCFS). Job 45 takes 10 nodes which is double the number in SLURM mode. Job 46 waits for Job 45 to finish instead of processing data at the same time.

This simulation shows in a non-busy scenario, the system can help to accelerate jobs for every user.

5.3.3 Scenario 2: Heavily loaded cluster

In a non-busy scenario, the overall resource utilization is not high. The proposed user-level resource management system initially aims to solve the problems of the limitations of the backfilling scheduling policy. Therefore, we create a heavy synthetic workload to study the resource utilization taking into account the backfill scheduling algorithm. There will be 24 jobs submitted to the cluster/system which includes 8 calibration jobs and 16 normal jobs in this busy scenario. The workload is listed in Submit list 2. It can be seen from Fig. 5.6, the utilization can be measured, and on average 90.57% of the resources are occupied.

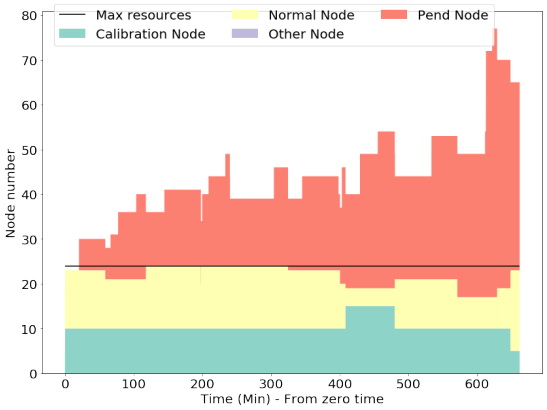

Figure 5.6: Resource utilization on SLURM-only mode, busy case - The overall resource utilization is 90.57%

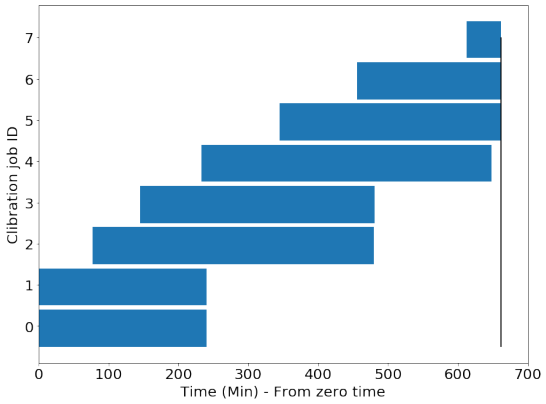

Figure 5.7: Gantt chart of calibration jobs - Jobs need to take a while waiting for resources

However, during the test, part of the resources is idle due to the limit of scheduling policy, even though there are plenty of jobs on the pending. Fig. 5.7 shows that that calibration jobs take a longer time to complete compared to scenario 1 (non-busy scenario).

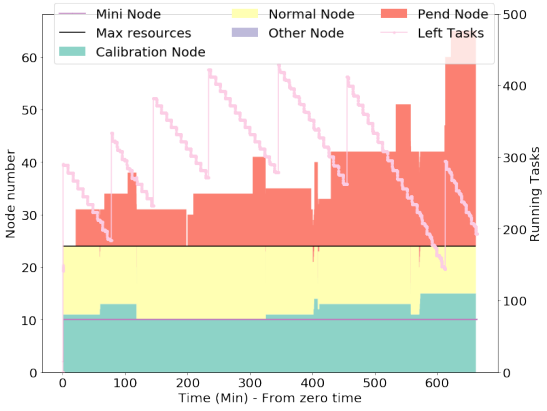

Figure 5.8: Resource utilization after introducing this system ,busy case - The overall resource utilization is 99.86%

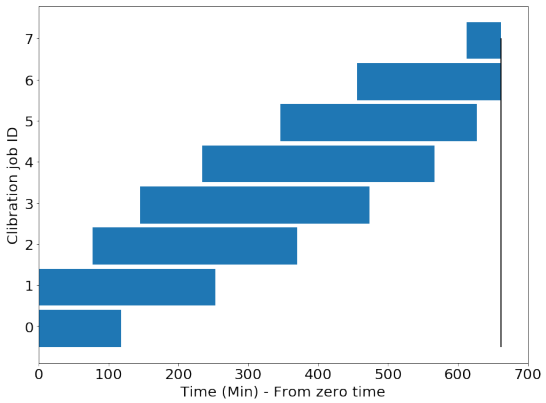

Figure 5.9: Gantt chart of calibration jobs - Users need less time for waiting

In the second part of the test we switched to scale mode, the overall nominal utilization climbs to 99.86%. In Fig. 5.8, the pink curve represents the number of pending and running tasks. During the entire simulation, the curve stays at a high position, which indicates there is no wasted resource taken by the system. Therefore, the nominal resource utilization is close to the real resource utilization of physical computation capability.

Job ID	0	1	2	3	4	5	6	7
SLURM-only mode(Min)	240.01	240.01	402.60	335.53	414.91	316.52	206.51	49.17
Scale mode(Min)	118.15	251.99	191.73	328.04	332.95	281.57	206.51	49.18

Table 5.2: Waiting time of jobs comparison on SLURM-only mode and Scale mode

The calibration applications, by comparing the waiting time of calibration jobs, the extra resources also benefit the calibration jobs. It can be observed from Table. 5.2 and Fig. 5.9 that, except for the jobs of 0 and 1, they are affected by the queue strategy and the jobs of 6 and 7, which are not finished at the end of the simulation. All other jobs, the jobs from 2-5, take less time after the introduction of this system.

Besides the scenario experiments, we also performed a test in the production environment hoping to measure the resource utilization under different workload patterns. The experiment was performed in the DAS5-Leiden site with full 23 available working nodes in seven days. But unfortunately, due to the lack of other users, the workload is lighter than our non-busy scenario. Therefore, we were not able to test the performance in different environments.

6

Discussions

In previous chapters, we purposed a user-level resource management system and performed experiments to evaluate how it impacts the overall cluster resource management. There are still some points not covered in the previous chapters, which can help readers to understand our system.

The system is driven by the idea to improve the calibration of the LOFAR use case. As mentioned, there are already two existed implementations for image calibration. The main reason for not following and extending the current MPI and Spark versions lies in the cluster setting. We can let our executors carry out the MPI program and process data. However, the MPI application expects much work to fulfill the requirements of fault tolerance features and dynamic settings. With IPL and Xenon, the workload and difficulty for development are reduced significantly. Moreover, it is possible to extend the current Spark version of the implementation. One possible solution on top of Spark implementation may be Kubernetes plus docker container so that it can support the dynamic scaling setting. Unfortunately, our test platform DAS-5, as a scientific cluster, does not support Docker containers, therefore, we cannot follow this path.

There are a few points in the core algorithm that we would also like to discuss. First, as mentioned in Chapter 3, this system does not support GPU features and the job array of the SLURM. The barriers originate from the middleware Xenon. For the GPU information, different clusters follow different ways to indicate which node carries the GPUs. Since Xenon intends to provide uniformed interfaces adapted to multiple resource management tools, it cannot provide the cluster-specified information. For the job array, Xenon has a problem in extracting the information of the job array. The job array can be configured with maximum running tasks, so there are a fixed number of tasks running at a specific time. Once a task is finished, a new task is submitted to the queue. Job array is a useful

feature that ensures a consistent workload and highly predictable. We are not fully capable to adapt to GPUs and job array until Xenon provides a solution to the issues.

The performance of the computation layer is affected by many parameters. For example, the batch size is the one that end-user can make choices, which determines the granularity of sub-tasks of calibration jobs. The longer a task to execute, the more performance lost when the node fails or gets released by the system. However, on the other hand, finer granularity leads to larger communication overhead. Considering the dynamic change of cluster, an appropriate batch size should be determined based on the environments characteristics. Another critical parameter is the minimum number of nodes. It is determined by the mapping function and the real-time workload of calibration jobs. An example is shown in Eq. (5.1). The mapping function can be modified to coordinate with other features and aspects.

7

Conclusion

In this report, we proposed a self-adjusted auto-provision system that aims to achieve as high resource utilization as possible through making adapting to the cluster backfill scheduling algorithm. The system tries to harvest all the idle resources, and be friendly to other users in the meantime. In general, with more resources putting in use, the cluster is available to process more jobs.

We choose the LOFAR image calibration process as our test use case, and the calibration process is implemented in a distributed form. The overall goal is to increase resource utilization and, at the same time, speed up the calibration jobs. After the adaption of the system, the results show that the calibration jobs get accelerated by more resources assigned, and the time which end-users have to wait is shortened. Besides, the resources allocated to other users jobs do not change a lot. In general, no user receives bad effects after introducing this system in our test cases.

For future work, the GPU and job array should be supported because they are very common to modern clusters. Since the GPU features are cluster-specified, we also need to provide an interface to collect the related information. Additionally, the core algorithm has room for improvement as well. For instance, to reduce the scheduling time of SLURM, we should enable this system to request resources with a larger size when needed.

The results of experiments and simulations show that the cluster can reach the nominal resource utilization of 99.8%, and users save waiting time ranging from 5%(busy workload) to 50%(idle workload) compared with the baseline(SLURM-only). In general, this system has achieved its objective to increase resource utilization and accelerate parallel jobs.

8

Appendix

8.1 Algoritms

Algorithm 1 Scaling policy

procedure SCALING
  if No pending job **then**
    Submit I one-node calibration jobs with $RT \leftarrow UMLIMITED$;
  else
    Get the pending job on top of the queue J_{Top}
    Get the $maxTime$ for backfilling
    if $R < MiniNode$ **then**
      Take I nodes, return
    end if
    if $N_{Top} - I <= R - MiniNode$ **then**
      Release $N_{Top} - I$ nodes, return;give a way to job waiting for resource
    end if
    for J_i in $runningJobs$ **do**
      if $TL_i - RT_i < miniTime$ **then**
        A job will finish soon, not change, return
      end if
    end for
    for J_i in $pendingJobs$ **do**
      if $TL_i < maxTime$ **then**
        if $N_i <= I$ **then**
          SLURM will schedule it, not change, return
        end if
        if $N_i - I <= R - MiniNode$ **then**
          Release $N_i - I$ nodes, return; give a way for J_i
        end if
      end if
    end for
  end if
  No job can be filled in; take I nodes, return
end procedure

Back to the content

Algorithm 2 Calculate $maxTime$

```
1:  procedure getMaxTime
2:      Get the job J_Top on top of the queue
3:      requiredNode ← I − N_Top
4:      MaxTime ← UNLIMITED
5:      for J_i in Sorted(runningJobs) do
6:          MaxTime ← (TL_I − RT_i)
7:          requiredNode ← requiredNode − N_i
8:          if requiredNode <= 0 then
9:              Return MaxTime
10:         end if
11:     end for
12: end procedure
```

Back to the content

Algorithm 3 Send tasks to worker

```
1: procedure DELIVERTASK
2:     Lock idleWorkerQueue and runningJobMap
3:     for Job_i in runningJobMap.valueSet() do
4:         if Job_i.isEmpty() then
5:             All tasks sent, continue
6:         end if
7:         task_j ← Job_i.PopTask()
8:         srcId ← idleWorkerQueue.poll()
9:         while srcId !=null do
10:            masterSendPort connect to worker by srcId
11:            Write message and send task_j
12:            if sending succeed then
13:                Break while loop
14:            else
15:                Fetch a new srcId
16:            end if
17:        end while
18:        if srcId == null then
19:            Job_i.loadRedo(task_j)
20:        end if
21:    end for
22: end procedure
```

Back to the content

Algorithm 4 Upcall Procedure

```
1: procedure UpCall(readMassage)
2:     if readMassage from worker then
3:         message ←(ControlMessage) readMessage.readObject()
4:         Lock idleWorkerQueue,runningJobMap and runningNodes
5:         runningTask ← runningNodes.remove(readMessage.origin().ibisIdentifier())
6:         if message.isEmptyRequest() == true then                ▷ First request
7:             if runningTask !=null then
8:                 Insert runningTask to the job in runningJobMap according to Job ID
9:             end if
10:        else
11:            job ← runningJobMap.get(message.getJobID())
12:            job.finishOneTask()
13:            if job.isFinished() then
14:                This job is finished; log results and remove it from runningJobMap
15:            end if
16:        end if
17:        idleWorkerQueue.offer(readMessage.origin().ibisIdentifier())
18:    else                                          ▷ worker gets task from the master
19:        task ← (Task) readMessage.readObject()
20:        Lock workerTaskQueue
21:        workerTaskQueue.offer(task)
22:    end if
23: end procedure
```

Back to the content

8.2 SubmitLists

8.2.1 Submit list 1

NodeNum=4 Type=Normal TimeLimit=1:57:57 RealTime=7077 SubmitTimeStamp=3405

NodeNum=2 Type=Normal TimeLimit=0:58:50 RealTime=3530 SubmitTimeStamp=6744

NodeNum=5 Type=Calibration TimeLimit=4:00:01 RealTime=14400 SubmitTimeStamp=13125 Parameter

NodeNum=7 Type=Normal TimeLimit=3:17:42 RealTime=11862 SubmitTimeStamp=21556

NodeNum=5 Type=Calibration TimeLimit=4:00:01 RealTime=14400 SubmitTimeStamp=27934 Parameter

NodeNum=3 Type=Normal TimeLimit=3:22:25 RealTime=12145 SubmitTimeStamp=3995500

NodeNum=4 Type=Normal TimeLimit=3:20:2 RealTime=12002 SubmitTimeStamp=6235156

NodeNum=6 Type=Normal TimeLimit=UNLIMITED RealTime=5017 SubmitTimeStamp=11996676

NodeNum=4 Type=Normal TimeLimit=UNLIMITED RealTime=10373 SubmitTimeStamp=12551230

NodeNum=5 Type=Calibration TimeLimit=4:00:01 RealTime=14400 SubmitTimeStamp=14016994 Parameter

NodeNum=7 Type=Normal TimeLimit=2:28:50 RealTime=8930 SubmitTimeStamp=18259084

NodeNum=9 Type=Normal TimeLimit=2:19:19 RealTime=8359 SubmitTimeStamp=24192376

NodeNum=9 Type=Normal TimeLimit=2:39:51 RealTime=9591 SubmitTimeStamp=25775085

NodeNum=9 Type=Normal TimeLimit=2:37:27 RealTime=9447 SubmitTimeStamp=36696690

NodeNum=5 Type=Normal TimeLimit=2:37:39 RealTime=9459 SubmitTimeStamp=36697980

NodeNum=5 Type=Calibration TimeLimit=4:00:01 RealTime=14400 SubmitTimeStamp=36767426 Parameter

NodeNum=5 Type=Normal TimeLimit=3:22:26 RealTime=12146 SubmitTimeStamp=36771775

NodeNum=8 Type=Normal TimeLimit=3:11:24 RealTime=11484 SubmitTimeStamp=36773060

NodeNum=5 Type=Normal TimeLimit=2:43:18 RealTime=9798 SubmitTimeStamp=37318439

8.2.2 Submit list 2

NodeNum=4 Type=Normal TimeLimit=1:57:57 RealTime=7077 SubmitTimeStamp=3405

NodeNum=2 Type=Normal TimeLimit=0:58:50 RealTime=3530 SubmitTimeStamp=6744

NodeNum=5 Type=Calibration TimeLimit=4:00:01 RealTime=13500 SubmitTimeStamp=13125 Parameter

NodeNum=7 Type=Normal TimeLimit=3:17:42 RealTime=11862 SubmitTimeStamp=21556

NodeNum=5 Type=Calibration TimeLimit=4:00:01 RealTime=13500 SubmitTimeStamp=27934 Parameter

NodeNum=7 Type=Normal TimeLimit=UNLIMITED RealTime=12412 SubmitTimeStamp=1241591

NodeNum=3 Type=Normal TimeLimit=3:22:25 RealTime=12145 SubmitTimeStamp=3995500

NodeNum=5 Type=Calibration TimeLimit=4:00:01 RealTime=13500 SubmitTimeStamp=4658139 Parameter

NodeNum=4 Type=Normal TimeLimit=3:20:2 RealTime=12002 SubmitTimeStamp=6235156

NodeNum=5 Type=Calibration TimeLimit=4:00:01 RealTime=13500 SubmitTimeStamp=8696543 Parameter

NodeNum=6 Type=Normal TimeLimit=UNLIMITED RealTime=5017 SubmitTimeStamp=11996676

NodeNum=4 Type=Normal TimeLimit=UNLIMITED RealTime=10373 SubmitTimeStamp=12551230

NodeNum=5 Type=Calibration TimeLimit=4:00:01 RealTime=13500 SubmitTimeStamp=14016994 Parameter

NodeNum=7 Type=Normal TimeLimit=2:28:50 RealTime=8930 SubmitTimeStamp=18259084

NodeNum=5 Type=Calibration TimeLimit=4:00:01 RealTime=13500 SubmitTimeStamp=20726878 Parameter

NodeNum=9 Type=Normal TimeLimit=2:19:19 RealTime=8359 SubmitTimeStamp=24192376

NodeNum=9 Type=Normal TimeLimit=2:39:51 RealTime=9591 SubmitTimeStamp=25775085

NodeNum=5 Type=Calibration TimeLimit=4:00:01 RealTime=13500 SubmitTimeStamp=27327961 Parameter

NodeNum=9 Type=Normal TimeLimit=2:37:27 RealTime=9447 SubmitTimeStamp=32015325

NodeNum=5 Type=Normal TimeLimit=2:37:39 RealTime=9459 SubmitTimeStamp=36697980

NodeNum=5 Type=Calibration TimeLimit=4:00:01 RealTime=13500 SubmitTimeStamp=36767426 Parameter

NodeNum=5 Type=Normal TimeLimit=3:22:26 RealTime=12146 SubmitTimeStamp=36771775

NodeNum=8 Type=Normal TimeLimit=3:11:24 RealTime=11484 SubmitTimeStamp=36773060

NodeNum=5 Type=Normal TimeLimit=2:43:18 RealTime=9798 SubmitTimeStamp=37318439

8.3 Terminology table

Term	Description
Horizontal scaling	Scale by adding more machines into your pool of resources.
Vertical scaling	Scale by adding more power (CPU, RAM, GPU) to an existing machine.

Table 8.1: Terminology table

References

[1] Hanno Spreeuw, Souley Madougou, Ronald Van Haren, Berend Weel, Adam Belloum, and Jason Maassen. **Unlocking the LOFAR LTA**. pages 467–470, 2019. 1

[2] **PROviding Computing solutions for ExaScale ChallengeS**. (777533):1–163, 2020. 1

[3] S. Kazemi, S. Yatawatta, S. Zaroubi, P. Lampropoulos, A. G. de Bruyn, L. V.E. Koopmans, and J. Noordam. **Radio interferometric calibration using the SAGE algorithm**. *Monthly Notices of the Royal Astronomical Society*, **414**(2):1656–1666, 2011. 1

[4] Kittisak Sajjapongse, Xiang Wang, and Michela Becchi. **A preemption-based runtime to efficiently schedule multi-process applications on heterogeneous clusters with gpus**. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 179–190, 2013. 7

[5] L. Cheng, Q. Zhang, and R. Boutaba. **Mitigating the negative impact of preemption on heterogeneous MapReduce workloads**. In *2011 7th International Conference on Network and Service Management*, pages 1–9, 2011. 7

[6] Paul H Hargrove and Jason C Duell. **Berkeley lab checkpoint/restart (blcr) for linux clusters**. In *Journal of Physics: Conference Series*, **46**, page 494, 2006. 7

[7] A. Suresh and P. Vijayakarthick. **Improving scheduling of backfill algorithms using balanced spiral method for cloud metascheduler**. In *2011 International Conference on Recent Trends in Information Technology (ICRTIT)*, pages 624–627, 2011. 7

REFERENCES

[8] Suvendu Chandan Nayak, Sasmita Parida, Chitaranjan Tripathy, and Prasant Kumar Pattnaik. **Dynamic Backfilling Algorithm to Increase Resource Utilization in Cloud Computing**. *International Journal of Information Technology and Web Engineering (IJITWE)*, **14**(1):1–26, 2019. 7

[9] J. Wang and W. Guo. **The Application of Backfilling in Cluster Systems**. In *2009 WRI International Conference on Communications and Mobile Computing*, **3**, pages 55–59, 2009. 8

[10] David Talby and Dror G Feitelson. **Supporting priorities and improving utilization of the IBM SP scheduler using slack-based backfilling**. In *Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. IPPS/SPDP 1999*, pages 513–517. IEEE, 1999. 8

[11] Barry G Lawson and Evgenia Smirni. **Multiple-queue backfilling scheduling with priorities and reservations for parallel systems**. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 72–87. Springer, 2002. 8

[12] Zeynab Moradpour Hafshejani, Seyedeh Leili Mirtaheri, Ehsan Mousavi Khaneghah, and Mohsen Sharifi. **An efficient method for improving backfill job scheduling algorithm in cluster computing systems**. *International Journal of Soft Computing and Software Engineering [JSCSE]*, pages 422–429, 2013. 8

[13] Fatos Xhafa and Ajith Abraham. **Computational models and heuristic methods for Grid scheduling problems**. *Future generation computer systems*, **26**(4):608–621, 2010. 8

[14] Graham Ritchie and John Levine. **A fast, effective local search for scheduling independent jobs in heterogeneous computing environments**. 2003. 8

[15] Ajith Abraham, Rajkumar Buyya, and Baikunth Nath. **Natures heuristics for scheduling jobs on computational grids**. In *The 8th IEEE international conference on advanced computing and communications (ADCOM 2000)*, pages 45–52, 2000. 8

[16] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. **Resource management with deep reinforcement learning**. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 50–56, 2016. 9

[17] Z. Liu, H. Zhang, B. Rao, and L. Wang. **A Reinforcement Learning Based Resource Management Approach for Time-critical Workloads in Distributed Computing Environment**. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 252–261, 2018. 9

[18] E. Gaussier, D. Glesser, V. Reis, and D. Trystram. **Improving backfilling by using machine learning to predict running times**. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–10, 2015. 9

[19] H. E. Bal, J. Maassen, R. V. van Nieuwpoort, N. Drost, R. Kemp, T. van Kessel, N. Palmer, G. Wrzesinska, T. Kielmann, K. van Reeuwijk, F. Seinstra, C. Jacobs, and K. Verstoep. **Real-World Distributed Computer with Ibis**. *Computer*, **43**(8):54–62, 2010. 19