MAXIMIZING PARALLELIZATION OPPORTUNITIES BY AUTOMATICALLY INFERRING OPTIMAL CONTAINER MEMORY FOR ASYMMETRICAL MAP TASKS

Shubhendra Shrimal

A Thesis

Submitted to the Graduate College of Bowling Green
State University in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE

August 2016

Committee:

Robert Dyer, Advisor

Robert C. Green II

Jong Kwan Lee

ABSTRACT

Robert Dyer, Advisor

The MapReduce programming model provides efficient parallel processing via two user-defined tasks: *map* and *reduce*. Input files are split into individual records and each record is processed by a single map task. Implementations of the model, such as Apache Hadoop, typically assume each record is similarly sized and thus they allocate the same memory for every map task. However, certain data sets, such as Wikipedia or the project data from SourceForge, do not follow this assumption and instead contain very asymmetrically sized records. To handle such data sets, the current state of the art solution is to determine the maximum memory requirements for the largest record and then configure the system to allocate this much memory for all map tasks. This simple solution however, decreases the maximum number of tasks running in parallel, leading to lower overall CPU utilization on the cluster. In this work, we propose three solutions to help maintain as much parallelism as possible while processing such asymmetric data sets.

As our first solution we propose a *fall-back approach*, where initially all map tasks run with the standard allocation of memory. If any task attempt fails due to memory reasons, subsequent attempts of that task are run with twice the memory/heap of the previous attempt. This solution is fully automatic and requires no configuration of Hadoop. Our second solution allows users to *manually specify* specific map tasks to be allocated more memory. Thus, if users know a priori which records are larger, they can avoid letting that map task run, fail, and retry with the first approach. Our third solution is to *automatically infer* which map tasks have higher memory requirements, based on learning from prior runs of the same input data which tasks failed and had to fall-back with our first solution. If multiple jobs are run on the same input, this solution avoids the need for users to manually specify the tasks requiring more memory. We evaluate these approaches by measuring performance on several data sets and comparing to the state of the art solution. Our evaluation shows up to 37–48% performance improvement for our approaches when compared to the state of the art solution, due to increased parallelism in the system, and with minimal overhead.

ACKNOWLEDGMENTS

First and foremost, I would like to thank Dr. Robert Dyer, my graduate adviser, for all his support and help from the very first day of my master's at Bowling Green State University. From timely meetings, suggestions, providing various ideas to work on, and taking very interesting courses like Big Data Analytics, Compilers and Algorithms at BGSU, I greatly appreciate all the effort he invested to help me in my thesis. He has been a great source of inspiration and motivation for me to work on my thesis.

I would also like to thank the Computer Science Department at Bowling Green State University and my thesis committee members, Dr. Robert Green and Dr. Jong Kwan Lee for providing very valuable suggestions and insights.

This research described in this thesis was supported in part by the US National Science Foundation (NSF) under grants CNS-1512947 and CCF-1518776, without which I could not have undertaken this research work.

A special thanks to all my friends and family for their support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1. INTRODUCTION

Apache Hadoop [1] is an open source *MapReduce* [2] framework for distributed processing of very large data sets and has seen tremendous adoption. MapReduce systems consist of two user-defined tasks: *map* and *reduce*. The input data set is split into *records* and each record is given to the map task for processing (in parallel). The outputs of the map tasks are then sent to the reducer to aggregate into the final result. Each map and reduce task in Hadoop is allocated memory and CPU resources. Hadoop typically assumes the input records are similarly sized and thus allocates every map task the same amount of resources.

However, not every data set contains records of similar size. Data sets such as Wikipedia's page dumps or the project data from SourceForge [3, 4] contains very asymmetrically sized records. For such data sets, Hadoop assumes all records are similarly sized and allocates equal resources to each map task. However, in the case of very large records there are chances of `OutOfMemoryException` or Java Heap errors leading to the entire job failing.

The current state-of-the-art solution for processing such data sets requires users to manually configure Hadoop map tasks such that the largest record among all the records can be processed successfully. To ensure this, users need to determine the maximum memory of the largest record and then configure the parameters accordingly. This results in allocating every map task the higher memory resources. Since every task now requires more resources, the number of map tasks that can run in parallel reduces, decreasing the overall parallelism and increasing the amount of time to process the data.

To better support processing of such asymmetric data sets, in this work we introduce three new approaches: *Fall-back, Manual* and *Automatic Inference* of large tasks. Our approaches allocate additional resources to only the map tasks processing larger records, rather than allocating higher resources to every map task. We modified the implementation of Hadoop to allow allocation of different resources to different map containers. By allocating higher memory to only certain map tasks, more maps will run in parallel as more memory is available for scheduling more map tasks.

First, we provide an automatic fault tolerance mechanism in Hadoop to deal with maps failing due to memory or heap issues when processing large records. Retried tasks are automatically given double the memory of their previous attempt. This approach is fully automated and requires no configuration of Hadoop. Second, we modified Hadoop to allow MapReduce to request map containers of different sizes. This approach allows users to manually specify the map tasks that require more memory (via a configuration parameter) so that the scheduler can assign them to larger containers. In our final approach, we propose an automated inference system that learns from previous jobs which tasks failed and required more memory. This system then uses the second approach to configure future runs to allocate more memory for those map tasks, thus avoiding the need to try them, let them fail, and then retry with more memory. This approach leaves the fall-back mode enabled, in case the system guesses wrong.

To evaluate these approaches, we used both real-world data sets and synthetic data sets. We measured the overhead of our approaches, which was constant and under one second. We also measured the performance improvement compared to the standard, unmodified Hadoop. Our approaches show significant improvement in processing time for both real-world and synthetic data sets. Our Fall-back approach is up to 37% faster on average for data sets having 10–30% large records without requiring any configuration changes in Hadoop, whereas our Manual approach shows up to 48% performance improvement for data sets having 10–50% large records. These performance improvements are the result of improved parallelism in the system, as more number of tasks are able to run in parallel.

The next chapter gives some background information on Hadoop MapReduce. Chapter 3 motivates the problem with an example. We discuss some related works in Chapter 4. Our approach is outlined in Chapter 5 and evaluated in Chapter 6. Chapter 7 discusses future work. Finally we conclude in Chapter 8.

CHAPTER 2. BACKGROUND

In this chapter we discuss key components of the Hadoop [1] MapReduce framework via a simple WordCount [5] application. We also discuss the YARN [6] architecture and how MapReduce applications run on top of it.

2.1 Overview of MapReduce

With enormously increasing data and the need to process data faster, there has been an extensive use of applications created using Hadoop MapReduce, here forward referred to just as Hadoop. Vast amounts of data are processed using these MapReduce applications on Hadoop clusters. A typical Hadoop cluster consists of several nodes over a well-connected network. A Hadoop cluster can be multi-node or single-node, usually comprising of commodity hardware and is based on the master/slave computation model. Hadoop performs distributed processing of data in map and reduce phases. In the map phase, input data is processed using several map tasks running in parallel. Every map task generates an intermediate output which is further processed using reducer tasks in the reduce phase. Each node in the cluster handles the map and reduce tasks allocated to them. After the reducer phase is completed, final results are generated.

2.1.1 The Map and Reduce Functions

Hadoop allows users to define map and reduce functions. Each record split of the input data set is processed according to the user defined map function. Compared to sequential data processing approaches, MapReduce framework can process data extremely fast. Every map task takes the <Key, Value> pairs as the input and generates an intermediate output consisting of <Key, Value> type. The number of maps running in parallel depends on total available memory across the cluster. The intermediate output is sorted and further processed to generate final results using the reducer function. The reducer task starts after completion of a minimum number of map tasks and this preemption of reducer tasks helps in achieving faster generation of the results. There can be more than one reducer task running in the system, but they again merge their outputs to generate final results.

2.1.2 Hadoop WordCount

One example of data processing using Hadoop is the WordCount problem. On starting the cluster, we start the ResourceManager, NodeManager, NameNodes and DataNodes. Once all are up and running, the Hadoop WordCount application is submitted. The submission of a job initiates the ApplicationMaster which is responsible for requesting resources to the ResourceManager. ResourceManager negotiates all the requested resources and allocates them in form of containers on various nodes of the cluster. A container consists of memory and vCores and can be allocated on any of the nodes based on the available resources on that node. Every task is processed inside a container and requires Java heap memory allocation. Figure 2.1 shows the processing of input data in different phases of Hadoop.

The input data is split into record splits and each record is processed by mapper function defined by the user. The Hadoop WordCount reads each line of the input data set as a record (the choice of input record reader and its format is application based). The mapper function processes these records by taking each word as key and assigns them a value according to the user defined logic. The intermediate output of the mapper consists of <Key,Value> pairs which are sorted and shuffled in the sort/shuffle phase for ultimate processing by the reducer.

The reducer task collects the sorted output of each map task, and aggregates the list of values corresponding to each key. The final output is stored in HDFS [7] with the count of each word. The ApplicationMaster makes a remote request call to ResourceManager for allocation of resources. ResourceManager allocates containers corresponding to these requests. Every map or reduce task is processed inside a container. Once the container for any task is allocated on the node by interaction of ResourceManager with NodeManager, the task is executed inside it and is marked as complete after successful processing.

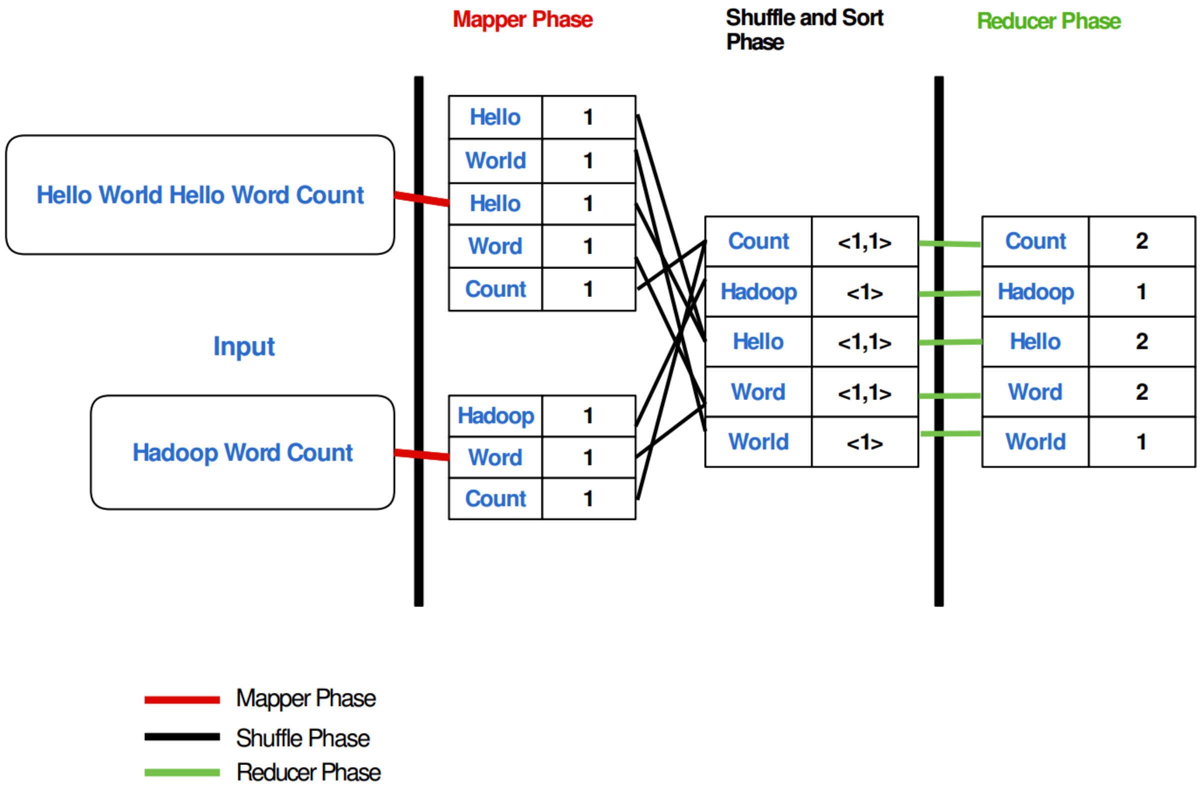


Figure 2.1: Hadoop MapReduce example WordCount

2.2 Apache Hadoop YARN

Earlier implementation of Hadoop MapReduce included Job and Task trackers. They used to handle the processing of map and reduce slots (tasks) on different nodes of the Hadoop cluster. All MapReduce applications were submitted to the JobTracker which communicated with the NameNode regarding the input storage of the data on different DataNodes. Each node processed the allocated map or reduce slots which were monitored using a TaskTracker. If any of the slots on any of the nodes failed, then it allocated a new slot. The JobTracker was responsible for scheduling and allocating resources to the map and reduce slots. Both JobTracker and TaskTracker are now replaced in newer versions of Hadoop by YARN. The newer version of Hadoop also includes processing of many other applications which can take graphical inputs or streaming data inputs along with processing of MapReduce applications.

Apache YARN [6], also called MRv2, segregates the JobTracker's task of resource management and task scheduling per application. YARN architecture consists of a global ResourceManager and several NodeManagers running on different nodes. ResourceManager primarily constitutes of a scheduler and works with NodeManagers and ApplicationMasters. Figure 2.2 shows the details about YARN architecture. Users submit the application to the ResourceManager and corresponding to every application there is an ApplicationMaster. The ApplicationMaster1 (AM1) in Figure 2.2 communicates with ResourceManager to request containers to process the application shown by thick arrow with the solid head. This in return gets approved by ResourceManager after checking the memory available on the different nodes with NodeManager shown by the dotted arrow with no filled head. ApplicationMaster itself is started by ResourceManager inside a container. Containers are allocated by ResourceManager on different nodes.

YARN provides a Capacity Scheduler [8] and other schedulers to handle multiple applications submitted to ResourceManager. The number of NodeManagers depends on the number of nodes running on the cluster, and number of ApplicationMasters depends on the number of applications submitted to the Hadoop cluster. No resources are allocated without approval of ResourceManager.

With YARN architecture, Hadoop can process many more types of applications like Dryad [9], SPARK [10] along with MapReduce applications. For every application submitted, the ResourceManager starts an ApplicationMaster. The MapReduce ApplicationMaster is called as `MRAppMaster`.

Every task of any running application requests allocation of certain resources. Resources in Hadoop framework are measured in terms of `capability` <Memory, vCores>. Depending upon the locality of data in the cluster and available resources, ResourceManager allocates resources on the different nodes of the cluster. Some of the key features of YARN are introduction of containers and data locality. Every map/reduce task is processed inside a container as a new `YARNchild` process. In YARN, allocating equal resources to all the map tasks, refers to allocation of containers with same capabilities <Memory, vCores>.

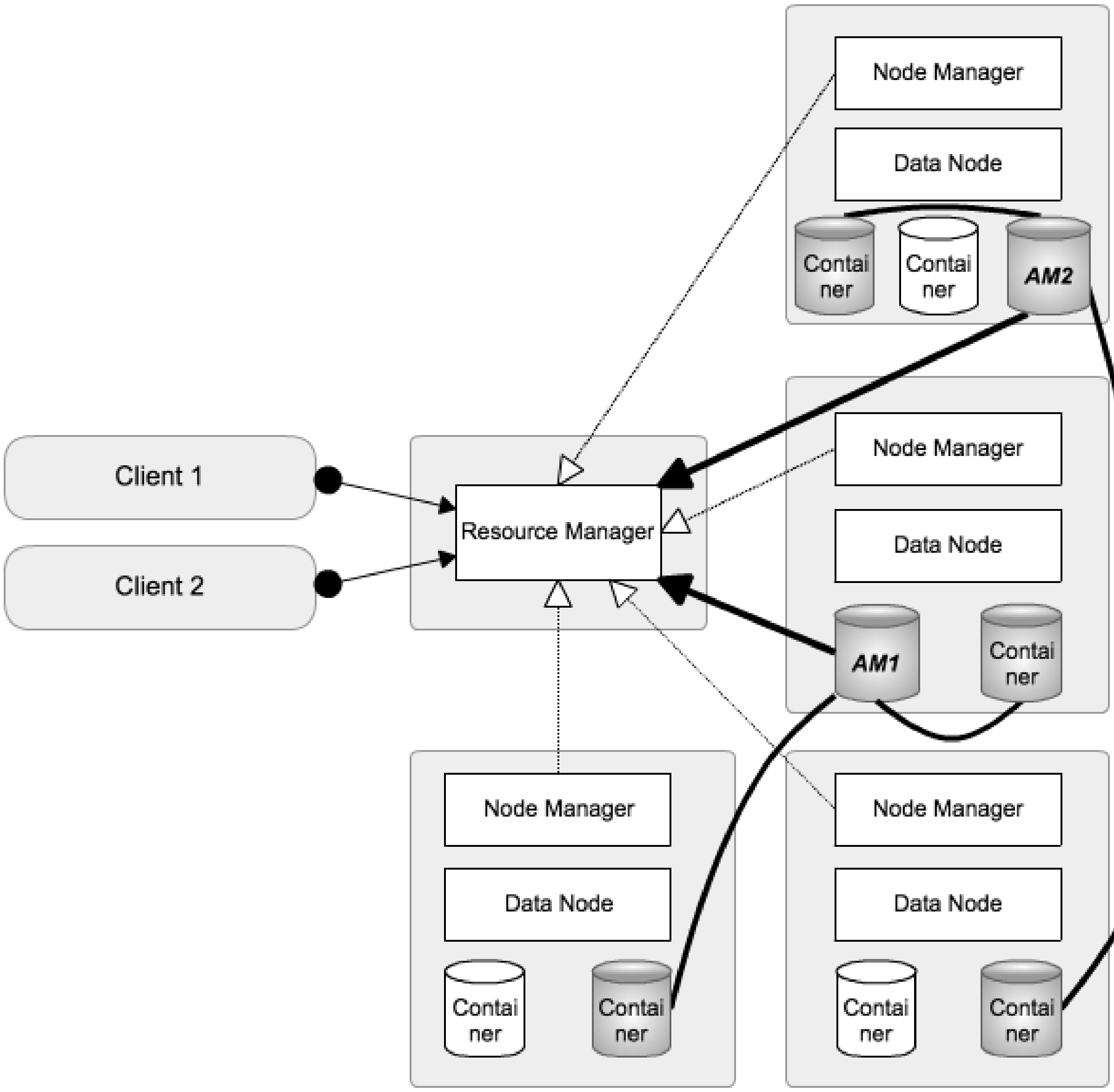


Figure 2.2: YARN architecture: showing how applications are allocated containers

To execute a task inside a container, along with certain capabilities <Memory, vCores>, Java heap memory is also required. Without sufficient heap memory and capability <Memory, vCores> to process the allocated task, a task results in a failure. The heap footprint or consumption of heap memory increases with the increase in amount of data a task processes. If a task is allocated resources and heap which are not sufficient enough to process it, the task fails, and thus the application fails too.

ApplicationMaster starts with the start of submission of application by the user and runs inside a container on a node in the cluster allocated by the ResourceManager. ApplicationMaster raises `ResourceRequests` according to the default configuration or according to the configuration set by the user. All `ResourceRequests` in the current state of art demand exactly same amount of resources to process any map task. If a map task requires higher resources to process the records, the user needs to manually set map parameters to increase the resource requirement. This increase in the amount of resources set by user reflects on every `ResourceRequest` and is applied to all the map tasks, as currently all the map tasks are treated equally.

ApplicationMasters `ResourceRequest` are fulfilled by ResourceManagers task scheduler. Based on data locality and available resources on nodes, it allocates the resource to the `ResourceRequests`. `ResourceAllocation` is a process which involves interaction of ResourceManager with NodeManager. Every node has a NodeManager running on it which manages and monitors all the resources available on that node. Resources on nodes are allocated as containers which executes the tasks assigned to them. The progress of the tasks running inside the containers of a node are well-monitored by NodeManager. In case of any failure, the NodeManager reports back the details of the failed task to the ResourceManager. If frequent task failures occur on any node, it causes the ResourceManager to `blacklist` that node and no task in future is allocated on that node.

2.3 Running MapReduce on YARN

MapReduce applications along with other applications run on YARN. As referred above, every application creates an ApplicationMaster which deals further with the global ResourceManager. Similarly, MapReduce applications also create their own `MRAppMaster` which request resources to the ResourceManager for processing different tasks.

Whenever a MapReduce job is submitted by the user, the job is given to a ResourceManager. The ResourceManager communicates with one of the NodeManagers to allocate a container for `MRAppMaster`. Once the `MRAppMaster` is started, the record splits are checked to get the number of map tasks required for the MapReduce application. Corresponding to the record splits the container resource allocation requests are made from the `TaskAttemptImpl`. A task goes through different transitions here. The transition of any MapReduce task from `unassigned` to `assigned` state signifies allocation of a container to that task. The ResourceManager allocates all the requested containers using `RMContainerAllocator`. Once all the requests are allocated with help from scheduler considering data locality and efficiency, tasks are marked as running. They all are executed inside their respective containers. The transition from assigned to running and from running to successful marks the completion of the task. All the map tasks are given same set of priorities and resources. The reducer kick starts after certain number of map tasks are completed successfully and is given a higher priority. The output of the map tasks are stored as intermediate outputs in memory and go through shuffle and sort phases. Based on the logic specified by the user for the mapper, intermediate outputs are produced. The reducer collects the intermediate outputs in form of <Key, Value> pairs and performs the computation specified in the reducer function to generate the final results.

CHAPTER 3. MOTIVATION

Hadoop [1] was designed to process very large data sets. In this section we examine two real-world data sets to show how they exhibit asymmetry in the size of their records and postulate how that might affect performance.

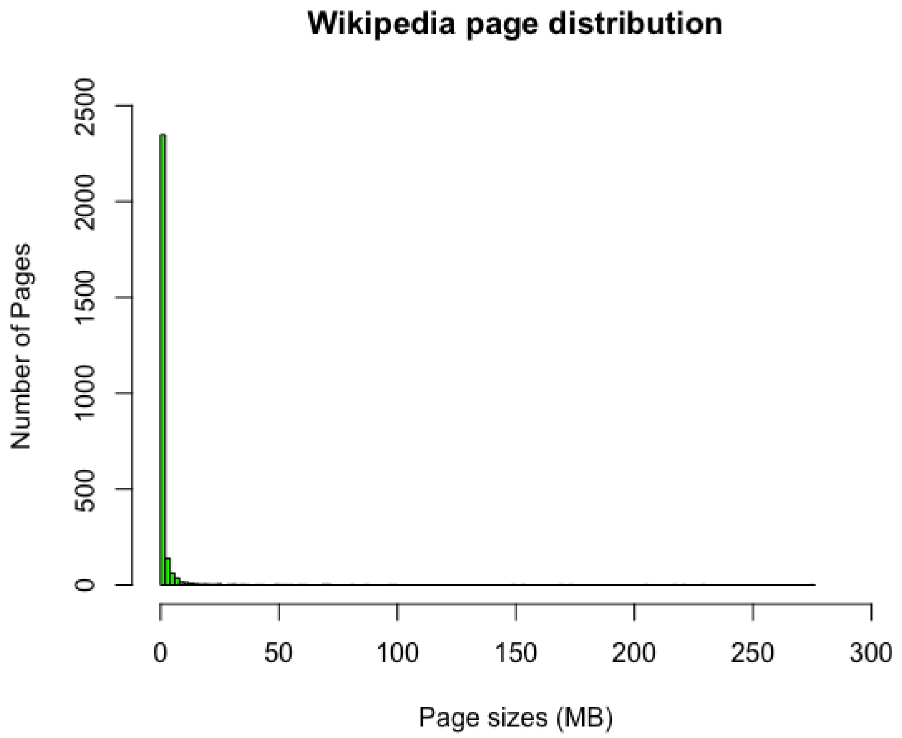


Figure 3.1: Distribution of Wikipedia page sizes

3.1 Wikipedia Page Dumps

Wikipedia [11] provides XML dumps of their data [12]. When considering pages (with page content, full revision history, and comments), there is a lot of variance in the size of individual pages. For example, consider Figure 3.1 that shows a histogram of page size (in MB) for a 5.7GB subset of this dump consisting of 2,682 pages. It is obvious this data set is asymmetric, following roughly a log distribution with many small pages (less than 200MB) and long tail with a few larger pages.

If someone wants to process this data with Hadoop, they will have to configure Hadoop's map task memory to support the largest page. For this data set that requires 2GB containers (actually just slightly over 1GB, but YARN only increments containers by 1GB). If we imagine a cluster with 100GB available memory, this would mean we could only allocate 50 map tasks in parallel. The majority of those tasks would process the smaller data and thus essentially waste memory resources. In addition, the cluster is under-utilized in terms of CPU as each map still only requests a single vCPU. In the ideal situation, we would prefer to instead allocate perhaps a single 2GB container and then 98 1GB containers. This would drastically improve the parallelization in the system and should lead to much improved performance!

3.2 SourceForge Projects

Source code repositories like SourceForge [13] contain projects of vastly different sizes. These repositories contain source code, revision history, issue reports, etc. The amount of data corresponding to each project on SourceForge can vary depending upon the size of the source code and number of revisions. Some projects are extremely small, consisting of perhaps a handful of files and possibly even only a single revision. This happens when people are just archiving older projects. Other projects are extremely complex and active, consisting of thousands of files and many revisions on each file. Thus this data set also exhibits asymmetry. We will see how this data set affects performance in later chapters when we use the Boa project data (which is derived from SourceForge data).

CHAPTER 4. RELATED WORKS

In this chapter we discuss works closely related to the problem of efficiently processing asymmetric data sets. We group these works into several high-level categories: time based approaches, priority based approaches, configuration based approaches, and simulation approaches.

4.1 Time Based Approaches

Zaharia *et al.* [14] discussed the scheduling of map and reduce tasks based on time delay to help ensure the tasks support data locality. The allocation of a task to a specific slot is delayed a small amount of time to give an opportunity to the scheduler to assign the task and achieve data locality. Hadoop by default tries to allocate tasks by considering data locality. Similarly, Polo *et al.* [15] discussed the resource-aware, adaptive placement of map or reduce tasks by implementing a resource-aware scheduling (RAS) algorithm to assign tasks to slots in a dynamic fashion. Tasks for a specific job are allocated to a slot based on a job completion time estimator and user-specified completion time goals for a job. When multiple jobs are submitted, according to the user provided completion time goals for each job, more tasks from a specific job are placed in slots by the task tracker.

Cheng *et al.* [16] discussed the concept of heterogeneous workloads and the negative impact of scheduling which arises due to the scheduling of tasks with different memory needs. Their approach involves ordering the tasks based on calculating the expected time taken by a task to complete. Similarly, Ananthanarayanan *et al.* [17] suggested to restart the task on a different node or run a duplicate task in parallel and then kill the slower task. The restart is based on the time which a task will take to complete and if the task takes longer then estimated completion time it is restarted based on resource awareness and data locality. The task is placed in such a way to minimize the load in the network.

Chen *et al.* [18] suggested a smart speculative strategy based on progress rate and process bandwidth of the tasks. They check if there are any stragglers by using a weighted moving average to calculate a task's remaining time for completion. Depending upon the estimated time for completion they re-schedule the task. This rescheduling of tasks is not memory dependent and the job can't recover if a task completely fails before completion. Also, to re-run it successfully they need to re-configure Hadoop to allocate higher resources for every task.

Most of the time based approaches discussed involve a purposeful insertion of delay in the system to achieve better data locality. Our approach does not introduce any kind of additional delay in the system. These approaches also do not allow allocating different resources for asymmetric data sets as our approach does.

4.2 Priority Based Approaches

Visalakshi and Karthik [19] suggested that TaskTrackers are prioritized based on factors like number of failed tasks, number of tasks which ran successfully earlier, and resources available for allocation on a node. They allocate priorities to the task based on the above factors. Trying failed tasks on different nodes can result in successful run of those tasks, but if the task failed for `OutOfMemoryException`/heap issues then it can't avoid future failures, as it will again request the same amount of resources for which it failed in the past.

Li *et al.* [20] proposed a deadline based algorithm where the Hadoop slave nodes help in deciding the priority of the tasks to schedule. The time deadline and the progress a task makes helps in deciding the priority of scheduling of the task. But it also requires re-configuration of Hadoop according to the amount of resources the largest task requires to run.

4.3 Configuration Based Approaches

Configuration tuning of Hadoop has been one of the more challenging areas for researchers, as determining the configuration parameters can be a tedious task for a lot of users. Li *et al.* [21] provide an online MapReduce system called MRONLINE. MRONLINE chooses the best configuration based on test runs of the application. The configurations are generated by repeated test configuration calculation, which itself can be time consuming and may involve extra computation. If any task fails in their test runs, they have to provide higher resources to all the tasks to make them run successfully. Similar to MRONLINE, Polo *et al.* [22] also provided an online estimator of time for completion of tasks used to optimize configuration parameters. This is similar to other time-based approaches but changes the configuration of Hadoop parameters. In case of any task failure they have to re-configure all the tasks to higher resources which are equal to the resource requested by the task processing the largest record.

Just like these tuning approaches Ding *et al.* [23] proposed Jellyfish, which is also an adaptive approach to configure Hadoop. It collects performance statistics and resource allocations according to the jobs submitted by users. Their approach monitors the tasks for a certain amount of time until it generates configurations for the tasks. In the future, if the same job is submitted they can use these configurations. Also, they have to configure all the tasks equal resources in case of any job failures as to calculate configurations they need to run the application successfully at least once, hence all the tasks will be assigned higher resources. In the case of online or on-the-fly tuning, once the configurations are generated they have to kill the task and restart a new task on a new container. Killing of tasks purposefully results in extra effort on the schedulers end, as it needs to allocate new container again. Their approach allocates elastic containers which increase and shrink in capacity based on the idle available resources in the system. This avoids the data-locality of the tasks as their approach merges idle resources on different nodes. Whereas our approaches never have to kill any allocated task, or wait for any amount of time and never need to allocate higher resources to all the tasks.

Khoussainova *et al.* [24] introduced an offline configuration optimizer, Perfxplain, which automatically generates and debugs the logs of previously run applications. The general method Perfxplain uses is the comparison of logs for different jobs to identify the optimum configurations. The optimum configuration provided by them will not include the case of asymmetric data sets. When there are heterogeneous tasks present in the system, it causes higher resource allocation to every task for running the application successfully. This will again allocate all the task the maximum resources and never consider the case of allocating specific tasks more resources, since logs will only be generated if there is at least one successful run which is a result of higher allocation of resources to every task.

4.4 Recurring Workloads and Simulation Approaches

MapReduce applications sometimes involve re-processing of the same input data sets. In the past, there has been research related to caching and improving the processing of Hadoop MapReduce applications. Bu *et al.* [25] have looked at applications which deal with such recurring loads. They show with the help of smart caching, the processing time for computation of recurring loads can be improved. However, for heterogeneous workloads their work can't provide optimization resulting in repetitive failures because of memory or Java Heap size issues and ultimately resulting in re-configuration of all the tasks higher resources.

Simulation based approaches hold job execution until it is simulated and optimum configuration parameters are inferred. Wang *et al.* [26] suggested to simulate for faster processing times. An expected application performance is calculated by a MapReduce application simulator called MRPref. MRPref has access to the network topology and node details. Their simulation results in modification of parameters like the number of reducer tasks running in a job, replication factor on HDFS, data locality and task scheduling on nodes. Similarly Herodotou *et al.* [27] maintain performance metrics for each job which depend on configuration, resource allocation, input data, job's performance, and based on the performance of the completed task they profiled. These simulations are not feasible in the case of heterogeneous task allocations. If the data is distributed over all the map tasks evenly, we can plan and estimate the running time and performance based on simulation. But if the data is distributed asymmetrically it will always require re-configurations of Hadoop requesting higher resources for all the tasks.

4.4.1 Auto-Configuration and Machine Learning Based Approaches

Lama *et al.* [28] suggest the use of a machine learning approach for automatic allocation of resources by providing parameters to minimize the cost in terms of resources. AROMA uses machine learning for resource allocation. Based on past ran jobs it suggests the configuration of parameters. Similarly, Verma *et al.* [29] suggested a time deadline based approach to process the MapReduce applications in a shared cluster. The applications are allocated resources according to the soft deadline decided. Their approach consists of a job profiler, resource calculator, and a performance model. This approach doesn't directly relate to our approach but when the different applications are restricted for completion in time, their approach allocates more resources to the whole application in the cluster (not the tasks) compared to other running applications, so that it tries to process faster.

Liao *et al.* [30] suggested a cost-based and machine learning based auto-tuning approach. They fine tune the map memory parameter of Hadoop MapReduce which affects all the map tasks rather than individual map task. Also to configure Hadoop, they require approximately *30* runs. They propose a search based engine which checks the configuration logs, and modifies the default parameter which affects the memory of all the map tasks.

4.5 Summary

In summary, task scheduling, resource allocation based on statistical analysis of configurations, placement of nodes in network for better performance, and machine learning approaches are some of the areas which have been addressed in the past. However, all of these approaches would configure Hadoop such that every map has the higher resources needed for the largest record, thus lowering the overall parallelism in the cluster when compared to our approach. Many of these approaches can be complementary to ours and in the future we may investigate combining them.

CHAPTER 5. APPROACH

In this chapter we discuss our approaches in detail and outline their implementation. The three approaches we propose are: Fall-back, Manual, and automatic inference. All the three approaches use the concept of allocation of higher resources to the tasks which requires higher resources rather than allocating higher resources to every task. We implemented our approaches on Hadoop version 2.7.1, but the basic idea should apply to any YARN-based Hadoop. In the following sections we describe the general flow of our approaches along with details about their implementation.

5.1 Approach 1: Fall-Back Approach

The Fall-back approach is an effort to provide greater fault tolerance along with faster processing to Hadoop MapReduce applications processing asymmetric data sets. To make the job run successfully in Default Hadoop, we need to allocate additional resources to every task rather than only certain tasks which actually require it. The current state of Hadoop doesn't support recovery from failed attempts of a task for issues where container memory is not sufficient enough to run the task or when heap size is not sufficient enough to execute that task. The submitted job is marked as failed and the only solution currently is to restart the job with different configuration allocating more memory to all tasks. To address this problem and improve fault tolerance, we suggest and propose the use of Fall-back approach where if any task fails for issues like `OutOfMemoryException`, in consecutive retries for that failed task it provides a container with more memory resources.

Currently in Hadoop, when a map task fails it creates a new failed map task attempt. The `MRAppMaster` (MapReduce application master) again requests resources from the ResourceManager for the failed map task attempt. The ResourceManager then allocates the same sized container based on availability of resources on nodes and task priorities. The failed map tasks are given higher priority because if the task completely fails in all attempts, the job will be killed sooner. The shortcoming of this approach is that it tries the failed map task attempts (failed due to `OutOfMemoryException`) on the same set of resources again, ensuring the retries all fail too.

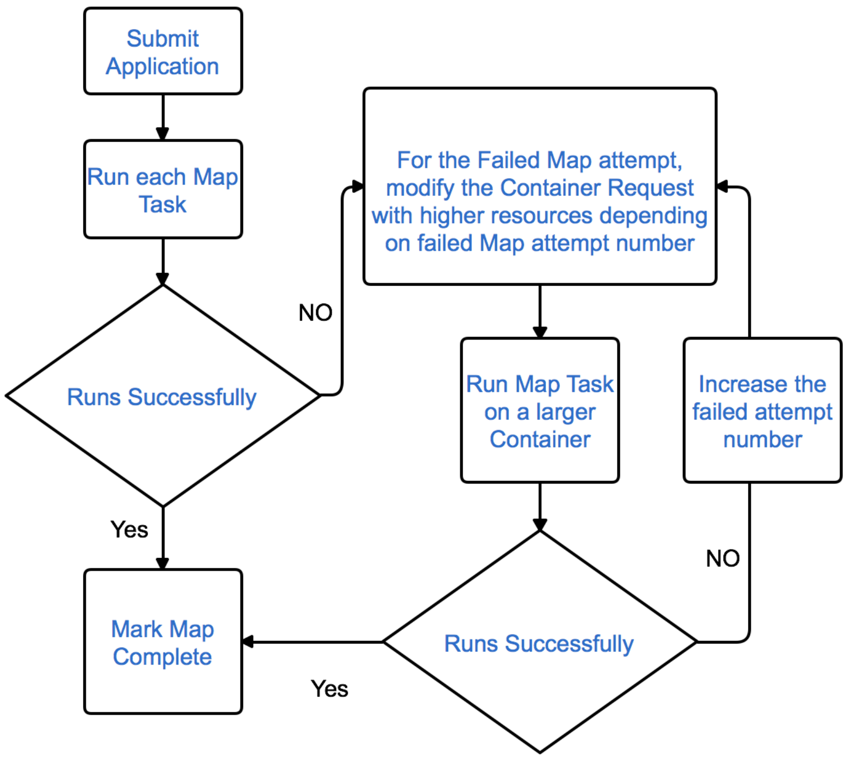

Submit Application
Run each Map Task
Runs Successfully
NO
Yes
For the Failed Map attempt, modify the Container Request with higher resources depending on failed Map attempt number
Run Map Task on a larger Container
Increase the failed attempt number
NO
Mark Map Complete
Yes
Runs Successfully

Figure 5.1: General flow of Fall-back approach

Instead, Figure 5.1 shows an overview of the Fall-back approach. Whenever a map task is initiated, a new task attempt is created. All the transitions of a task are implemented in the `TaskAttemptImpl` class. Based on the type of event, a transition is triggered. For a failed map task or a new task, the `RequestContainerTransition` API creates a `ContainerRequestEvent`. Based on the task attempt number and reason of failure, we modify this `ContainerRequestEvent`. The container memory requested for every new failed attempt is doubled and the Java heap size is set to 0.8 times the requested container memory. Thus, the failed attempt is made to run on a larger container, giving that particular task more resources to hopefully process the data successfully.

The ResourceManager is started with Hadoop cluster and handles the container requests by the `TaskAttemptImpl`. The RMContainerAllocator confirms the event type of

`ContainerAllocatorEvent`, if it is of type `ContainerRequest` it generates a request event called `reqEvent` and further checks the type of task (map or reduce). We modified the priority associated with every reqEvent for every map task. If the map task is a retry of a failed task it will be allocated a priority based on the amount of memory it will request. All failed tasks are assigned a higher priority, with larger containers having the highest priority.

With these changes, a map task failing for memory issues will be given more and more memory resources (doubling with each retry) until either it succeeds, there are no containers able to provide the requested resources, or it is still unable to succeed with the maximum resources allowed. These tasks are given higher priority to help ensure if they are unable to succeed, the whole job will fail as early as possible. This is consistent with the unmodified Hadoop's philosophy.

5.2 Approach 2: Manually Specifying Large Tasks

The second approach to handling asymmetric data sets is a Manual approach. The Manual approach allows users to specify those tasks that are large and will require higher resources. This is specified via a custom configuration parameter. Thus if users know which tasks in the data set are large and require more resources, they can manually specify those tasks and avoid the need to rely on the Fall-back approach. This should increase the overall performance as the large tasks do not require to run, fail, and retry with more resources instead immediately allocating the higher resources to the tasks.

For this approach we introduced a new configuration property as a part of *mapred-site.xml* named `maptaskmemory`. This property expects a comma-separated list of pairs of map task number and the amount of memory to allocate for that task. For example, if map tasks 3, 6 and 30 require more memory, users can specify details about these tasks in `maptaskmemory` as: "`3:2048, 6:3076, 30:2048`". In this example, map task 6 needs 3GB while map tasks 3 and 30 need 2GB. We modified Hadoop to read this property and allocate mentioned resources to the corresponding task, which currently is not possible in the Default version of Hadoop. Thus containers for these maps which are specified by the user will be allocated customized resources.

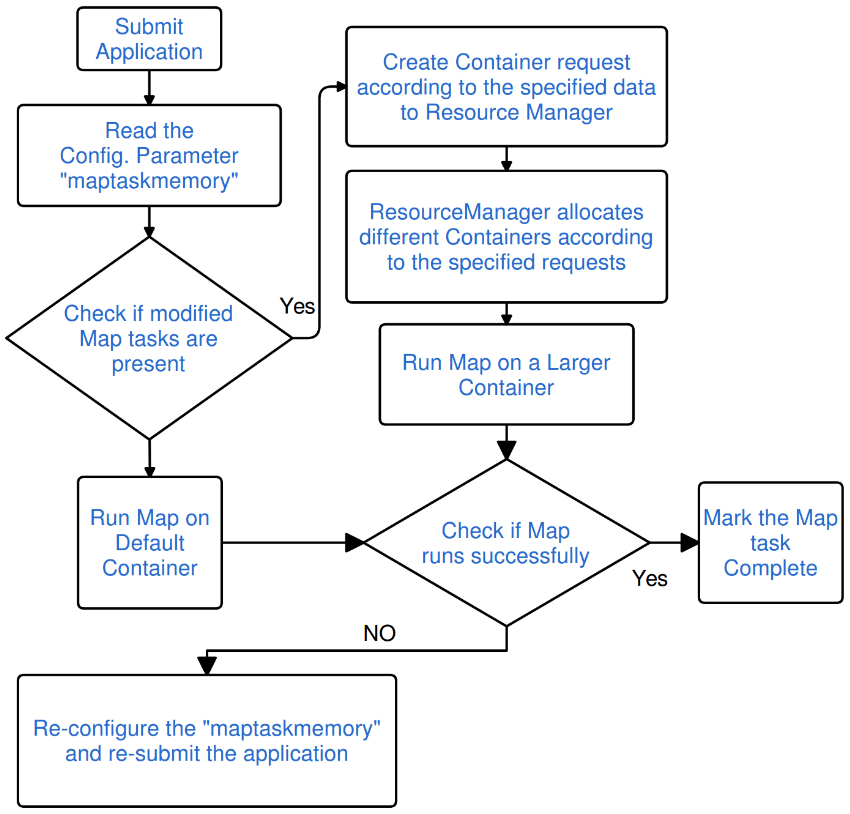

Submit Application
Read the Config. Parameter "maptaskmemory"
Check if modified Map tasks are present
Yes
Create Container request according to the specified data to Resource Manager
ResourceManager allocates different Containers according to the specified requests
Run Map on a Larger Container
Run Map on Default Container
Check if Map runs successfully
Yes
Mark the Map task Complete
NO
Re-configure the "maptaskmemory" and re-submit the application

Figure 5.2: General flow of Manual approach

Figure 5.2 shows an overview of the Manual approach. When a job is submitted, map tasks are immediately allocated resources. `TaskAttemptImpl` constructs resource requests for all the tasks using `ContainerRequestEvent` and by reading the property `maptaskmemory` specified by user. We modified the generation of the `ResourceRequests` which can now request different size of resources. Corresponding to the modified `ContainerRequestEvent`, `RMContainerAllocator` is also modified to accept and create containers with different resources. If the container requested by the user exceeds the largest possible container size, their request will be re-sized to the largest container size. The introduced container allocation method allocates all the map task requests mentioned in the configuration property the exact memory requested. Map tasks which require higher resources are allocated larger containers along with higher heap size.

The approach uses task priorities to ensure large containers are properly allocated to the larger tasks that need them. Otherwise, a large container could be created and a small task assigned to it, thus wasting memory resources.

5.3 Approach 3: Automatically Inferring Large Tasks

In this approach, we propose automatically inferring of tasks which need more resources. To avoid user efforts to manually configure map tasks (see Section 5.2, we propose to automatically infer the large tasks which are failed based on the available logs of previous runs. If users have run jobs with a certain input data set, and if it failed or ran successfully the logs corresponding to that job are aggregated and stored in HDFS. If another user in the future runs a job on the same input, we can automatically read the logs and figure out the configuration parameter which was used for the successful run of that input data set previously, and configure Hadoop with those values for our defined `maptaskmemory` parameter. This parameter will be populated with the values corresponding to the successful runs, if any.

Figure 5.3 shows the general flow of the automated inference approach. To read the logs stored by Hadoop MapReduce from HDFS, we plan to use scripts which take the values for previous successful runs of jobs corresponding to the same input data sets. Based on the logs it determines which tasks required more resources and configures the `maptaskmemory` parameter. This process would be automatic thus avoiding the need for users to manually configure the parameter. If the approach is not able to correctly infer the right values, the Fall-back approach is still enabled and future runs could learn from these failures.

Although we did not implement this approach, we used this process to find the values for the manually defined settings in our evaluation chapter (see Chapter 6).

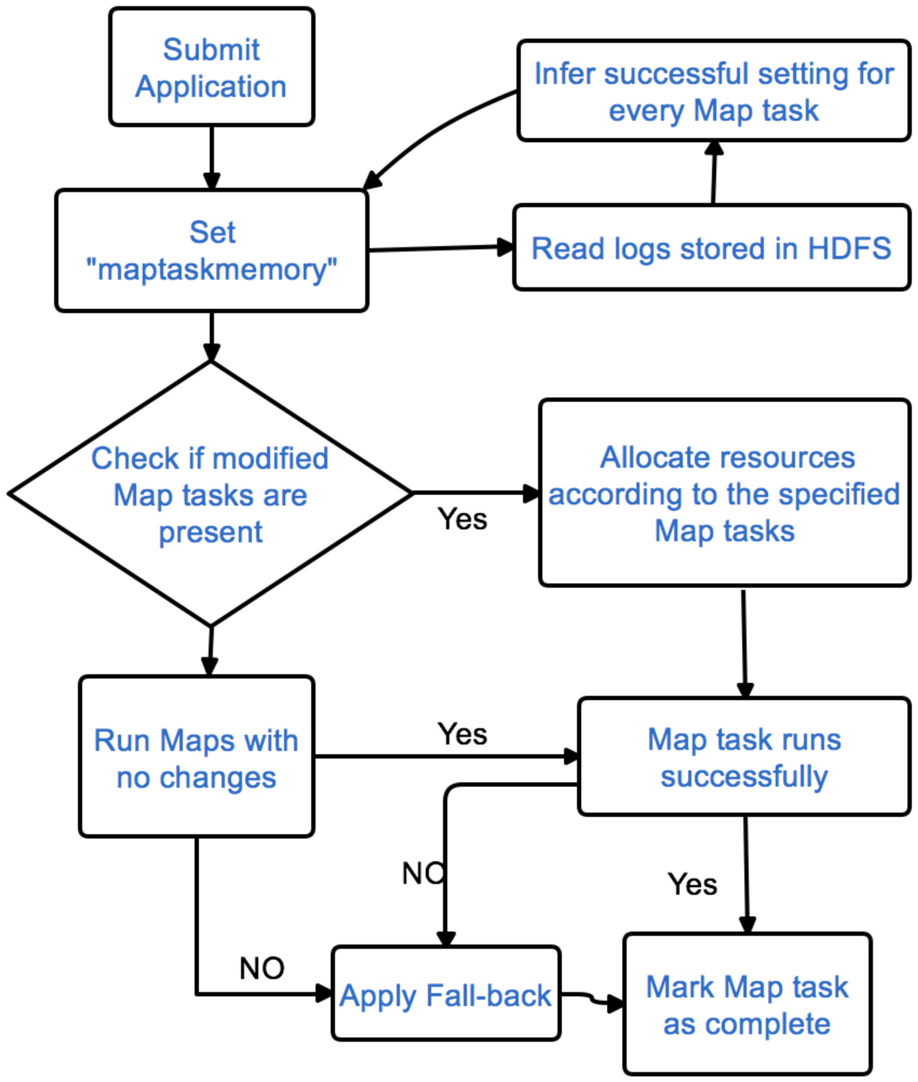

Submit Application
Infer successful setting for every Map task
Set "maptaskmemory"
Read logs stored in HDFS
Check if modified Map tasks are present
Yes
Allocate resources according to the specified Map tasks
Run Maps with no changes
Yes
Map task runs successfully
NO
Yes
NO
Apply Fall-back
Mark Map task as complete

Figure 5.3: General flow of automatic inference approach

CHAPTER 6. EVALUATION

In this chapter we evaluate our implemented approaches. To evaluate our approaches, we used cloud-based Hadoop clusters provided by CloudLab [31, 32]. In this section we discuss several research questions related to the performance of the different approaches. We implemented our approaches on Hadoop version 2.7.1. We created a CloudLab profile to install the default, unmodified Hadoop version 2.7.1 and our modified versions for Manual and Fall-back side by side. In this chapter we propose several research questions and then run experiments to evaluate each one.

6.1 Evaluation Setup

On CloudLab, which provides cloud-based clusters, we requested four node clusters (except where noted otherwise) consisting of only physical (not virtual) nodes. The Hadoop cluster follows the master/slave computation model. In our setup, a single master node runs the ResourceManager and the NameNode on it, whereas the slave nodes run a NodeManager and a DataNode on them. All the nodes communicate over an Ethernet link. When the nodes are well connected and successfully pass the link test provided by CloudLab the cluster is marked as up and running. In the following section we provide more information about the specification of the hardware, software and configuration of the cluster.

6.1.1 Hardware and System Configuration

The Hadoop cluster on CloudLab provides an option to select number of nodes along with option of choosing the hardware type of the nodes, which in our case are all physical nodes. We used a cluster consisting of three slave nodes and a single master node. Figure 6.1 is the profile of the cluster used to install the disk image on all the nodes of the cluster, which are running Ubuntu 14.04.1 LTS. The hardware type of the nodes on our CloudLab clusters was enforced to `c8220`, consisting of 2x10 core Intel 2.2GHz CPUs, 256GB memory, and dual-port 10GB Ethernet. The

```python
import geni.portal as portal
import geni.rspec.pg as RSpec
import geni.rspec.igext

def Node( name ):
    node = RSpec.RawPC( name )
    node.hardware_type = "c8220"
    node.disk_image = "urn:publicid:IDN+apt.emulab.net+image+emulab-ops:UBUNTU14-64-STD"
    node.addService( RSpec.Install( "http://apache.cs.utah.edu/hadoop/common/hadoop-2.7.1/hadoop-2.7.1.tar.gz", "/usr/local" ) )

    node.addService( RSpec.Install( "http://boa.cs.iastate.edu/cloudlab/hadoop-2.7.1-setup.tar.gz", "/tmp" ) )
    node.addService( RSpec.Execute( "sh", "sudo /tmp/setup/java7-install.sh 2.7.1" ) )
    node.addService( RSpec.Execute( "sh", "sudo /tmp/setup/init-hdfs.sh 2.7.1" ) )

    node.addService( RSpec.Install( "http://boa.cs.iastate.edu/shubh/hadoop-2.7.1-fallback.tar.gz", "/tmp/fallback" ) )
    node.addService( RSpec.Execute( "sh", "sudo /tmp/setup/mv-hadoop.sh 2.7.1 fallback" ) )

    node.addService( RSpec.Install( "http://boa.cs.iastate.edu/shubh/hadoop-2.7.1-manual.tar.gz", "/tmp/manual" ) )
    node.addService( RSpec.Execute( "sh", "sudo /tmp/setup/mv-hadoop.sh 2.7.1 manual" ) )
    return node

pc = portal.Context()
rspec = RSpec.Request()
lan = RSpec.LAN()
rspec.addResource( lan )

for i in range( 4 ):
    if (i == 0)
        node = Node( "master" )
    else
        node = Node( "slave" + str( i ) )
    iface = node.addInterface( "if0" )
    lan.addInterface( iface )
    rspec.addResource( node )

tour = geni.rspec.igext.Tour()
rspec.addTour( tour )

pc.printRequestRSpec( rspec )
```

Figure 6.1: CloudLab profile for our Hadoop cluster with all 3 versions installed

profile script ensures that we install the default and modified versions of Hadoop on all the nodes and use same type of hardware for every experiment.

For Hadoop, we used a standard configuration (except where needed to modify the map memory). We configured two, 1 TB 7200 RPM SATA disk drives per slave node for use by HDFS [7]. We configured the number of virtual cores (vCores) and memory on each slave as 10 vCores and 10GB, giving a total of 30 vCores and 30GB available to Hadoop. Every task by default gets 1024MB container, 800MB JVM heap size, and 1 vCore.

For each experiment in this chapter, all data was always collected on the same CloudLab cluster instance. Thus results within an experiment are directly comparable but results between experiments were possibly run on different physical machines (though with the same hardware) and may not be comparable.

6.2 RQ1: Do our Approaches Incur Significant Overhead?

The first research question we investigated was if the new approaches incur any significant overhead when compared to the default approach. To answer this question, we run the same Hadoop program on the same input data set and with all versions using the same memory configuration (which ensures no memory failures).

We recorded the time taken by every version of Hadoop with data sets that don't require any changes in configuration of Hadoop and run successfully in the very first attempt. With homogeneously distributed data, each map processes the same amount of data and is requesting same amount of resources. The choice of homogeneously distributed data sets were purposefully made so that all the tasks run without any failure. By doing so, we calculated if our approaches result in any significant overhead.

The time taken by different versions to process the symmetric data was recorded for 10 runs on each approach, after running and not collecting performance results once to allow for warm up effects. For this purpose we used an identity map function and set number of reducers to 0. We

Table 6.1: Time taken to process 100 maps for data set of 1MB files

Experiment	Default	Fall-back	Manual
Run 1	28s	29s	29s
Run 2	25s	26s	25s
Run 3	26s	27s	27s
Run 4	26s	26s	26s
Run 5	25s	26s	25s
Run 6	26s	27s	26s
Run 7	26s	27s	28s
Run 8	26s	28s	26s
Run 9	27s	28s	28s
Run 10	26s	26s	26s
Average	26.1s	27s (+0.9s)	26.6s (+0.5s)

Table 6.2: Time taken to process 200 maps for data set of 1MB files

Experiment	Default	Fall-back	Manual
Run 1	43s	44s	44s
Run 2	43s	43s	44s
Run 3	44s	44s	43s
Run 4	43s	44s	42s
Run 5	43s	43s	44s
Run 6	43s	44s	43s
Run 7	43s	43s	43s
Run 8	42s	44s	44s
Run 9	43s	43s	44s
Run 10	43s	44s	44s
Average	43s	43.6 (+0.6s)	43.5s (+0.5s)

Table 6.3: Time taken to process 500 maps for data set of 1MB files

Experiment	Default	Fall-back	Manual
Run 1	92s	92s	94s
Run 2	94s	93s	94s
Run 3	92s	93s	92s
Run 4	93s	95s	93s
Run 5	92s	94s	92s
Run 6	93s	94s	94s
Run 7	93s	92s	92s
Run 8	93s	93s	94s
Run 9	93s	93s	94s
Run 10	93s	93s	92s
Average	92.8s	93.2s (+0.4s)	93.1s (+0.3s)

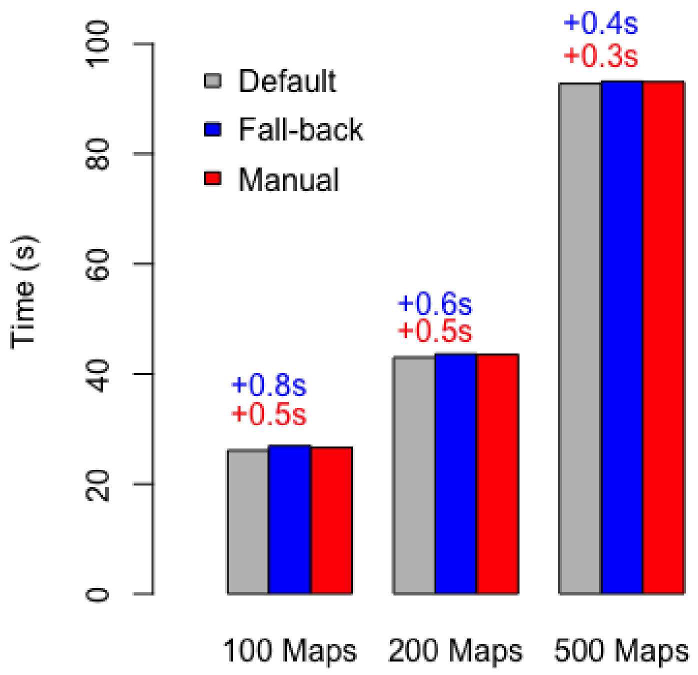


Figure 6.2: Number of maps: 100, 200 and 500 data sets: 100MB, 200MB and 500MB

performed the calculations on three different data sets consisting of 100, 200 and 500 1MB files. We used a whole file input format that provides each file as a record to each map task. The goal was to see if the Fall-back or Manual approaches were slower in such a case, indicating what the overhead of the approaches are.

The Fall-back and Manual approaches both show overhead of approximately less then 1 sec and this overhead was constant across the different data sets. Table 6.1, Table 6.2 and Table 6.3 show all the results for the three data sets. Figure 6.2 shows the average values for each approach on each data set. The difference in average run-time between the Default, Fall-back and Manual versions of Hadoop are approximately constant and doesn't vary with increase in the number of files. As the number of maps equals to the number of files in the data sets and the overhead remains constant with the increase in number of maps, our approach doesn't introduce any significant overhead in the system. In the worst case, the Fall-back approach incurred a 3.4% overhead for the smallest files. While the overhead is roughly constant, as a percentage it will change when the time for maps decreases.

Table 6.4: *Small*, *Medium* and *Large* data sets, all with 100MB small files

How Many Large	Small (500MB large files)	Medium (1,050MB large files)	Large (2,000MB large files)
10%	13.20GB	19.04GB	28.32GB
20%	17.57GB	28.32GB	46.87GB
30%	21.48GB	37.59GB	65.42GB
40%	25.39GB	46.95GB	83.98GB
50%	29.29GB	56.15GB	102.53GB

6.3 RQ2: Do Our Approaches Reduce Overall Time Taken to Run Successfully?

To evaluate if our suggested approaches improve the overall run-time while processing asymmetric data sets, we compared the run-times of our Fall-back and Manual approaches to the Default. The data sets used were divided into three categories: Small, Medium and Large. These data sets are asymmetric and consist of small files of size 100MB, along with large file size which vary according to the type of data set. Table 6.4 shows details about these data sets. For the Small category the large file size is 500MB, for the Medium category the large file size is 1,050MB and for the Large category the large file size is 2,000MB.

After 10 runs for every data set in each category we recorded all the time details in Table 6.5, Table 6.6, and Table 6.7. We recorded the run times for different categories and different approaches and calculated the minimum, maximum, median, and mean for each of them. The relative value (shown in parentheses for Fall-back and Manual approaches) provides the relative performance with respect to the Default version which signifies that the approach is X times faster or slower compared to the Default approach.

When we run the Default and Manual versions for any of the asymmetric data sets, both of these approaches fail for the very first run due to out of memory. Both of the versions required manual re-configuration of Hadoop. But the Fall-back version never fails for any of the asymmetric data sets as it recovers on its own and re-configures itself while running. This gives added advantage to the Fall-back approach. Once the Default and Manual versions are properly re-configured, the Manual approach shows fastest processing of data for any of the asymmetric data sets.

Table 6.5: Run-times for *small* (100mb small, 500mb large) data sets

Large	Statistic	Default	Fall-back	Manual
10%	min	43s	33s (1.30X)	30s (1.43X)
	max	45s	36s (1.25X)	31s (1.45X)
	median	43.5s	34.0s (1.27X)	30.0s (1.45X)
	mean	43.8s	34.2s (1.28X)	30.2s (1.45X)
20%	min	44s	39s (1.12X)	34s (1.29X)
	max	46s	41s (1.12X)	35s (1.31X)
	median	44.5s	39.0s (1.14X)	34.0s (1.30X)
	mean	44.8s	39.5s (1.13X)	34.3s (1.31X)
30%	min	45s	44s (1.02X)	36s (1.25X)
	max	46s	45s (1.02X)	37s (1.24X)
	median	45.5s	44.5s (1.02X)	36.0s (1.26X)
	mean	45.5s	44.5s (1.02X)	36.3s (1.25X)
40%	min	45s	49s (0.91X)	38s (1.18X)
	max	47s	52s (0.90X)	41s (1.14X)
	median	45.5s	50.0s (0.91X)	40.0s (1.13X)
	mean	45.7s	50.3s (0.91X)	39.8s (1.15X)
50%	min	45s	54s (0.83X)	42s (1.07X)
	max	47s	56s (0.83X)	43s (1.09X)
	median	47.0s	55.0s (0.85X)	42.0s (1.11X)
	mean	46.5s	55.1s (0.84X)	42.2s (1.10X)

Table 6.6: Run-times for *medium* (100mb small, 1050mb large) data sets

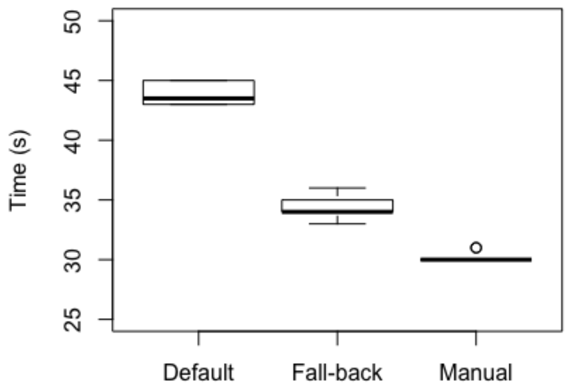
Large	Statistic	Default	Fall-back	Manual
10%	min	94s	47s (2.00X)	36s (2.61X)
	max	97s	48s (2.02X)	39s (2.48X)
	median	95.0s	47.0s (2.02X)	38.0s (2.50X)
	mean	94.8s	47.4s (2.00X)	37.8s (2.51X)
20%	min	97s	65s (1.49X)	54s (1.79X)
	max	98s	68s (1.44X)	57s (1.71X)
	median	98.0s	66.0s (1.48X)	56.0s (1.75X)
	mean	97.7s	65.9s (1.48X)	55.6s (1.76X)
30%	min	100s	80s (1.25X)	58s (1.72X)
	max	102s	82s (1.24X)	61s (1.67X)
	median	101.0s	81.0s (1.24X)	60.0s (1.68X)
	mean	100.8s	81.0s (1.24X)	59.7s (1.69X)
40%	min	104s	100s (1.04X)	71s (1.46X)
	max	106s	103s (1.02X)	74s (1.43X)
	median	105.0s	102.0s (1.02X)	72.0s (1.45X)
	mean	104.9s	101.6s (1.03X)	72.1s (1.45X)
50%	min	106s	125s (0.84X)	78s (1.35X)
	max	107s	127s (0.84X)	82s (1.35X)
	median	107.0s	126.0s (0.84X)	79.0s (1.35X)
	mean	106.7s	126.1s (0.85X)	79.4s (1.34X)

Table 6.7: Run-times for *large* (100mb small, 2000mb large) data sets

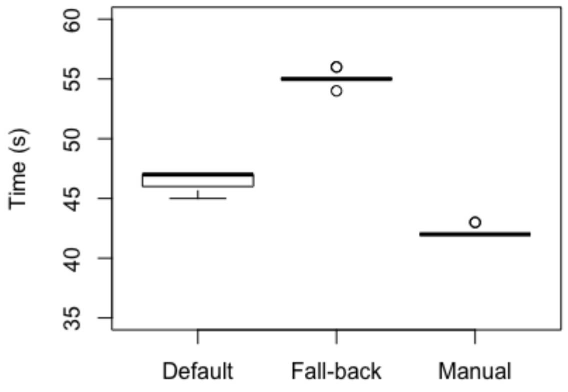
Large	Statistic	Default	Fall-back	Manual
10%	min	189s	87s (2.17X)	61s (3.09X)
	max	190s	90s (2.11X)	64s (2.96X)
	median	189.5s	89.0s (2.13X)	63.0s (3.00X)
	mean	189.5s	88.7s (2.13X)	62.9s (3.01X)
20%	min	200s	137s (1.45X)	54s (3.70X)
	max	202s	141s (1.43X)	57s (3.54X)
	median	201.0s	139.0s (1.44X)	56.0s (3.58X)
	mean	201.2s	138.9s (1.44X)	55.6s (3.61X)
30%	min	216s	207s (1.04X)	136s (1.58X)
	max	219s	210s (1.04X)	138s (1.58X)
	median	217.0s	208.0s (1.04X)	137.0s (1.58X)
	mean	217.3s	208.3s (1.04X)	137.3s (1.58X)
40%	min	224s	241s (0.92X)	162s (1.38X)
	max	226s	243s (0.93X)	163s (1.38X)
	median	226.0s	241.0s (0.93X)	162.5s (1.39X)
	mean	225.0s	241.4s (0.93X)	162.5s (1.38X)
50%	min	241s	347s (0.69X)	222s (1.08X)
	max	250s	352s (0.71X)	224s (1.11X)
	median	248.5s	350.0s (0.71X)	223.0s (1.11X)
	mean	247.9s	349.1s (0.71X)	222.8s (1.11X)

The data sets with only 10% large files, regardless of the size of those large files, showed faster run times for both of our approaches. As we move towards the data sets with 40-50% large files, the Fall-back approach takes more time when compared with the Default approach, whereas the Manual version still shows significant improvement in run-time.

Details about the results are plotted for 10% large files and 50% large files for each of the Small, Medium and Large category. The box-plots in Figure 6.3a, Figure 6.4a, and Figure 6.5a show the performance comparison when 10% of the tasks are large in every category of data sets. Similarly, box-plots in Figure 6.3b, Figure 6.4b, and Figure 6.5b show the performance comparison when 50% of the tasks are large. The performance of the Manual version is fastest in all the plots but notice the performance change for the Fall-back approach, which becomes the slowest approach at 50% large.

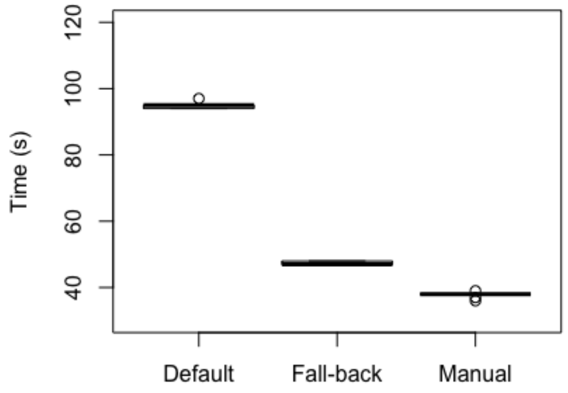


(a) 10% large files

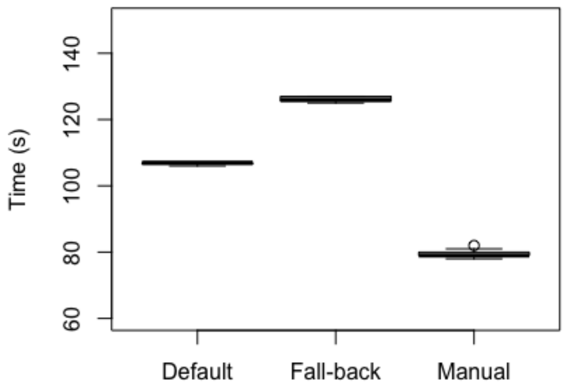


(b) 50% large files

Figure 6.3: Box plots for *Small* data set

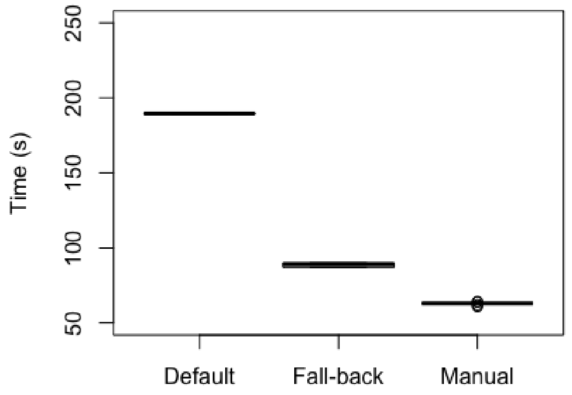


(a) 10% large files

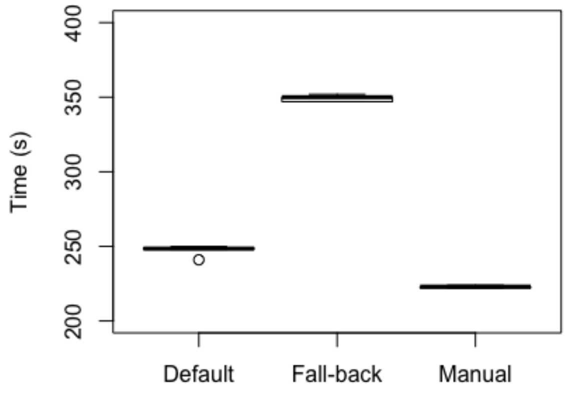


(b) 50% large files

Figure 6.4: Box plots for *Medium* data set



(a) 10% large files



(b) 50% large files

Figure 6.5: Box plots for *Large* data set

Figure 6.6, Figure 6.7, and Figure 6.8 show the mean run-time corresponding to all categories of data sets. We restricted our evaluation up to 50% large file data sets for all categories as we suggest if the data set consists of more than 50% large files, its better to allocate all of the tasks higher resources. Our approaches are backward compatible and can also allocate all the map tasks equal amount of resources in case the data sets involve more than 50% large files. By using our approaches for asymmetric data sets with up to 50% large file, we can save a significant amount of processing time.

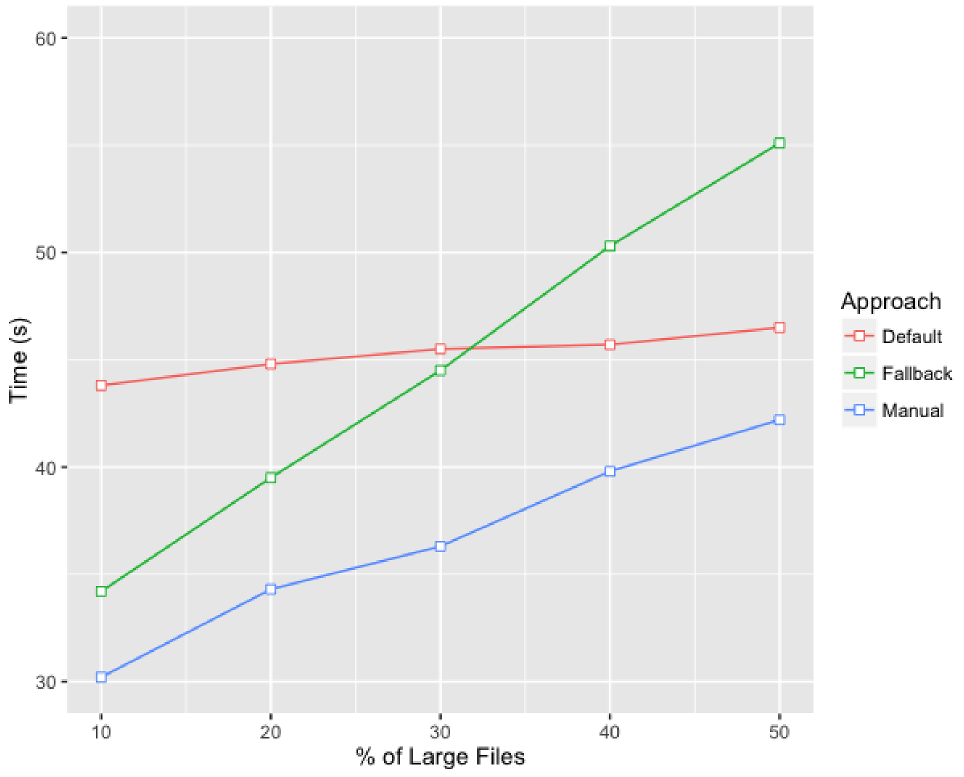


Figure 6.6: *Small* data set performance comparison

6.3.1 When 90% Large Files Are Present in the Data Set

We used a data set which contains 90% large files of size 1,050MB and small files of 100MB to compare the run-time of Default, Fall-back, and Manual versions. We performed 10 runs for each versions on the same data set. We configured the Manual approach so that 90% of the tasks were given more memory. Our Fall-back approach performed slowest compared to Default and Manual

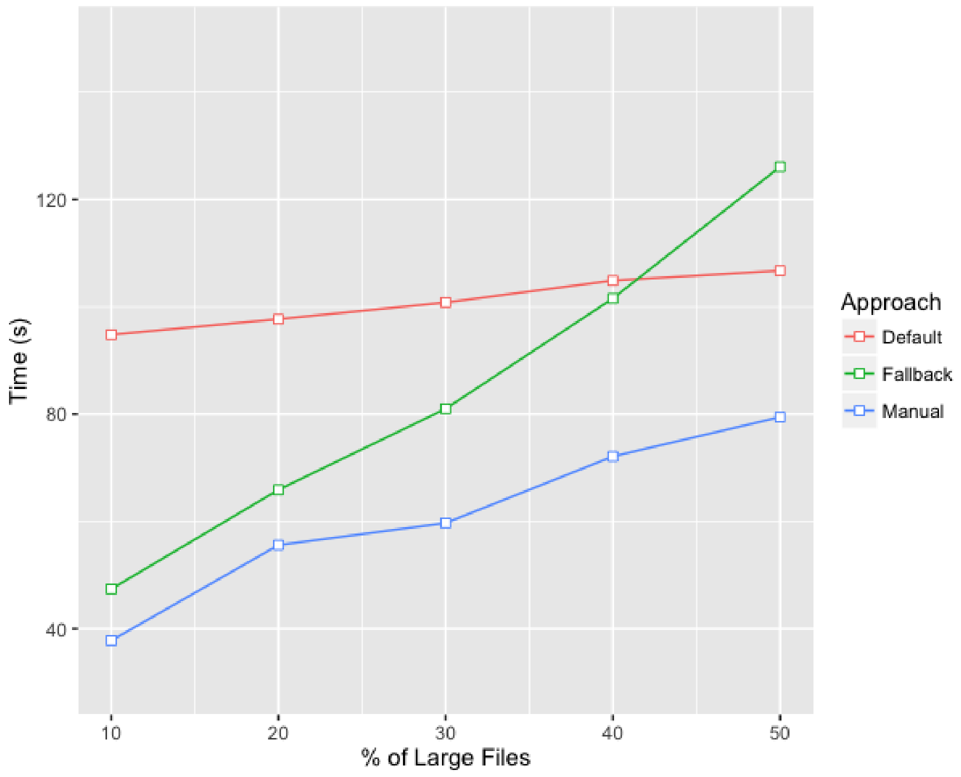


Figure 6.7: *Medium* data set performance comparison

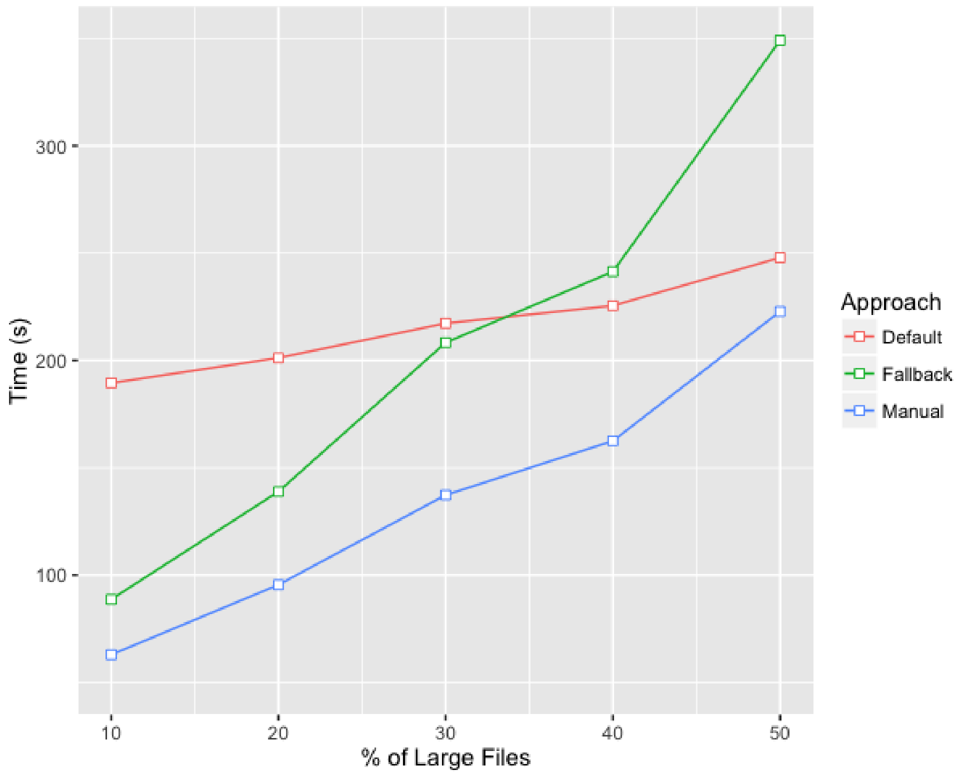


Figure 6.8: *Large* data set performance comparison

approaches whereas the Manual version was also slower compared to Default version.

Table 6.8: Time taken to process 90%, 1050MB large files

Experiment	Default	Fall-back	Manual
Run 1	124s	183s	131s
Run 2	124s	183s	131s
Run 3	124s	185s	132s
Run 4	122s	184s	128s
Run 5	126s	184s	131s
Run 6	124s	183s	131s
Run 7	124s	183s	128s
Run 8	122s	184s	129s
Run 9	126s	183s	131s
Run 10	124s	183s	132s
Average	124s	183.5s (-32.4%)	130.4s (-4.90%)

Table 6.8 shows the performance of different approaches when the data set contain 90% large files. We tried to confirm why the Manual approach was slower but we were not able to verify why every single map task takes slightly more time. We assume an overhead of roughly one second but we were unable to calculate other possible factors like the disk buffer and any caching behavior. Nevertheless, the Manual approach was comparable to the Default approach. If the number of large records are more, Fall-back approach performs significantly slower as it needs to re-attempt more tasks. However, the Fall-back approach never requires any configuration changes as compared to Default and Manual so although it was much slower, in the case where Hadoop was not properly configured to handle this data set it would fail in the Default approach while the Fall-back would actually finish successfully.

6.4 RQ3: Do Our Approaches Improve Overall Parallelism?

Our approach allocates memory more efficiently, which results in availability of more memory on different nodes which in turn allows the ResourceManager to schedule more tasks. This increases overall parallelism in the system as more tasks can be scheduled and run in parallel. The overall parallelism of the system can be defined by the number of tasks running in parallel with respect to time. More tasks in parallel typically means faster processing of the application (assuming tasks take similar time regardless of parallelism, which is not always true due to I/O bottlenecks and caching).

Each map task's start and end time was plotted with respect to time to visually show the number of tasks running. Figure 6.9, Figure 6.10 and Figure 6.11 show the tasks running in parallel for different approaches for the 10% large file data set with large file size as 2GB. Figure 6.9 shows that the Default version runs only a constant number (3, due to configuring map tasks to use 8GB memory each) of tasks in parallel throughout the processing of the data set. Whereas the Fall-back and Manual approaches show more number of tasks running in parallel. Since more parallel tasks are running in the case of our approaches, the time taken by them to process the data set is comparatively less with respect to the Default version.

The Fall-back approach fails (shown in red) for certain number of attempts depending upon the data set and number of attempts it needs to run the task successfully, but still exhibited very high parallelism for periods of time. Our Manual approach shows the maximum parallelism with respect to time. As the tasks which require higher memory are allocated higher resources directly, extra resources are never occupied and are made available to the ResourceManager to schedule more map tasks and improving overall parallelism.

6.5 RQ4: How Do Our Approaches Perform on Real-Life Data Sets?

As we discussed earlier, real-life data sets can be asymmetric. To verify how our approaches perform on such real-life data sets, we performed experiments on data sets used by Boa [3] consisting of projects from online source code repositories like SourceForge and on a dump of Wikipedia pages [12]. In this section we discuss the results on these data sets.

6.5.1 Processing Boa's Data Set

As every project in source code repositories can be of different size, Boa's data set is highly asymmetric. We used three different queries on the Boa data set. The Boa data set manually controls splitting behavior and produces 214 input splits resulting in 214 map tasks, despite being

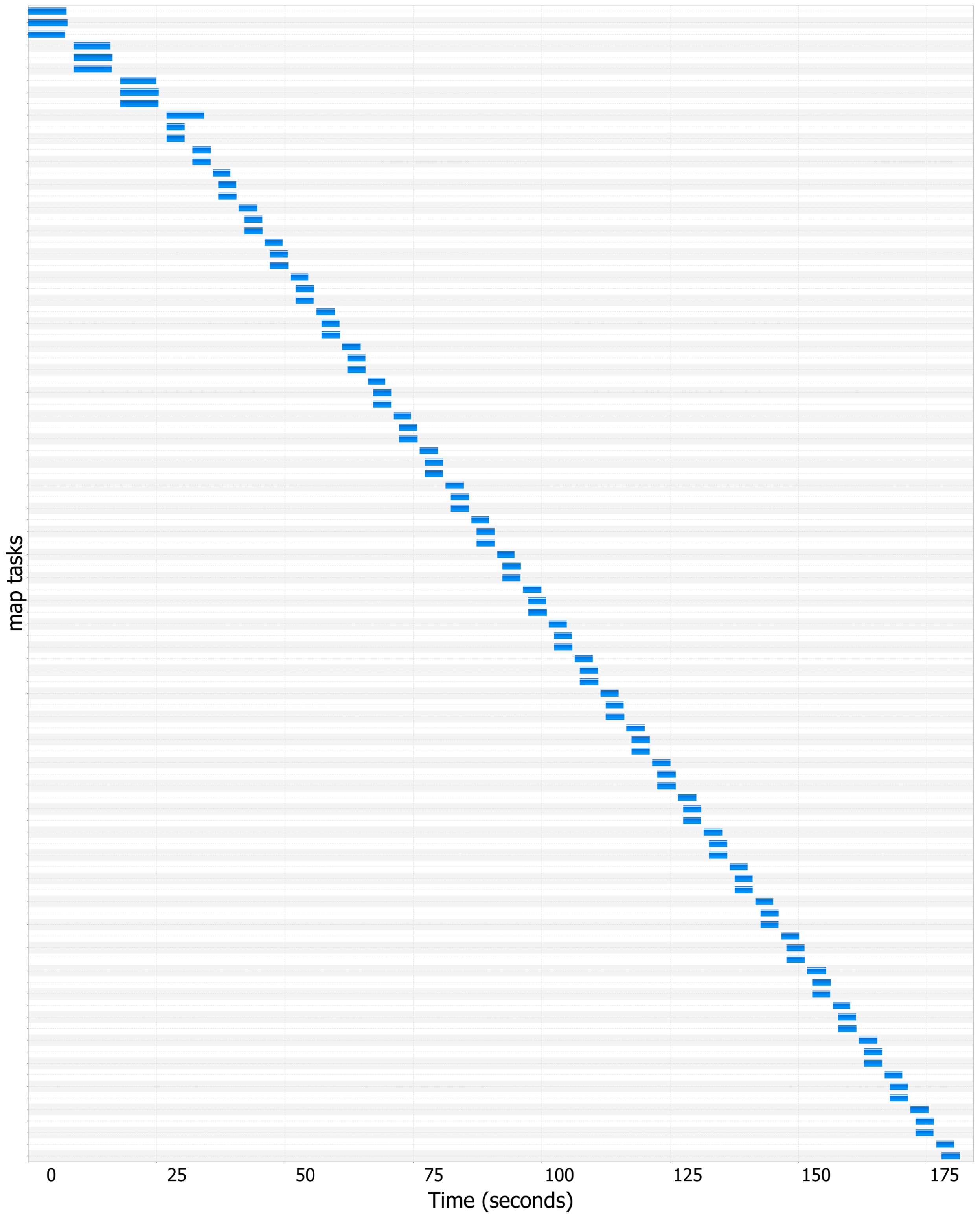


Figure 6.9: Number of tasks running in parallel for Default approach

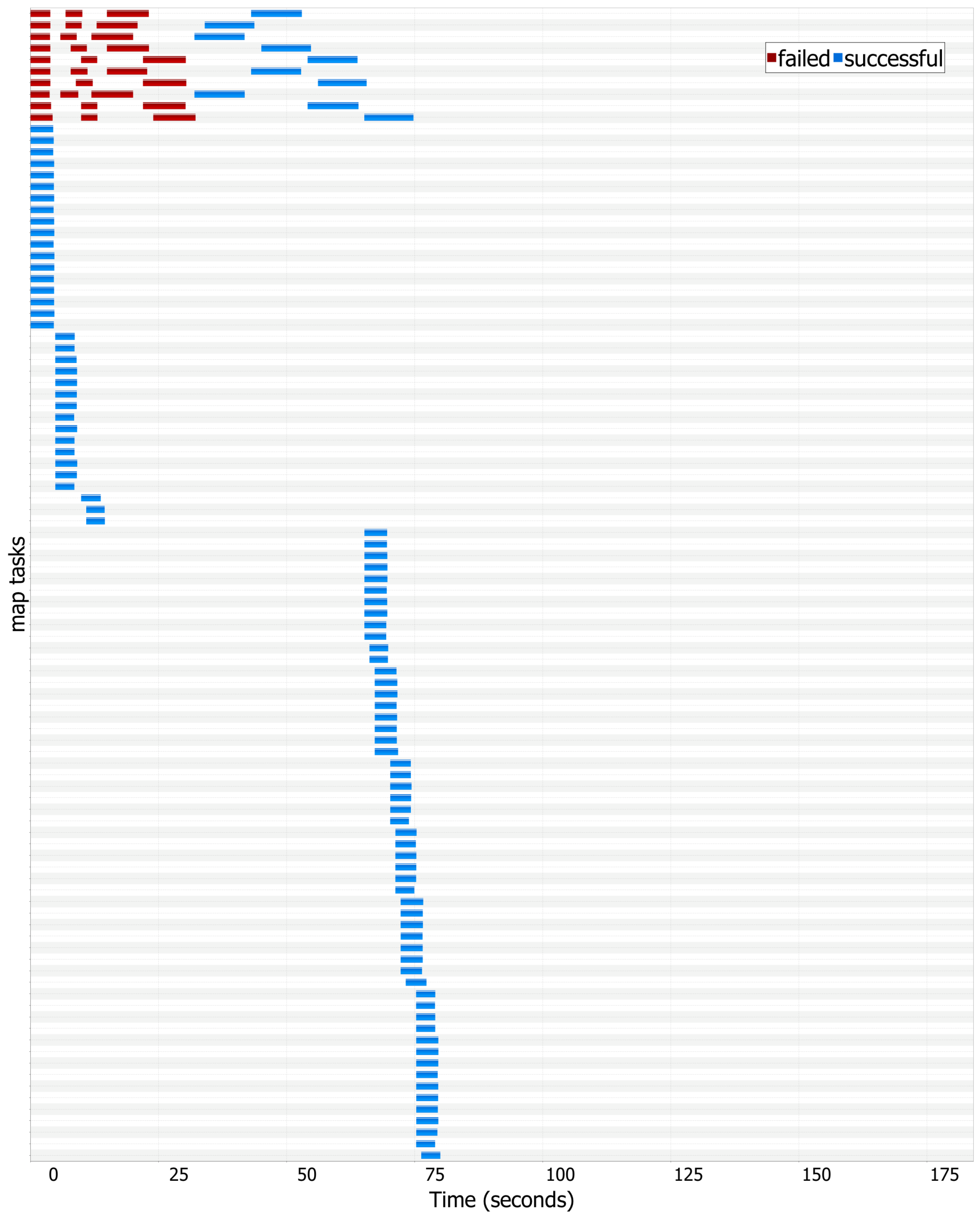


Figure 6.10: Number of tasks running in parallel for Fall-back approach

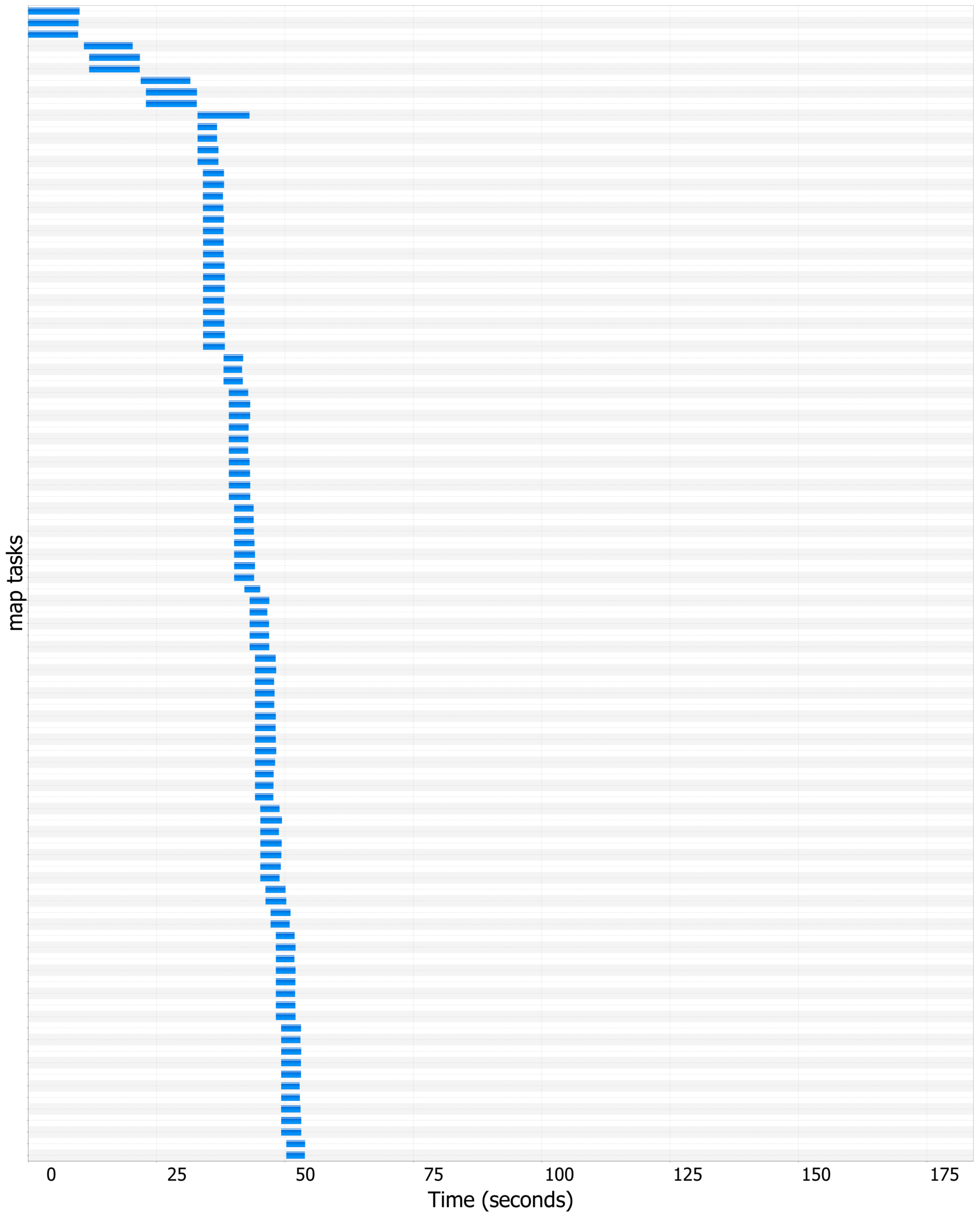


Figure 6.11: Number of tasks running in parallel for manual approach

around 70GB in size. The three Boa queries are: finding the 10 most used programming languages, the 5 largest projects in terms of AST (abstract syntax tree) nodes, and finding Varargs (variable arguments) usage over time.

Table 6.9 shows the time taken by the Boa query to find the 10 most used programming languages. This query is one of the simplest queries in Boa. The query runs successfully for the Default approach and because of the overhead are slower for Fall-back and Manual. Again however note the difference is less than 1 second.

Table 6.9: Boa query 1: finding the 10 most used programming languages

Experiment	Default	Fall-back	Manual
Run 1	23s	24s	24s
Run 2	23s	24s	24s
Run 3	23s	24s	24s
Run 4	23s	24s	24s
Run 5	23s	24s	24s
Run 6	23s	24s	23s
Run 7	23s	24s	23s
Run 8	23s	24s	24s
Run 9	23s	24s	24s
Run 10	23s	24s	23s
Average	23s	24s (-4.4%)	23.7s (-3%)

Table 6.10: Boa query 2: the 5 largest projects in terms of AST nodes

Experiment	Default	Fall-back	Manual
Run 1	17m17s	16m22s	14m32s
Run 2	17m19s	16m45s	14m34s
Run 3	17m17s	16m22s	14m43s
Run 4	17m17s	16m42s	14m31s
Run 5	17m10s	16m40s	14m40s
Run 6	17m19s	16m37s	14m40s
Run 7	17m13s	16m42s	14m43s
Run 8	17m10s	16m41s	14m29s
Run 9	17m14s	16m42s	14m45s
Run 10	17m17s	16m42s	14m39s
Average	17m16s	16m38s (+3.66%)	14m38s (+15.24%)

Table 6.10 shows the run-time of the Boa query to know the 5 largest projects in terms of AST nodes. The results show that both of our approaches are faster compared to the Default approach. Figure 6.12, Figure 6.13, and Figure 6.14 visualize the map tasks for all the approaches.

Notice the lengths of the bars appear longer in the Fall-back and Manual approaches, as the time taken for map tasks was longer in these approaches. This was due to most maps having half the memory, compared to the Default approach. These tasks were incurring GC overhead, while in the Default version the GC never triggered. In the Manual approach, the tasks that were manually allocated more memory were also slower compared to the Default approach (but faster than the Fall-back approach). This is possibly because of the disk buffer or other caching, which we were unable to identify and calculate. Despite these anomalies, since the Fall-back and Manual approaches were able to run more tasks in parallel they had better overall performance compared to the Default approach.

Table 6.11: Boa query 3: Varargs usage over time

Experiment	Default	Fall-back	Manual
Run 1	50m47s	*Failed	46m25s
Run 2	51m25s	*Failed	46m38s
Run 3	50m46s	46m49s	46m24s
Run 4	50m28s	*Failed	46m30s
Run 5	50m54s	*Failed	47m10s
Run 6	50m31s	46m47s	46m34s
Run 7	50m55s	47m38s	46m25s
Run 8	50m43s	47m38s	46m30s
Run 9	51m00s	*Failed	46m34s
Run 10	50m59s	*Failed	46m32s
Average	50m51s	46m58s (+7.6)%	46m34s (+8.4%)

Table 6.11 shows the run-time for the Boa query calculating the usage of Variable arguments. The Default approach failed for Java heap memory errors and timed-out exceptions for the first run, so we modified the configurations for the Default approach to have 2GB map memory. Our Fall-back approach also failed for timed-out exceptions. As our approach only deals with `OutOfMemoryException`, it can't handle the timed-out exceptions for map tasks and the job failed frequently. For some cases where it was not timing out, it showed better performance over

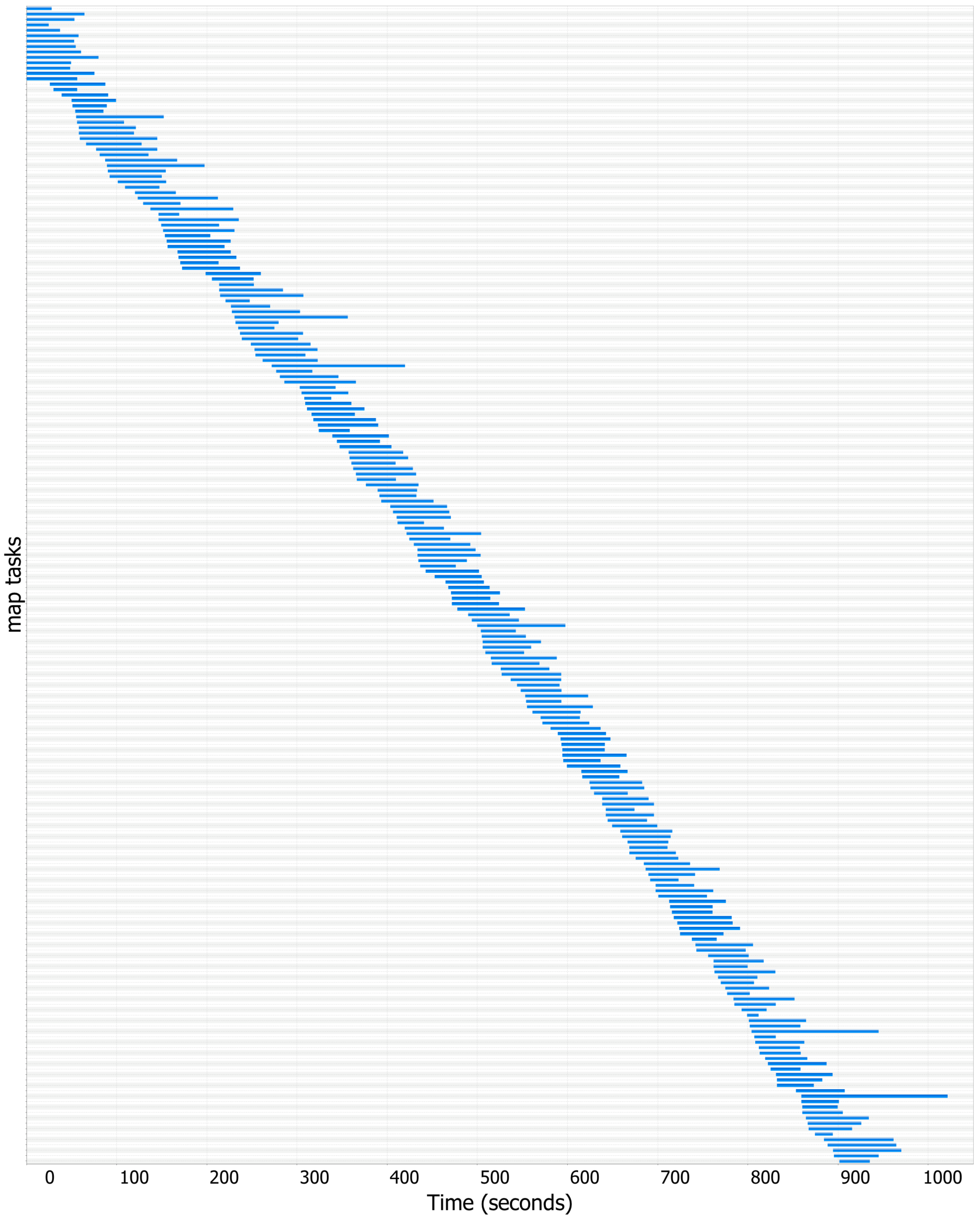


Figure 6.12: Number of tasks running in parallel for Boa's AstCount using default approach

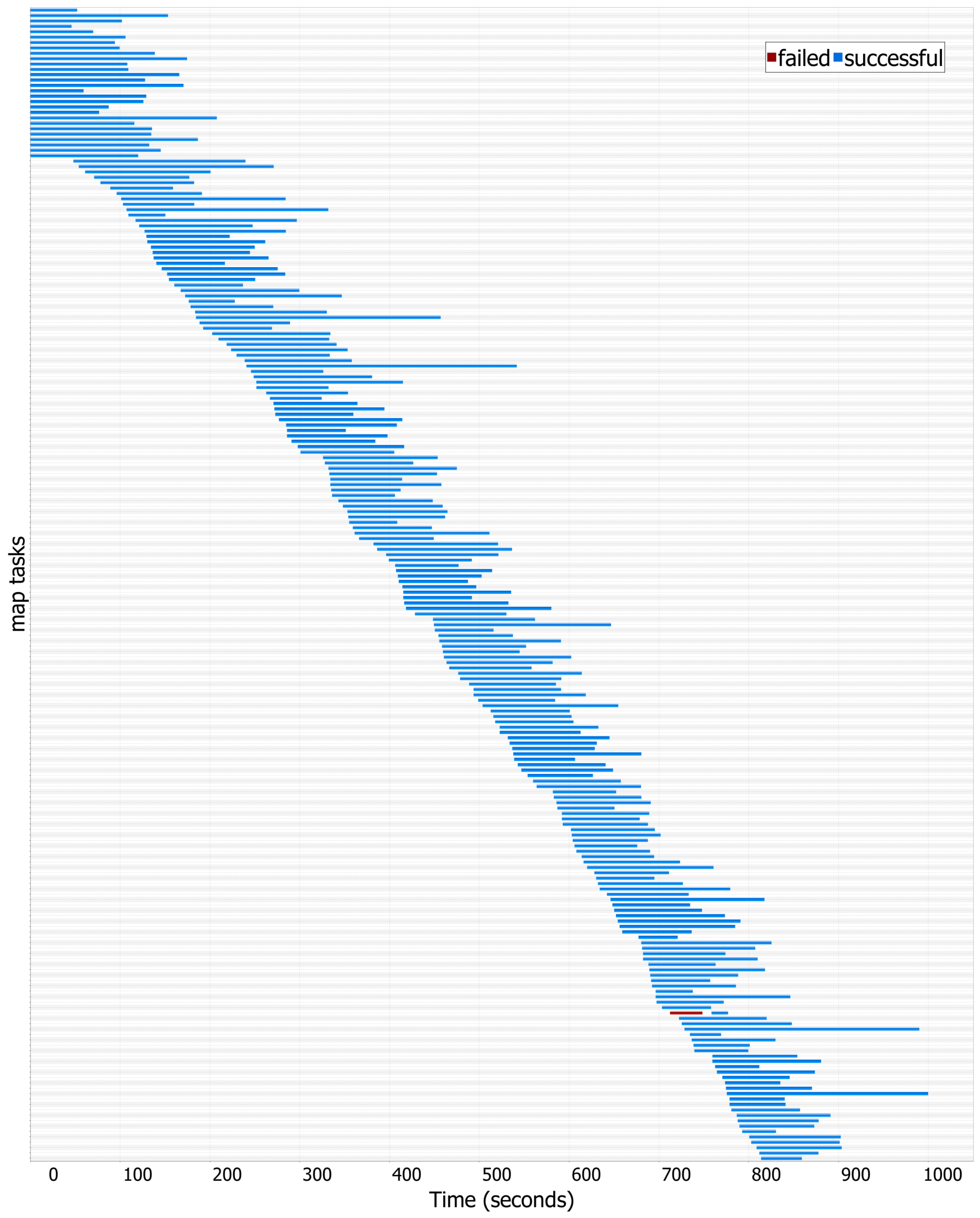


Figure 6.13: Number of parallel tasks running for Boa's AstCount using fall-back approach

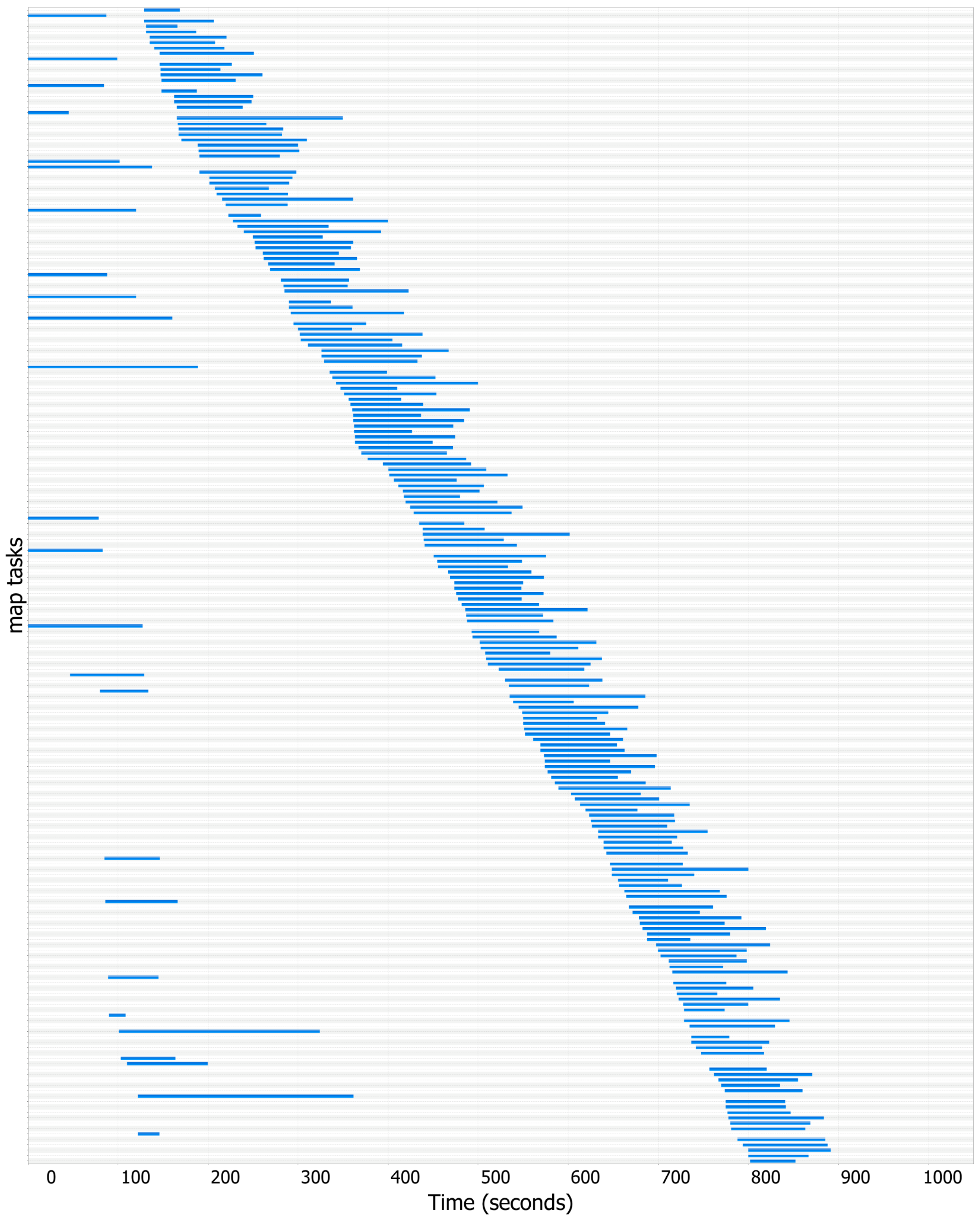


Figure 6.14: Number of parallel tasks running for Boa’s AstCount using manual approach

the Default approach and recovered from any Java heap size errors. Our Manual approach for this Boa query ran faster compared to the Default approach.

6.5.2 Processing Wikipedia's Data Set

Wikipedia [11] is one of the largest online encyclopedias. Every article is of different size and is revised frequently. This provides a very asymmetrical data set where some articles can contain a lot of data and some articles are very small. We took Wikipedia's page dump [12] for pages with their page history and revision details. The Wikipedia page dump we used was 5.7GB (this is a subset of the full dump). We ran a query to calculate the count of number of revisions made per page. Table 6.12 shows details about the time taken by all the approaches to count the number of revisions per page. Both of our approaches show significant performance improvement compared to the Default approach.

Table 6.12: Time taken to count the number of revisions per page

Experiment	Default	Fall-back	Manual
Run 1	1012s	811s	728s
Run 2	1018s	808s	725s
Run 3	1010s	810s	724s
Run 4	1014s	811s	720s
Run 5	1010s	811s	720s
Run 6	1012s	809s	723s
Run 7	1010s	811s	720s
Run 8	1012s	808s	720s
Run 9	1012s	809s	720s
Run 10	1012s	811s	722.3s
Average	1012.2s	809.9s (+19.98%)	722.3s (+28.64%)

Figure 6.15, Figure 6.16 and Figure 6.17 visualize the map tasks for the Wikipedia data set. In our approaches, more tasks run in parallel which significantly improves the overall processing time. The failed tasks in the Fall-back approach run on a larger container while re-attempting and subsequent re-attempts run successfully. The tasks manually specified are picked and processed early. As more number of tasks run in parallel in both of our approaches overall processing time is improved significantly.

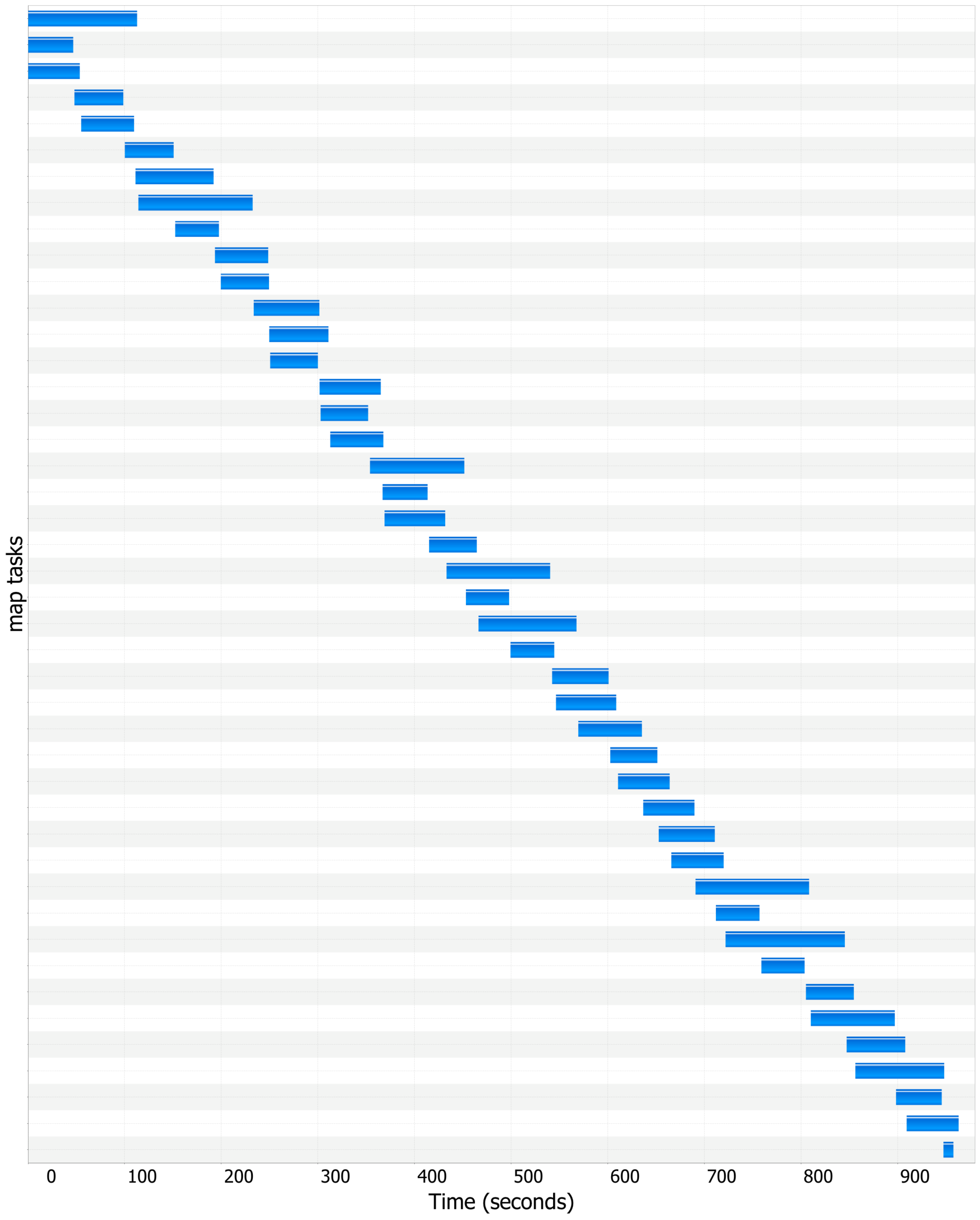


Figure 6.15: Number of parallel tasks running for Wikipedia pages using default approach

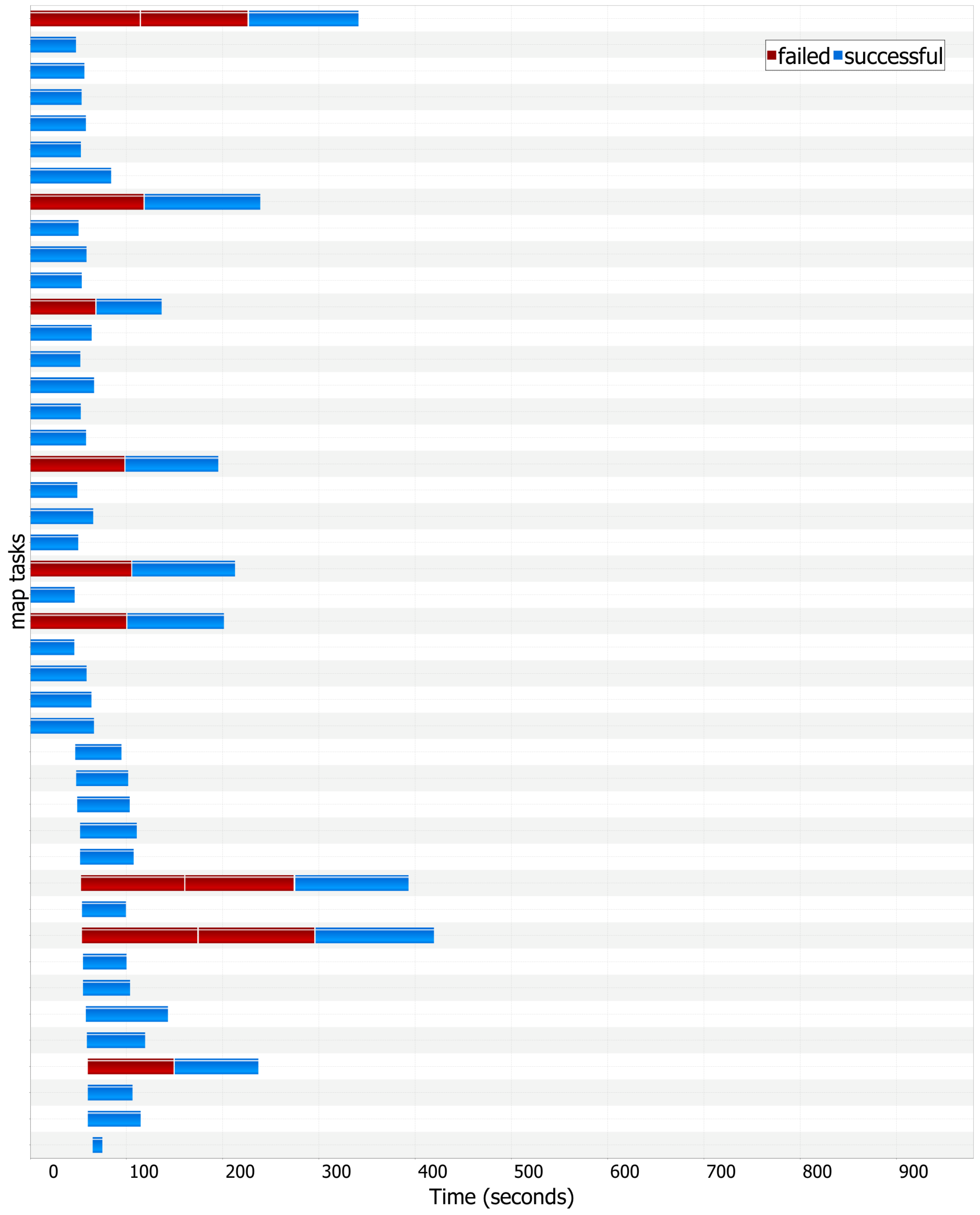


Figure 6.16: Number of parallel tasks running for Wikipedia pages using fall-back approach

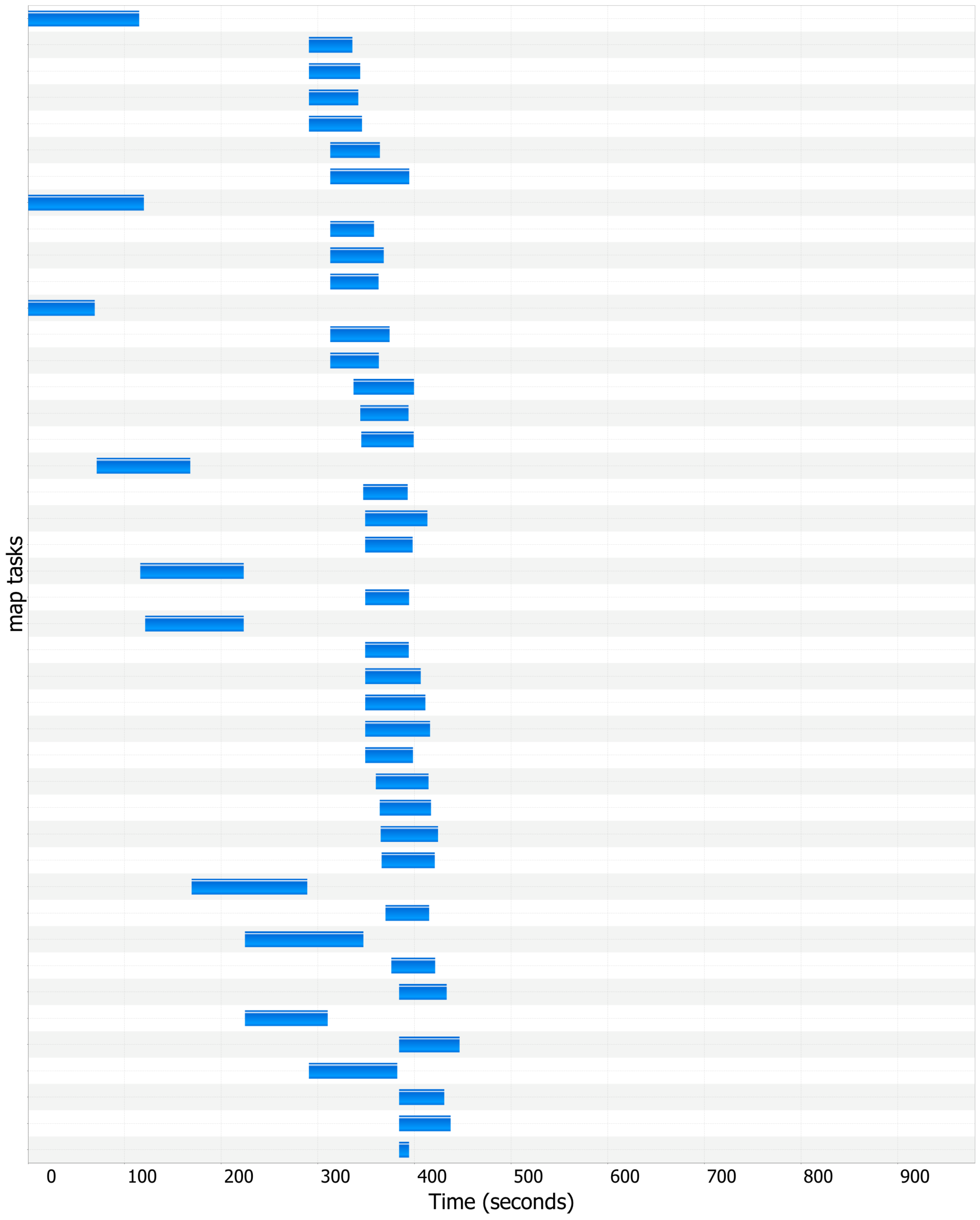


Figure 6.17: Number of parallel tasks running for Wikipedia pages using manual approach

6.6 RQ5: Do Our Approaches Affect Data-Locality?

Data-locality is the process of moving computation to where the data is allocated and is one of the more important features of Hadoop. A lot of IO time is saved if we process data close to the place where it is stored in a cluster instead of transferring over network. The ResourceManager tries to allocate the map tasks near the data it is going to process. The ResourceManager schedules containers by node-aware container allocation, rack-aware container allocation and other local map allocation scheme for every map task. Since we have not modified the ResourceManager's container scheduling process on different nodes, we expect similar data-locality as the Default approach.

Even in the Fall-back and Manual approaches, the first time a task attempt runs the scheduler tries to schedule it data local. For the Fall-back version, any failed attempt again uses the default scheduling strategy which relaxes data locality on task retries. Thus we expected to see more rack-local allocations for Fall-back. But for both approaches, since we are increasing the parallelism in the system it may lead to difficulties with data-local scheduling.

To evaluate if we impact the default data-locality feature of Hadoop, we used a 9 node cluster consisting of a master node and 8 slave nodes. The data set is stored in HDFS [7] with a default replication factor of 3. This allows the scheduler to consider data locality when scheduling. The experiments were on data sets comprising of 10% and 50% large files with small file size as 100MB and large file size as 2GB. We recorded the time taken to process the data by all the approaches for 10 successful runs.

Table 6.13 shows that for 10% large data set the Default approach allocates 99-100 tasks as data-local, The Fall-back approach allocates 96-99 tasks data-local and Manual allocates 96-100 tasks data-local. Similarly, Table 6.14 shows 97-99 data-local tasks for Default, 96-98 data-local tasks for Fall-back and 96-99 data-local tasks for Manual. Both Table 6.13 and Table 6.14 show approximately similar behavior: data-locality is affected by our approaches, but only slightly. For Fall-back this is worse because each retried attempt is allowed to be rack-local. For Manual, since

Table 6.13: Data-local and rack-local allocation with 2GB, 10% large files

	Default			Fall-back			Manual		
Run	Data local	Rack local	Runtime	Data local	Rack local	Runtime	Data local	Rack local	Runtime
Run1	100	0	71s	96	4	45s	97	3	30s
Run2	99	1	72s	99	1	44s	98	2	29s
Run3	99	1	71s	99	1	43s	100	0	29s
Run4	100	0	71s	97	3	44s	96	4	29s
Run5	100	0	71s	98	2	44s	96	4	30s
Run6	100	0	72s	96	4	44s	98	2	30s
Run7	99	1	72s	98	2	43s	97	3	31s
Run8	99	1	71s	96	4	43s	96	4	30s
Run9	100	0	72s	99	1	43s	100	0	30s
Run10	100	0	71s	98	2	44s	97	3	29s

Table 6.14: Data-local and rack-local allocation with 2GB, 50% large files

	Default			Fall-back			Manual		
Run	Data local	Rack local	Runtime	Data local	Rack local	Runtime	Data local	Rack local	Runtime
Run1	98	2	93s	96	4	121s	96	4	81s
Run2	98	2	92s	97	3	119s	96	4	82s
Run3	98	2	93s	98	2	121s	97	3	83s
Run4	98	2	93s	97	3	119s	98	2	80s
Run5	98	2	92s	98	2	121s	98	2	81s
Run6	97	3	92s	96	4	121s	96	4	80s
Run7	99	1	92s	98	2	118s	99	1	80s
Run8	97	3	92s	96	4	121s	98	2	80s
Run9	98	2	92s	96	4	119s	99	1	80s
Run10	98	1	92s	98	2	118s	97	3	80s

the parallelism was much higher (around 88 parallel tasks vs 9 in Default) it was more difficult for the scheduler to keep every task data-local.

6.7 Threats to Validity

In this section we discuss potential internal and external threats to the validity of our experiments.

6.7.1 Internal Threats

This study was tested on several different data sets. Our synthetic data sets are stored in HDFS and are sorted because of directory structure, however real life data sets need not always be sorted. This is not a cause of concern, as we have tested our applications on two real-world data sets as well, which are not sorted. Our approaches can suffer construct validity if the number of tasks running at a given time are strictly fixed as this will reduce the maximum number of tasks which can run in parallel. For Section 6.2 we need to purposefully select the homogeneously distributed data set as we need to calculate the overhead incurred by all the versions of Hadoop without any re-attempts or failures. As we mostly used a four node cluster for the testing, our approaches may suffer threats if ran on very large clusters consisting of many nodes. Also there are many configuration parameters which Hadoop offers and could affect the results. In cases when speculative tasks come into play our approach can suffer because speculative tasks can consume some of the available resources and affect the performance.

6.7.2 External Threats

External threats to our experiments can be the user defined mapper and reducer functions, which can take very long time to process the task. Another possible external threat can be data-locality of the tasks. Data-locality varies with every run of the application. All approaches can suffer slow processing of data because of non-data local task allocations. Also open source projects can have many other external threats too. The generalization of our approaches can't be made as there can be other possible external factors, parameters or flaws in our approach which we were not able to test, but the central idea of scheduling more tasks in parallel by allocating higher resources to only those tasks which requires higher resources will hold. Other external threats can be the user's inefficient allocation of the map memory parameter as it can also affect the performance in Manual and Default approaches.

CHAPTER 7. FUTURE WORK

Our modified version of Hadoop MapReduce postulates allocation of different map memory for different records. In the future, we plan to extend this approach to reducer tasks as well. We are also planning to focus on automatically inferring the container sizes based on both the input data set and the specific user query. In this chapter we discus these approaches for future work in more detail.

7.1 Allocating Exactly the Amount of Resources a Task Needs

Currently, in the Fall-back approach we allocate double the memory resources for retries of failed task attempts. It is very probable that tasks may require less resources than double of the previous failed attempt. For example if a task fails for a 2GB container, we allocate it a 4GB container for the new attempt. It is possible the task might only require a 2300MB container and thus we could allocate a 3GB container instead.

In the future we may increase resources linearly rather than doubling the resource for every retry. This has the potential of optimizing the allocated resources and leaving room for more containers. The drawback of such an approach is it may take more retries. This optimization is currently possible in the Manual approach, as every map task can be allocated exactly the amount of resources it needs.

7.2 Implementing for Reducers

The proposed approaches all focus solely on container resource allocation for map tasks. In the future we plan to also examine how these approaches could be extended to support memory issues that occur inside reducer tasks.

7.3 Accounting for User Queries During Automatic Inference

The proposed approaches only consider the size of the input data set when automatically inferring the container allocations. We plan to extend this approach to infer the container allocations not only based on input data sets, but also on the specific user queries. For example, in Boa [3] certain user queries actually load significant amounts of additional data from another file on HDFS. If simple queries that do not load this data run first, our current approach may automatically infer smaller resource allocations that fit these simple queries. If a larger query runs later, it would not have sufficient resources and need to use the Fall-back approach. In practice, queries in Boa are a mix of these two and thus any choice the automated approach infers would most likely be wrong for many queries. Thus in the future, we plan to investigate an automated approach that examines the actual Hadoop code and determines how to configure the container resources accordingly.

CHAPTER 8. CONCLUSION

The MapReduce programming model has seen widespread adoption for processing of big data. This model as implemented in Apache Hadoop typically assumes the individual records to be processed are of similar size. Several real-world data sets, such as project data on SourceForge or the page dumps from Wikipedia, are asymmetric and fail this assumption. The state-of-the-art solution to handling such asymmetric data sets is to determine the memory resources required for the largest individual input record and then allocate all map tasks with those resources. This leads to few number of map tasks running in parallel, thus under-utilizing the cluster resources and potentially leading to longer overall job times.

In this work, we addressed this issue of maintaining parallelism of map tasks in Hadoop when processing such asymmetric data sets. We proposed, implemented, and evaluated three approaches: Fall-back, Manual, and automatically inferred. The first approach tries running a map task and if it has memory issues, retries the task with double the memory resources. This approach requires no configuration of Hadoop. The second approach allows users to manually specify which tasks need more memory. This manual approach allows for tuning the memory requirements of a job to exactly what it needs. The third approach learns from prior runs which tasks failed for memory and automatically configures them to use more memory in the future.

We tested these approaches on a wide variety of data sets including real-life data from SourceForge and Wikipedia. We investigated a series of challenging research questions about the overhead incurred, performance improvements, and improved parallelism of the approaches via a sequence of experiments on Hadoop clusters on CloudLab [31, 32]. Both of the implemented approaches have negligible overhead (< 1 second) and have shown improvements in overall processing time of the asymmetric data sets.

The Manual approach was fastest when compared to the Fall-back and Default approaches for data sets containing 10–50% large files. The Fall-back approach was faster than the Default approach when the data sets contain 10–30% large files. The Fall-back approach doesn't require user effort to configure Hadoop. Thus, more parallelization in the system can be achieved if we allocate higher resources to the map tasks processing larger records as compared to allocation of higher resources to all the map tasks.

In the future we plan to consider user queries when automatically inferring container sizes. We also plan to extend our approach to support reducers and use a more optimal fallback approach.

BIBLIOGRAPHY

[1] Apache Software Foundation, "Welcome to Apache Hadoop!" http://hadoop.apache.org/, 2016.

[2] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE. IEEE Press, 2013, pp. 422–431.

[4] C. V. Lopes and J. Ossher, "How scale affects structure in Java programs," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. SPLASH. ACM, 2015, pp. 675–694.

[5] Apache Software Foundation, "MapReduce Tutorial," https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html, 2016.

[6] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013, pp. 5:1–5:16. [Online]. Available: http://doi.acm.org.ezproxy.bgsu.edu:8080/10.1145/2523616.2523633

[7] J. Zaharia, S. Rixner, and A. L. Cox, "The Hadoop distributed filesystem: Balancing portability and performance," in *2010 IEEE International Symposium on Performance Analysis of Systems & Software*, ser. ISPASS. IEEE, 2010, pp. 122–133.

[8] "CapacityScheduler Guide," https://hadoop.apache.org/docs/r1.2.1/capacity_scheduler.html, 2013.

[9] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3. ACM, 2007, pp. 59–72.

[10] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, pp. 10–10, 2010.

[11] "Wikipedia," https://en.wikipedia.org/, 2015.

[12] Wikimedia Foundation, Inc., "Wikipedia: Database download," https://en.wikipedia.org/wiki/Wikipedia:Database_download, 2016.

[13] "SourceForge.Net," https://www.sourceforge.net/, 2016.

[14] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Job scheduling for multi-user mapreduce clusters," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-55*, 2009.

[15] J. Polo, C. Castillo, D. Carrera, Y. Becerra, I. Whalley, M. Steinder, J. Torres, and E. Ayguadé, "Resource-aware adaptive scheduling for MapReduce clusters," in *Middleware 2011*. Springer, 2011, pp. 187–207.

[16] L. Cheng, Q. Zhang, and R. Boutaba, "Mitigating the negative impact of preemption on heterogeneous MapReduce workloads," in *Proceedings of the 7th International Conference on Network and Services Management*, ser. CNSM '11. Laxenburg, Austria, Austria: International Federation for Information Processing, 2011, pp. 189–197. [Online]. Available: http://dl.acm.org.ezproxy.bgsu.edu:8080/citation.cfm?id=2147671.2147701

[17] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using Mantri," in *Proceedings of the 9th*

USENIX Conference on Operating Systems Design and Implementation, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 265–278. [Online]. Available: http://dl.acm.org.ezproxy.bgsu.edu:8080/citation.cfm?id=1924943.1924962

[18] Q. Chen, C. Liu, and Z. Xiao, "Improving MapReduce performance using smart speculative execution strategy," *IEEE Transactions on Computers*, vol. 63, no. 4, pp. 954–967, 2014.

[19] V. P. and T. U. Karthik, "Mapreduce scheduler using classifiers for heterogeneous workloads," *IJCSNS*, vol. 11, no. 4, p. 68, 2011.

[20] S. Li, S. Hu, S. Wang, L. Su, T. Abdelzaher, I. Gupta, and R. Pace, "WOHA: deadline-aware map-reduce workflow scheduling framework over hadoop clusters," in *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*. IEEE, 2014, pp. 93–103.

[21] M. Li, L. Zeng, S. Meng, J. Tan, L. Zhang, A. R. Butt, and N. Fuller, "MRONLINE: MapReduce Online Performance Tuning," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '14. New York, NY, USA: ACM, 2014, pp. 165–176. [Online]. Available: http://doi.acm.org/10.1145/2600212.2600229

[22] J. Polo, D. Carrera, Y. Becerra, J. Torres, E. Ayguadé, M. Steinder, and I. Whalley, "Performance-driven task co-scheduling for MapReduce environments," in *Network Operations and Management Symposium (NOMS), 2010 IEEE*. IEEE, 2010, pp. 373–380.

[23] X. Ding, Y. Liu, and D. Qian, "JellyFish: Online Performance Tuning with Adaptive Configuration and Elastic Container in Hadoop YARN," in *Parallel and Distributed Systems (ICPADS), 2015 IEEE 21st International Conference on*. IEEE, 2015, pp. 831–836.

[24] N. Khoussainova, M. Balazinska, and D. Suciu, "Perfxplain: debugging mapreduce job performance," *Proceedings of the VLDB Endowment*, vol. 5, no. 7, pp. 598–609, 2012.

[25] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: Efficient iterative data processing on large clusters," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 285–296, Sep. 2010. [Online]. Available: http://dx.doi.org/10.14778/1920841.1920881

[26] G. Wang, A. R. Butt, P. Pandey, and K. Gupta, "A simulation approach to evaluating design decisions in mapreduce setups." in *MASCOTS*, vol. 9. Citeseer, 2009, pp. 1–11.

[27] H. Herodotou and S. Babu, "Profiling, what-if analysis, and cost-based optimization of mapreduce programs," *Proceedings of the VLDB Endowment*, vol. 4, no. 11, pp. 1111–1122, 2011.

[28] P. Lama and X. Zhou, "Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud," in *Proceedings of the 9th international conference on Autonomic computing*. ACM, 2012, pp. 63–72.

[29] A. Verma, L. Cherkasova, and R. H. Campbell, "ARIA: Automatic Resource Inference and Allocation for MapReduce Environments," in *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ser. ICAC '11. New York, NY, USA: ACM, 2011, pp. 235–244. [Online]. Available: http://doi.acm.org/10.1145/1998582.1998637

[30] G. Liao, K. Datta, and T. L. Willke, "Gunther: Search-based auto-tuning of MapReduce," in *European Conference on Parallel Processing*. Springer, 2013, pp. 406–419.

[31] R. Ricci, E. Eide, and The CloudLab Team, "Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications," *USENIX ;login:*, vol. 39, no. 6, Dec. 2014.

[32] The University of Utah, "Cloudlab," https://www.cloudlab.us/, 2016.