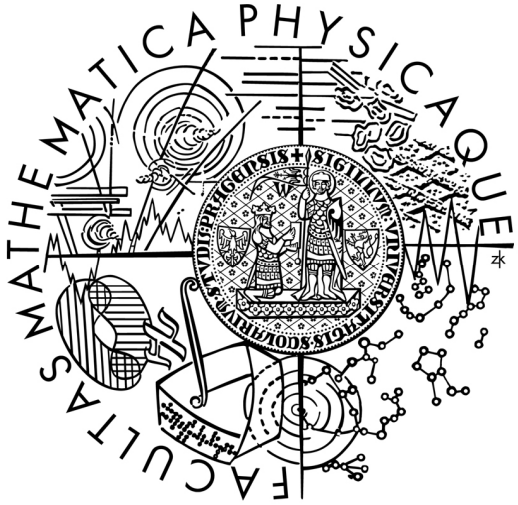
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Kateřina Opočenská

Dynamic Setup for Clusters with Multi-Master Architecture

Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Jakub Yaghob, Ph.D.
Studijní program: Informatika, Softwarové systémy

Ráda bych na tomto místě poděkovala svému vedoucímu, panu RNDr. Jakubu Yaghobovi, PhD. za možnost vypsání tohoto tématu jako diplomové práce na MFF a za jeho odborné vedení, především za cenné rady týkající se samotného textu.
Dále bych chtěla poděkovat skupině PH-SFT z CERNu za nabídku tohoto zajímavého tématu a za poskytnutou podporu v průběhu jeho zpracovávání. Jmenovitě děkuji svému konzultantovi z CERNu, panu Gerradu Ganisovi, za nápady a pomoc v průběhu vývoje a za veškerou praktickou podporu při implementaci a nasazování algoritmu do praxe. V neposlední řadě děkuji také paní Anně Robertové za korekturu angličtiny.

At this place, I would like to thank to my supervisor Mr. Jakub Yaghob, PhD. for the possibility of presenting this project as a master thesis at the Faculty of Mathematics and Physics, Charles University in Prague, and for the professional guidance, concerning specifically the thesis text itself.
Further, I would like to thank the CERN PH-SFT group for the offer of this interesting topic, and for the great support during the time I was working on it. Especially, my thanks to my CERN's consultant, Mr. Gerrardo Ganis, for the ideas and help during the development and for all technical support during the implementation and deployment of the algorithm.
Last but not least, I would like to thank Mrs. Anna Roberts for helping me to improve the English of this text.

Prohlašuji, že jsem svou diplomovou práci napsala samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 10. prosince 2009 Kateřina Opočenská

Table of contents

1. Introduction 6
1.1. Background 6
1.2. The PROOF system 7
1.3. Project motivation and main goals 8
1.4. Thesis structure description 9

PART I

2. Master-worker paradigm 10
2.1. Single master 10
2.2. Advantages and limitations 10
2.3. Hierarchical master-worker paradigm 12
2.4. Improving performance with hierarchical master-worker 13
3. General multi-master setup problem 14
3.1. Single master placement 14
3.1.1. Application work stages 15
3.1.2. System constraints 15
3.1.3. Additional constraints for SlaveRate 16
3.1.4. Transformation to maximum flow problem 17
3.1.5. Selecting the master (algorithm) 17
3.1.6. Environment and complexity 18
3.1.7. Conclusion 19
3.2. Multiple masters placement 20
3.2.1. Model variables 20
3.2.2. B-COVER problem formulation 22
3.2.3. B-COVER complexity: NP-hard 24
3.2.4. Conclusion 24

PART II

4. The PROOF system overview 25
4.1. ROOT framework 25
4.2. PROOF design goals 25
4.3. PROOF multi-tier master-worker architecture 26
4.4. Packetizer - load-balancing engine of PROOF 27
4.5. Merging outputs 28
4.6. Scheduling in PROOF 29
5. HEP data analysis with PROOF 30
5.1. Typical use-case 30
5.2. TSelector query interface 30
5.3. Simple sample PROOF session 32
6. PROOF query processing on single master configuration 35
6.1. Typical PROOF cluster 35
6.2. Task execution phases for single master 35
6.2.1. Initialization 36
6.2.2. Computation 36
6.2.3. Finalization 37
6.3. Resource utilization diagram 38
6.4. Idle periods during computation 38
6.5. Summary 39

7. PROOF query processing using more masters 41
7.1. One level multi-master configuration 41
7.2. Task execution phases for more masters 42
7.2.1. Top-master initialization 42
7.2.2. Initialization of one sub-master 43
7.2.3. Initialization of s sub-masters 43
7.2.4. Computation 44
7.2.5. Finalization of one sub-master 44
7.2.6. Finalization of s sub-masters 44
7.2.7. Top-master finalization 44
7.3. Execution time summary 44
7.4. How computation can be speeded up by more masters 45
7.5. Optimal number of masters for finalization 47
7.5.1. Another view on parallel finalization of more masters 48
7.5.2. Speed-up of parallel finalization 49
8. In search for multi-master setup algorithm 52
8.1. Using state-of-art knowledge 52
8.2. Record-based MMS algorithm 53
8.2.1. Algorithm's quick overview 53
8.2.2. Changing conditions 55
8.2.3. Detailed description of record-based MMS algorithm 55
8.2.4. Optimistic estimation formula 61
8.2.5. Pilot implementation learning 61
9. Merger-based multi-master setup algorithm 63
9.1. Optimal configuration for computation 63
9.2. Optimal configuration for finalization 64
9.3. Detailed description of merger-based algorithm 65
9.4. Correctness and finiteness 69
9.4.1. Algorithm is finite 70
9.4.2. Algorithm is correct 71
9.5. Supporting algorithm's robustness 71
9.5.1. Confirmation of merger's start-up success 73
9.5.2. Confirmation of successful merging from merger to worker 75
9.5.3. Confirmation of successful merging from worker to master 76
9.5.4. Summary 77
10. Benchmarks of merger-based algorithm 79
10.1. Measurement methodology 79
10.1.1. Test environment 79
10.1.2. Test approach 79
10.1.3. Test data 80
10.2. Benchmarks of queries using standard objects 81
10.3. Benchmarks of queries using optimized objects 82
11. Conclusion 84
11.1. Future work 84

References 86

Appendix A – Benchmarks of merger-based algorithm 87
Appendix B – Content of enclosed DVD 90

Název práce: Dynamic Setup for Clusters with Multi-Master Architecture
Autor: Kateřina Opočenská
e-mail autora: opocenska@gmail.com
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Jakub Yaghob, Ph.D.
e-mail vedoucího: jakub.yaghob@mff.cuni.cz

Abstrakt: Diplomová práce se zabývá problémem multi-master konfigurace pro počítačové clustery, na kterých běží systém PROOF. PROOF je framework postavený na master-worker architektuře, který se používá především na analýzu fyzikálních dat v CERNu (Evropská organizace pro jaderný výzkum). Cílem práce je určit optimální počet masterů, při jejichž použití je daná úloha zpracována v nejkratším čase. Na základě analýzy průběhu zpracování úlohy je představen a naimplementován tzv. merger-based algoritmus, tedy algoritmus založený na konceptu mergera. Merger je uzel, který se během výpočtu chová jako worker, ale během poslední, a nezřídka velmi náročné fáze slučování mezivýsledků plní roli mastera. Počet a přesné určení mergerů probíhá dynamicky během zpracování úlohy a je určeno jak velikostí clusteru, tak i jeho aktuálním výkonem. Na závěr práce je provedeno srovnání merger-based algoritmu s dosavadním klasickým přístupem, a to na různých úlohách a pro různé velikosti clusteru. Naměřené zrychlení je srovnáno s teoretickými hodnotami.

Klíčová slova: master-worker paradigma, multi-master konfigurace, ROOT/PROOF, merger

Title: Dynamic Setup for Clusters with Multi-Master Architecture
Author: Kateřina Opočenská
Author’s e-mail address: opocenska@gmail.com
Department: Department of Software Engineering
Supervisor: RNDr. Jakub Yaghob, Ph.D.
Supervisor's e-mail address: jakub.yaghob@mff.cuni.cz

Abstract: The work deals with the problem of multi-master setup for clusters running PROOF, which is a master-worker based framework used at CERN (European Organization for Nuclear Research), preferably for analysis of high energy physics data. The goal is to determine an optimal number of masters for the given task in order to make the task execution time as short as possible. Based on the analysis of PROOF processing work-flow, the merger-based algorithm is designed and implemented. It introduces a concept of the merger, which is a node acting as a worker during the computation phase, and as a master during the final phase of sub-results merging. The number and selection of merger nodes is performed dynamically, and depends both on the cluster size and it’s recent performance. The performance of the merger-based algorithm is compared to the standard approach on several queries and several sizes of the cluster. The measured speed-up is confronted with the previously invented theory.

Keywords: master-worker paradigm, multi-master configuration, ROOT/PROOF, merger

1. Introduction

1.1. Background

Today's high-energy physics (HEP) experiments produce extremely large amounts of data, which need to be stored and further processed. Following the recent start-up of the LHC (Large Hadron Collider), the world's biggest particle accelerator at CERN[1], Switzerland, physicists are getting ready for arrival of peta bytes of new data. They hope that its analysis could help them to answer the most important questions of today's particle physics, such as the origin of the matter, and thus of our Universe.

The LHC will generate 40 million proton-proton collisions per second at the center of each of its four main experiments (ALICE, ATLAS, CMS, LHCb). However, not all the collisions are interesting from the LHC physics program and the challenging task of specialized data acquisition systems, located very close to the detectors, is to reduce the huge collision rate to a manageable rate of O(100 Hz) including all interesting collisions. The output of the acquisition system is made of raw detector signals which are not directly usable for physics analysis. The raw data are therefore reconstructed, i.e., transformed into physical properties such as energies, charges, tracks etc., which can be by end-user physicists.

It is expected to record 100 – 200 'interesting' collisions per second. This turns into the registration of 10^{10} collisions per year, which means up to 15 peta bytes (15 million GB) annually. If we wrote all that data to CD's and put them one on another, we would get a stack that was about 20 kilometers high.

Expecting to have that huge amount of data coming every year, we need to think carefully about effective ways of its storage and processing. Obviously, only CERN's computing capacity would never be enough. However, the Worldwide LHC Computing Grid (WLCG) [1] project is hoping to solve this problem by exploiting various computing resources around the globe. The WLCG is a global collaboration of 33 countries, involving more than 140 computing centers world-wide. The one in the Czech Republic is located at Institute of Physics, Academy of Sciences of the Czech Republic in Prague. The goal of the WLCG is to create and maintain *data storage* and *computing infrastructure* for the data coming from the LHC experiments, and enabling the access to this data to thousands of involved physicists regardless of their physical location.

After the initial processing and back-up of the incoming data mainly on tapes at CERN (*Tier-0*), it is then distributed to about 10 primary locations around the world referred

[1] European Organization for Nuclear Research

as *Tier-1* centers. Tier-1's make the data available to around 120 *Tier-2* centers for specific analysis tasks. Individual physicists connect to Tier-2's from their home institutions (*Tier-3's*) to perform the analysis they demand.

To do this, they need powerful, yet not too complicated tools, allowing them to process the large amounts of the data in reasonable time. The only way to achieve this is to exploit the inherent parallelism of HEP data (the collisions are independent) and hence process in parallel different portions of the data samples.

The traditional way how to approach it is to use *batch systems* built on a *push architecture* meaning that the tasks are divided into several sub-tasks in advance. These sub-jobs are then run in parallel and in the end, their sub-results are merged. The main advantage is that there is no need for the program modification, i.e. the same user code can run locally as well as on a batch system. However, the length of the entire execution is limited by the execution time of the slowest sub-task, which leads to a significant prolongation in the case of an under-performing node or sequential submitting of some sub-tasks. Other weak points of the traditional batch systems are e.g., the real-time feedback and exploiting of multi-cores machines. All these disadvantages are supposed to be overcome by the *pull-architecture* based system introduced in the following sub-chapter.

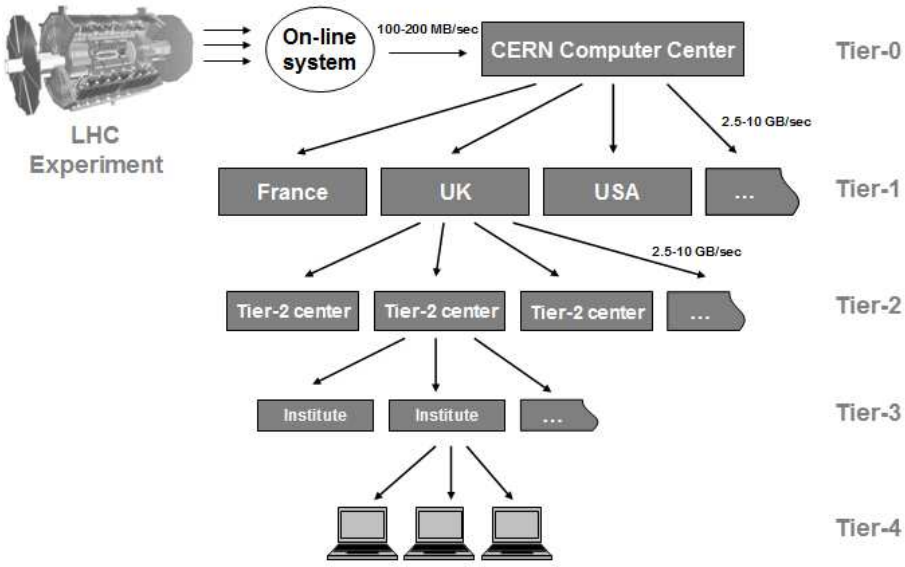


Figure 1 - Multi-tiered view of the WLCG
(Image source: CERN)

1.2. The PROOF system

As the ROOT software system [2] is heavily used as an open-source framework for HEP data analysis at CERN since the mid-1990s, it was natural to think of its extension from single-core computers to multi-core machines and computer clusters.

The project called Parallel ROOT Facility (PROOF) [3], [4] started in 1997 as a ROOT extension and a joint effort between CERN and the Massachusetts Institute of Technology. It is meant as an alternative to batch systems for central analysis facilities and departmental workgroups running Tier-2's and Tier-3's.

PROOF builds on the well known master-worker parallel computing paradigm. The master node distributes the work to a set of workers using a *pull architecture,* i.e. workers ask for a new sub-task when they have finished the previous one. In the end, the master also automatically merges their sub-results. PROOF can also use a multi-master setup where a set of statically defined sub-masters are each in charge of controlling the work of a given sub-set of workers. The top-master node then distributes and merges the work of these sub-masters.

One of main experiments, which has already adopted PROOF is ALICE [5]. It aims to use PROOF preferably for prompt analysis of proton-proton collisions data and pilot analysis of heavy ions (Pb-Pb first) collision data. The goal of its PROOF cluster CAF [6] is to have around 500 CPU's together with around 100 TB of locally selected data at the disposal of ALICE users. This cluster does not aim to replace the Grid for analysis but to provide fast access to significant data samples so that the development cycle of physics analyses is speeded up.

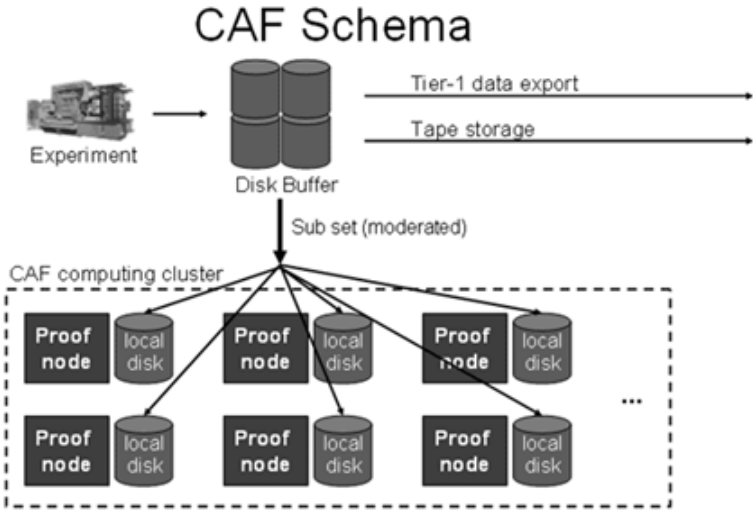


Figure 2 – Selecting data from ALICE experiment for storage on the CAF computing cluster
(Image source: [6])

1.3. Project motivation and main goals

The main goal of this thesis is to develop and implement an algorithm solving the *multi-master setup (MMS) problem* for PROOF. By the *multi-master setup* or *configuration,* we mean the master and worker roles' assignment among cluster nodes, which follows the classic or the hierarchical master-worker paradigm (details in Chapter 2).

The *problem of multi-master setup* is then the problem of assigning master and worker roles to individual nodes in order to minimize the total execution time of the given task. In other words, should more masters be used instead of the single master on a given cluster for a given task. If so, their count, organization, and exact location should be determined.

Currently, only static assignment of nodes via configuration files is supported; so the same cluster setup is used for all types of incoming tasks. In some cases, this may lead to a non-optimal performance as the analysis type could range from more data-bound to more CPU-bound. The system should be able to decide dynamically which nodes of the PROOF cluster should act as sub-masters if any and which ones as workers.

1.4. Thesis structure description

The work is divided into two main parts. In Part I, we focus on the master-worker paradigm and master setup problem in general; while in Part II we concentrate only on PROOF and its special features. This specification of initial conditions will also shift the core of our problem, as shown later.

Part I comprises Chapter 2 and Chapter 3. Chapter 2 gives the reader a basic understanding of the master-worker paradigm and its variation, called hierarchical master-worker. In Chapter 3, we focus on both single and multi-master setup problem in heterogeneous environments; and we present recent knowledge on this topic.

Part II starts with Chapter 4 devoted to detailed PROOF description followed by Chapter 5, focusing on PROOF from end-user's point of view. In Chapter 6, we analyze PROOF query processing and its individual phases when the single master configuration is used. Chapter 7 continues the topic, as its focus is mainly concerned on the processing of a query on a multi-master configuration. Both these chapters serve as sources of facts, features, observations and computations later used for the design of a MMS algorithm. First, we introduce the record-based algorithm in Chapter 8 and we also discuss its pilot implementation learning. In Chapter 9, we present the merged-based MMS algorithm, which was later successfully implemented and tested on the Alice CAF cluster at CERN. Some of its performance statistics and benchmarks are presented in Chapter 10. In the last segment, Chapter 11, we summarize the accomplishments of this project and discuss its possible future development.

PART I

2. Master-worker paradigm

In this chapter, we focus on the master-worker paradigm in general. We discuss and evaluate its usability, and both advantages and limitations. Moreover, we show how its scalability can be enhanced by deployment of more masters; and we provide an example where this approach has already helped.

The intent of this chapter is to offer to a reader a more broaden perspective before starting the description of the PROOF system itself in Chapter 4. This general introduction will also help us to distinguish more easily between the features coming from the paradigm itself and features which are linked specifically to PROOF.

2.1. Single master

In (single) master-worker or master-slave based applications, all the nodes have a role of a *worker* except a single node, which is called the *master*. In the simplest form, the master node starts the task, distributes the work to its workers, collects back their sub-results, and creates the final result by merging these sub-results. Accordingly, each worker node accepts the work assigned by the master, processes it and sends back its sub-result.
For some long-running tasks, it is more convenient and also safer to use a finer granularity when distributing the work. In such a case, the master gradually sends pieces of the task to its workers and can also gradually collect their sub-results. However, this also puts more load on the master node itself, as it runs both the work distribution and sub-results merging at once. An alternative is to distribute the work gradually in smaller pieces, collecting all sub-results at once in the end.

In some master-worker models, the master node can also perform direct computations, while in other models it only distributes and collects the work. Both variations can be converted to each other easily. If the master cannot perform any direct computation, we can simulate it by adding one more process - the worker process - on the same node. On the other hand, master's ability to compute can be usually suppressed by setting the appropriate computing rate of the master to zero.

2.2. Advantages and limitations

The master-worker paradigm fits perfectly to the processing of naturally parallel HEP data, as well as to many other tasks being parallelized today. Some other problems commonly

solved under the master-worker paradigm are Monte Carlo simulations [7], genetic algorithms [8], and N-body simulations [9]. These tasks are all characterized by performing the same operation on all *independent* pieces of input; in our case, these are the independent collision events. The result is that the same code can run on all the worker nodes, just the input differs for each of them.

Moreover, communication takes place just between workers and the master. There are neither communication nor synchronization requirements among workers so they can then process their parts independently and at their own processing rates. This could be especially useful in heterogeneous environments like computational grids. The available geographically distributed resources are usually of various powers, and the network latency is significantly higher. Thus the communication can become a limiting factor of the overall performance more easily than on a local homogeneous cluster. However, the requirement of the minimal communication is usually in the opposite with the demand of effective scheduling, as advanced scheduling strategies are more communicatively exhaustive than a simple work distribution.

Another feature that makes the master-worker approach suitable for still more popular grid computing is its failure tolerance. If a worker fails during the computation, its work can be dynamically reassigned to another worker; and it is the only part that needs to be recomputed due to the sub-tasks independence. On the other hand, if the master fails, the computation process must be restarted from the beginning. We say that the master creates the single point of failure in the master-worker applications.

Master-worker paradigm simplified summary	
Advantages	**Disadvantages**
Simple for design and implementation	Single point of failure
Large scale of applicability	Limited scalability
Weak communication and synchronization requirements	
Failure tolerant	

Table 1- Master-worker paradigm summary

Moreover, the master can easily become a bottle-neck in a whole system if there are too many workers which need to be served. In the case of too many workers trying to communicate with the single master, there is a high probability of some congestion. The incoming messages

from workers are not processed immediately then, but instead queued on the master. As a consequence, some workers may become idle during the computation phase as they do not get their input task immediately. It is always possible to find such number of workers, which is simply unmanageable for the single master. Therefore, we say that the master-worker applications have limited scalability.

Another related issue is the final merging phase on the master. The merging is not parallelized and could make a significant part to the whole execution time.

A natural solution to the above mentioned performance degradation problems is deploying of more masters. Each of them manages just a part of available workers; and therefore, later merges just a part of their partial results.

2.3. Hierarchical master-worker paradigm

We refer to the *hierarchical master worker paradigm* as a variation of the master-worker paradigm with two or more levels of masters. On the top level, there is the single master called the *top-master, coordinator* or *supervisor*. It distributes the incoming task to another level of masters, which are called *sub-masters*. Sub-masters either distribute the work to another level of sub-masters or directly to their workers. In other words, only sub-masters on the last level communicate directly with workers.

Again, the whole configuration follows a tree pattern, where the root of the tree is the top-master, leaves are workers, and the inner nodes are sub-masters. In fact, the same *distribute-and-merge* work pattern is applied on each level. The average size of the distributed work (sub-task) is expected to vary on each level. In order to keep all the workers busy, the top-master distributes greater parts of the work to its sub-masters than these sub-masters do to their workers.

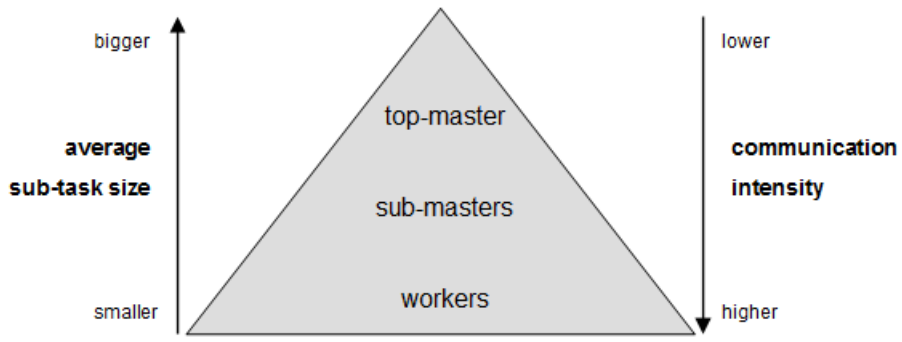


Figure 3 - Average sub-task size and communication intensity in the hierarchical master worker paradigm

As a consequence, the communication intensity is higher on lower levels, which could be taken advantage of especially in heterogeneous environments. The sub-master and all its

workers could be put on some tightly coupled computing resources as they are expected to communicate more frequently. The overall performance may then become better even if the single master has never been an obvious performance bottle-neck before.

Comparison of general features	
Single master	**More masters**
As many nodes as possible devoted to real computing	Some nodes at least partially devoted to management work
Limited scalability	Improved scalability
Simple work distribution	More work distribution phases
Communication channels determined by locations of workers	Smart positioning of sub-masters can improve communication channels in heterogeneous environments

Table 2 - Comparison of general features of the single and hierarchical master-worker paradigm

2.4. Improving performance with hierarchical master-worker

In [10], the authors discussed the impact of the hierarchical master-worker paradigm on the performance of an application, which solves the *BMI Eigenvalue Problem* by a parallel branch and bound algorithm. BMI Eigenvalue Problem is an optimization problem of minimizing the greatest eigenvalue of a bilinear matrix function. To solve the BMI Eigenvalue Problem, they proposed an algorithm, which is based on the hierarchical master-worker paradigm. They made a comparison of its performance (on a grid test-bed) with the performance of the previously used conventional master-worker based algorithms:

> *"The results showed that computation with the conventional master-worker paradigm is not suitable to efficiently solve the optimization problem with fine grain tasks on the WAN setting because communication overhead is too high compared to the cost of tasks. The hierarchical master-worker paradigm avoids performance degradation caused by high communication overhead by putting frequent communication between a master process and worker processes in tightly coupled computing resources. It also eliminates a performance bottleneck on a master process and improves performance scalability by distributing work among multiple master processes."*

3. General multi-master setup problem

In this chapter, we focus on the *general multi-master setup problem*, i.e., which nodes to choose as master(s) in heterogeneous environment in order to maximize the number of processed tasks per time unit. It is expected that not only computational resources, but also communication channels, are allowed to have different characteristics; therefore, the determination of the optimal master(s) location(s) can be a very complex task. However, it still has a high importance, as different configurations can considerably affect the overall execution time of processed tasks.

First, we present the work of *Shao, Berman, Wolski: Master/slave Computing on the Grid* [11], where the authors addressed the resource selection problem within the steady-state master-worker scheduling framework, i.e., how to determine a performance-efficient placement of master and slave processes running in shared, distributed, and heterogeneous environments. Then we follow up with the work of *Banino: Optimizing Locationing of Multiple Masters for Master-Worker Grid Applications* [12], which showed that extending of this problem to finding locations for more masters also significantly leverages its complexity. It is important to note that both works did not directly address the dynamically changing nature of large-scale computing platforms. However, they claimed that a dynamic context may be often viewed as a succession of static contexts.

3.1. Single master placement

The work-rate-based model for the master-worker application performance proposed in [11] builds on the network connectivity graph, where nodes represent processors, i.e., worker or master nodes and edges represent network links among these processors (details in Chapter 3.1.6). The goal is to determine the master processor *m* and the set of slave processors $s \in S$, so that the application's *work rate* (definition follows) is maximal.

In the following figure, we have a simple example with processors A, B, C, D connected through networks Net1, Net2 and Net3.

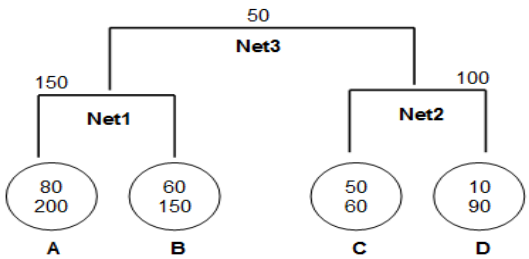


Figure 4 - Processors A, B, C and D connected through networks Net1, Net2 and Net3
(Image source: [11])

3.1.1. Application work stages

The *application's work* is defined as a set of divisible tasks, which are each completed by progressing through the following 4 stages:

1) *Transmission of a command* to initiate a task on one of the slave processors, including all the necessary data.
2) *Execution of the task* on the selected slave.
3) *Transmission of results* from the slave back to the master.
4) *Immediate processing of task results* from the slave that must be done by the master.

Considering Figure 4, if processor A is chosen as the master, a task intended for slave processor C during Stage 1 will employ the use of networks Net1, Net2 and Net3 to transfer required data from processor A to processor C. During Stage 2, the task will utilize processor time on C to run task computations. During Stage 3, the task will again utilize networks Net1, Net2 and Net3 to transfer results from C to A. Finally, during Stage 4, the task will utilize processor time on A to process the incoming results and to prepare for initiating of additional task transfers to C.

3.1.2. System constraints

Each system resource, i.e., a processor or a network, is limited by a constraint determining how many tasks it can process in a time unit. In the case of network *n*, $W_{Net}(n)$ determines how many of tasks it can transfer in a time unit. For processor *i*, we distinguish two different processing rates: master work-rate $W_{MasterCPU}(i)$ and slave work-rate $W_{SlaveCPU}(i)$. Formally:

- $W_{MasterCPU}(i)$ is the maximum master work rate of processor *i*. This is determined by processor *i's* capacity to perform Stage 4 computations for a specified application.
- $W_{SlaveCPU}(i)$ *is* the maximum slave work rate of processor *i*. This is determined by processor *i's* capacity to perform Stage 2 computations for a specified application.
- $W_{Net}(n)$ *is* the maximum communication rate of network *n*. This is determined by network *n's* capacity to perform Stage 1 and Stage 3 communication for a specified application.

In Figure 4, label by an edge represents W_{Net}, upper number by a processor represents $W_{SlaveCPU}$ and lower number by a processor represents $W_{MasterCPU}$.

Further we define:

- *SlaveRate(m, s)* is the task completion rate (in tasks per unit of time) occurring between master *m* and slave *s*.

Apparently, *SlaveRate(m, s)* is determined by the above mentioned system resource constraints as it involves the transfer and computation of the task as well as the back transfer of the result and its final processing on the master.

We can express the total rate of task completions for a master-worker application as the sum of task completions by individual workers because their processing is independent.

- $AppRate(m,S) = \sum_{s \in S} SlaveRate(m,s)$

To find the execution time for an application, we need to know total number of tasks *T*, which this application comprises. Then we can determine the execution time for master *m* and set of slave processors *S* in the following way:

- $ExecTime(m,S) = \frac{T}{AppRate(m,S)}$

3.1.3. Additional constraints for SlaveRate

Application's performance can be deduced from values for *SlaveRate(m, s).* In order to find out these values, the following constraints for system resources must be met:

1) $SlaveRate(m,i) \leq W_{SlaveCPU}(i)$

2) $\sum_{i \in S} SlaveRate(m,i) \leq W_{MasterCPU}(m)$

3) $\sum_{i \in ShareNet(G,S,m,n)} SlaveRate(m,i) \leq W_{Net}(n)$

Auxiliary function *ShareNet(G, S, m, n)* used in constraint 3 takes as input network connectivity graph *G*, set of slaves processes *S*, master process *m*, and network resource *n*. Its output is the set of slave processes from *S*, which share the use of *n* when communicating with *m*.

Simply said, the above mentioned constraints (1) – (3) reflect natural limits of the system. If some resources are used by more entities at once (network links, master node), then the performance of these resources is shared by these entities. Obviously, the goal is to find such *SlaveRate* values which meet these constraints and also yield the largest value of *AppRate(m, S).* The solution then corresponds to the configuration, which delivers the best achievable application's performance.

3.1.4. Transformation to maximum flow problem

We can convert the problem of determining *SlaveRate* values to the maximum flow problem where:

- Slave processes from set *S* are sources for flows and *m* is the sink for all flows.
- The flow constraints correspond to the $W_{MasterCPU}(i)$, $W_{SlaveCPU}(i)$, and $W_{Net}(n)$ capacities.
- The *SlaveRate(m, s)* values are the individual flows we wish to find.

The problem can be solved by using some of the well known max-flow algorithms. The basic idea is to run the algorithm for several *m* candidate processes and choose the one allowing the maximal flow in the system. It is obvious that the processor with the greatest $W_{MasterCPU}(i)$ does not have to necessarily allow the greatest application's performance. For example, considering the processors in Figure 4, host A has the biggest $W_{MasterCPU}(i)$; however, the best performance (the highest *AppRate*) is reached when processor B is chosen as the master (Table 3).

Master location m	$W_{MasterCPU}$ (m)	SlaveRate (m, A)	SlaveRate (m, B)	SlaveRate (m, C)	SlaveRate (m, D)	AppRate (m)
A	200	0	60	50	0	110
B	**150**	**80**	**0**	**50**	**0**	**130**
C	60	50	0	0	10	60
D	90	40	0	50	0	90

Table 3 – Application's work rate depends on the master location

3.1.5. Selecting the master (algorithm)

The algorithm presented in [11] is based on the well-known Ford-Fulkerson algorithm. The estimated flow rate for each master candidate is kept augmenting by adding the contributions of slave processors. First, the most effective nearby slaves are added, i.e., those with the highest $W_{SlaveCPU}(i)$, reachable within the same local network as the master candidate. Then the most effective slaves from other networks follow. Resource limits have to be checked all the time.

The entire master selection algorithm in the original form is provided below:

```
// Preparation
For all networks k
      Calculate maximum network capacity WNet(k)
For all processors j
      Calculate maximum master processor capacity WMasterCPU(j)
      Calculate maximum slave processor capacity WSlaveCPU(j)
```

```
For each candidate master processor p on local network n
{     Set sum for candidate slave work rates CandRate(p) = 0
      Set found set Found(p) to empty
      For all networks k
            Set network utilization sum NetUtil(k) = 0
      Get maximum capacity WNet(n) of local network n
      Get maximum master processor capacity WMasterCPU(p)

      // Add suitable processors from the same local network
      While CandRate(p) < WNet(n) and CandRate(p) < WMasterCPU(p)
      {     Select new processor s from same local network as p with the
                  largest available WSlaveCPU(s) value
            Get fraction F of WSlaveCPU(s) that will not cause utilization
            NetUtil(n) to exceed WNet(n)
                  Add F to CandRate(p)
                  Add F to NetUtil(n)
                  Add processor s to found set Found(p)
      }
      Total candidate work rate CandRate(p)= min(CandRate(p),WMasterCPU(p))
      Total local network utilization NetUtil(n) = CandRate(p)

      // Add suitable processors from other local networks
      While CandRate(p) < WNet(n) and CandRate(p) < WMasterCPU(p)
      {     Select new processor q from outside local network with
                  the largest available WSlaveCPU(q) value
            Get fraction F of WSlaveCPU(q) that will not cause
            utilization NetUtil(i) to exceed WNet(i) for any network i
                  Add F to CandRate(p)
                  Add F to NetUtil(n)
                  Add F to other NetUtil(k) where network k is involved in
                  communications between processors p and q
                  Add processor q to found set Found(p)
      }
}
Select processor p with largest CandRate(p) as master
Select processors from Found as its slaves
```

3.1.6. Environment and complexity

In [11] there is also presented a way how to obtain the input parameters for the model and how to derive a logical view of resource interconnection by using a logical network configuration discovery tool called Effective Network Views (ENV) [13]. The output of the ENV tool is a simplified network graph representation where the entire system can be viewed

as several sets of processors connected by local networks. Each of these local networks is then connected to other local networks by, at most, one level of remote networking as depicted in Figure 5.

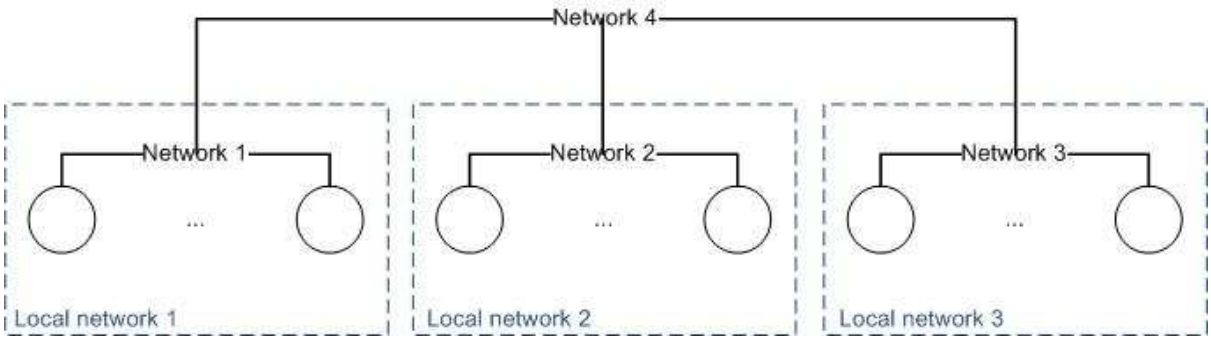


Figure 5 – Simplified logical graph representation produced by ENV

Therefore:

- Data transfers between nodes on the same local network pass through only one level of networking ("1 edge") and encounter only one network resource constraint.
- Data transfers between nodes in different local networks pass through three levels of networking ("3 edges") and encounter three networking constraints.
- All slave work rates must meet the resource constraints of the master processor.

At the most, four constraint tests must be then checked for each master-slave pair. Having *n* nodes in the system, there is *n * (n-1)* possible master-slave pairs. As the work needed for one pair is limited by the constant, the total algorithm complexity is $O(n^2)$.

3.1.7. Conclusion

It was shown that the master selection problem can be transformed to a maximal flow problem in a graph with a special simplified topology (Figure 5). Because of this topology, the problem of finding the master and the set of slaves with the highest *AppRate* can be solved by a max-flow-based algorithm in $O(n^2)$.

If we want to place more than one master and find a set of efficient slaves for each master, the problem becomes substantially more complex. In the system of *n* nodes, there are *n* possible locations for one master. However, if we want to place *s* masters in the *n* node system, there are $\binom{n}{s}$ possibilities, which makes the approach of an exhaustive trial of all pairs rather impractical. In the next chapter, we present [12] where the author showed that such a problem is NP hard.

3.2. Multiple masters placement

Cyril Banino: Optimizing Locationing of Multiple Masters for Master-Worker Grid Applications [12] introduced a cost model for establishing and operating of more masters on a platform with heterogeneous environment. The system is expected to deal with a large number of equal-sized *application tasks*. These application tasks are modeled as requiring some input data file of size β_I and producing some output data file of size β_O. Input files (tasks) are generated, and output files (results) are collected on master nodes.

The problem is to select such set of masters that it maximizes the *steady-state throughput of the platform*, i.e., the total number of all application tasks processed by all workers within one time-unit. All processors are expected to operate under the *full overlap, single-port* mode, which allows them to perform the following actions simultaneously:

- Receiving data from at most one of its neighbors.
- Performing some independent computation.
- Sending data to at most one of its neighbors.

This means that the master nodes do not only distribute the work and collect results, but they can also perform some of their own computations. The question is how much of the work to compute on themselves and how much to distribute to other nodes.

After the start-up phase, all the resources are expected to operate in a *periodic mode*. It means that for each node we can determine fraction of time spent on receiving, computation, and sending during one time-unit of the steady-state regime. This allows for computation of the steady state throughput of the entire platform.

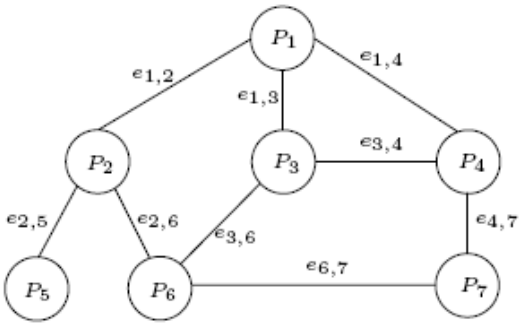


Figure 6 - Grid graph: Vertices represent processors, edges represent communication links
(Image source: [12])

3.2.1. Model variables

The target architectural framework is represented by graph $G = (V,E)$ as illustrated in Figure 6. Vertex $P_i \in V$ represents a computing resource of weight w_i $(w_i > 0)$, meaning

that processor P_i requires w_i units of time to process one task. In other words, the higher w_i, the slower the processor is.

Edge $e_{i,j}$: $P_i \rightarrow P_j$ represents a communicating resource having a bandwidth equal to $\gamma_{i,j}$, which limits the amount of the data that can be transferred on link $e_{i,j}$ per time unit in both directions. Further, we denote:

- $c_{i,j}$ – number of time units needed to transfer one *input task* from processor P_i to neighbor processor P_j $(c_{i,j} > 0)$[1]
- $c'_{i,j}$ – number of time units needed to transfer one *output task* from processor P_i to neighbor processor P_j $(c'_{i,j} > 0)$
- $J_m \subseteq V$ - the index set of the master candidate's processors

  $\forall i \in J_m$: $x_i \in \{0, 1\}$ the decision variable to place a master at location P_i, i.e., $x_i = 1$ if P_i is chosen as a master, and $x_i = 0$ otherwise
- f_i - the fixed cost of establishing a master at location P_i $(f_i > 0)$
- t_i - the per task cost of operating a master at location P_i $(t_i > 0)$
- $n(i)$ – the index set of the neighbors of processor P_i
- m_i - maximum number of input tasks that P_i can communicate to its neighbors per time unit. This is restricted by the inverse of the smallest communication time $c_{i,j}$ of the neighbors of P_i. Hence $m_i = \frac{1}{\min\{c_{i,j} \mid j \in n(i)\}}$
- m'_i - maximum number of output results that P_i can receive from its neighbors per time unit. $m'_i = \frac{1}{\min\{c'_{j,i} \mid j \in n(i)\}}$
- g_i - number of input tasks generated by P_i per time unit. g_i is limited by number of tasks P_i can process per time unit, i.e., by $\frac{1}{w_i} + m_i$
- g'_i - number of output files collected by P_i per time unit

During one time unit:

- α_i - fraction of time spent by P_i on computing
- $s_{i,j}$ - fraction of time spent by P_i on sending input tasks to its neighbor $P_j \in n(i)$
- $s'_{i,j}$ - fraction of time spent by P_i on sending output results its neighbor $P_j \in n(i)$
- $r_{i,j}$ - fraction of time spent by P_i on receiving input tasks from its neighbor $P_j \in n(i)$
- $r'_{i,j}$ - fraction of time spent by P_i on receiving output results from its neighbor $P_j \in n(i)$

[1] We do not expect by default the communication times $c_{i,j}$ and $c_{j,i}$ (similarly $c'_{i,j}$ and $c'_{j,i}$) to be equal, due to say, different I/O hardware device of processors P_i and P_j

3.2.2. B-COVER problem formulation

We call as *B-COVER problem* the problem of selecting a master locations set that optimizes the throughput of the platform within budget constraint *B*. Mathematical formulation of the B-COVER problem can be stated by the following integer linear program, whose objective is to maximize the throughput $n_{task}(G)$ of the platform graph *G*. The objective function is the number of tasks computed within one unit of time, i.e., the platform throughput.

Maximize: $$n_{task}(G) = \sum_{i \in V} \frac{\alpha_i}{w_i}$$

Subject to the following 12 equations:

(1) $\forall i : 0 \le \alpha_i \le 1$

(2) $\forall i, \forall j \in n(i) : 0 \le s_{i,j} \le 1$

(3) $\forall i, \forall j \in n(i) : 0 \le s'_{i,j} \le 1$

(4) $\forall i, \forall j \in n(i) : 0 \le r_{i,j} \le 1$

(5) $\forall i, \forall j \in n(i) : 0 \le r'_{i,j} \le 1$

Equations (1) – (5) express that all the activity variables (α_i, $s_{i,j}$, $s'_{i,j}$, $r_{i,j}$, $r'_{i,j}$) are fractions of one time unit, i.e., belonging to interval [0, 1].

(6) $\forall i, \forall j \in n(i) : s_{i,j} = r_{j,i}$

(7) $\forall i, \forall j \in n(i) : s'_{i,j} = r'_{j,i}$

Equations (6) and (7) ensure communication consistency: The time spent by P_i on sending input tasks (output results) to P_j equals to the time spent by P_j on receiving this input (output) from P_i.

(8) $\forall i \sum_{j \in n(i)} (s_{i,j} + s'_{i,j}) \le 1$

(9) $\forall i \sum_{j \in n(i)} (r_{i,j} + r'_{i,j}) \le 1$

Equation (8) and (9) ensure that send and receive operations to neighbors of P_i are sequential.

(10) $\forall i \in J_m : x_i + \sum_{j \in n(i)} s'_{i,j} \le 1$

Equation (10) enforces that masters ($x_i = 1$) do not receive input files from other processors.

$$(11)\ \forall i \in J_m : x_i + \sum_{j \in n(i)} r_{i,j} \leq 1$$

Equation (11) enforces that masters ($x_i = 1$) do not send output results to other processors.

$$(12)\ \forall e_{i,j} \in E : \left(\frac{s_{i,j}}{c_{i,j}} + \frac{r_{i,j}}{c_{j,i}}\right) * \beta_i + \left(\frac{s'_{i,j}}{c'_{i,j}} + \frac{r'_{i,j}}{c'_{j,i}}\right) * \beta_O \leq \gamma_{i,j}$$

Equation (12) ensures that link bandwidths cannot be exceeded. This constraint is due to our hypothesis that the same link $e_{i,j}$ may be used in both directions simultaneously.

$$(13)\ \forall i \notin J_m : g_i = 0$$

$$(14)\ \forall i \in J_m : 0 \leq g_i \leq \left(\frac{1}{w_i} + m_i\right) * x_i$$

Equations (13) and (14) say that only masters (P_i, where $i \in J_m$ and $x_i = 1$) can generate input tasks, i.e., to have $g_i > 0$.

$$(15)\ \forall i \notin J_m : g'_i = 0$$

$$(16)\ \forall i \in J_m : 0 \leq g'_i \leq \left(\frac{1}{w_i} + m'_i\right) * x_i$$

Equations (15) and (16) specify that only masters can collect generated output files, i.e., to have $g'_i > 0$.

$$(17)\ \forall i : g_i + \sum_{j \in n(i)} \frac{r_{i,j}}{c_{j,i}} = \frac{\alpha_i}{w_i} + \sum_{j \in n(i)} \frac{s_{i,j}}{c_{i,j}}$$

$$(18)\ \forall i : g'_i + \sum_{j \in n(i)} \frac{s'_{i,j}}{c'_{i,j}} = \frac{\alpha_i}{w_i} + \sum_{j \in n(i)} \frac{r'_{i,j}}{c'_{j,i}}$$

Equations (17) and (18) represent conservation laws: For every processor P_i, the number of input files generated, plus the number of input files received, equals to the number of input files processed plus the number of input files sent (17).

For every processor P_i, the number of output files collected, plus the number of output files sent, equals to the number of input files processed plus the number of output files received (18).

$$(19)\ \sum_{i \in J_m} (f_i * x_i + t_i * g_i) \leq B$$

Equation (19) ensures that the costs generated by establishing (f_i) and operating (t_i) the chosen master locations do not exceed the budget constraint B.

3.2.3. B-COVER complexity: NP-hard

Now we present the proof from [12] which shows that the task of determining master locations in the above described system is NP-hard. We build the proof on reducing the well-known MAXIMUM KNAPSACK (MK) problem [14] to the previously defined B-COVER problem.

Maximum knapsack

- **Instance:** Finite set U, for each $u \in U$ a size $s(u) \in Z+$ and a value $v(u) \in Z+$, a positive integer $B \in Z+$
- **Solution:** A subset $U' \subseteq U$ such that $\sum_{u \in U'} s(u) \leq B$
- **Measure:** Total weight of the chosen elements, i.e., $\sum_{u \in U'} v(u)$

We construct an instance of the B-COVER problem from the instance of MK in the following way: We create a set V containing processors P_i $(i = 1 \ldots |U|)$ and a bijective function $f : V \rightarrow U$.

$\forall P_i \in V$ we set: $w_i = \frac{1}{v(f(P_i))}$, $f_i = s(f(P_i))$ and $t_i = 0$

Further, we set $E = \varnothing$ and $J_m = V$.

The graph of the B-COVER instance has no edges, which means that tasks cannot be transferred among processors. Therefore, tasks can only be computed in the location where they are generated. A solution of the B-COVER instance consists in determining a subset $V' \subseteq V$ such that $\sum_{P_i \in V'} f_i \leq B$ in order to maximize the platform throughput, which is $\sum_{P_i \in V'} \frac{1}{w_i}$

It is straightforward that a solution of the B-COVER problem instance provides a direct solution of the MK instance. This proves that B-COVER is at least as difficult as MK. Since MK is known as NP-hard [14] and since the above mentioned transformation can be done in polynomial time, we can conclude that B-COVER is also NP-hard.

3.2.4. Conclusion

As this theoretical result is rather pessimistic, the author in [12] proposed a simple heuristic approach based on LP-relaxation, i.e., relaxing the integer constraints of an integer linear program. If we allow $\forall i \in J_m$, $0 \leq x_i \leq 1$, we obtain a linear program in rational numbers, which can be solved in polynomial time. The author claims that this approach has achieved very good performance on a wide range of simulations. However, its detailed description and specific results' presentation are beyond limits of this work.

PART II

4. The PROOF system overview

In this chapter, we introduce basic concepts of the PROOF system in terms of its master-worker architecture and design goals. We focus on how the load-balancing and scheduling is performed, which will be useful later when considering key features of the multi-master setup algorithm. Special attention is paid also to the last phase of task execution called the *finalization* or *merging*.

4.1. ROOT framework

The Parallel ROOT Facility, PROOF, is an extension to the ROOT system, a cross-platform object-oriented framework for HEP data analysis heavily used at CERN, Fermilab, and other nuclear physics laboratories around the world. It is also the preferred data analysis environment for all main LHC experiments.

ROOT consists of several parts dedicated to various purposes such as:

- Data processing (interactive/batch mode).
- Data analyzing (histograms, trees, advanced mathematical and statistical tools).
- Results visualization (explaining pads, 2D and 3D graphics, GUI editor).
- General and specialized simulations (virtual Monte Carlo, geometry packages).

A user interacts with ROOT via command line, GUI, or batch scripts. The primary command and scripting language for ROOT is C++; its embedded interpreter is called CINT [15]. ROOT is an open-source project; all the sources as well as the full documentation can be found on its official website.

4.2. PROOF design goals

PROOF has been developed with the following goals in mind:

- Adaptability
- Transparency
- Scalability

Adaptability stands for the appropriate reactions to changes in the system environment, such as the load on the nodes, failures, etc. For instance, if a worker node suddenly fails during the computation, its work is reassigned to another worker(s) automatically without the user having noticed.

Transparency means that there should be no difference in terms of user's interactivity and results when running an analysis in ROOT locally or on a PROOF cluster. In fact, exactly the same format of the analysis code can be used for both types of sessions. More details on the required code structure and input data format can be found in Chapter 5.

Scalability stands for the fact that the more workers are in operation, the faster the results should be. In other words, there are no implicit limitations on numbers of workers involved in the computation.

4.3. PROOF multi-tier master-worker architecture

PROOF realizes a 3-tier architecture based on the master-worker computing paradigm. The third entity besides the *master* and the *worker* is the *client* (end user's computer), and it basically starts the whole computation. The client connects to a specified cluster node, serving as an entry point of the PROOF computing facility and always later acting as the *master*.

The computation begins when the master receives a complete task description from the client. The task comprises the analysis code in a predefined format (Chapter 5) and addresses of processed data on the cluster or worldwide. Only exceptionally, it could contain also the processed data itself. However, this is not recommended for large data sets as they would have to be transferred not only from the client to the master, but later also from the master to individual workers.

After receiving the task, the master node decomposes it into several smaller independent parts and distributes them among its *workers*. Workers process their part of the task and send back their partial results. The master accepts these partial results and merges them into the final result, which is then sent back to the client.

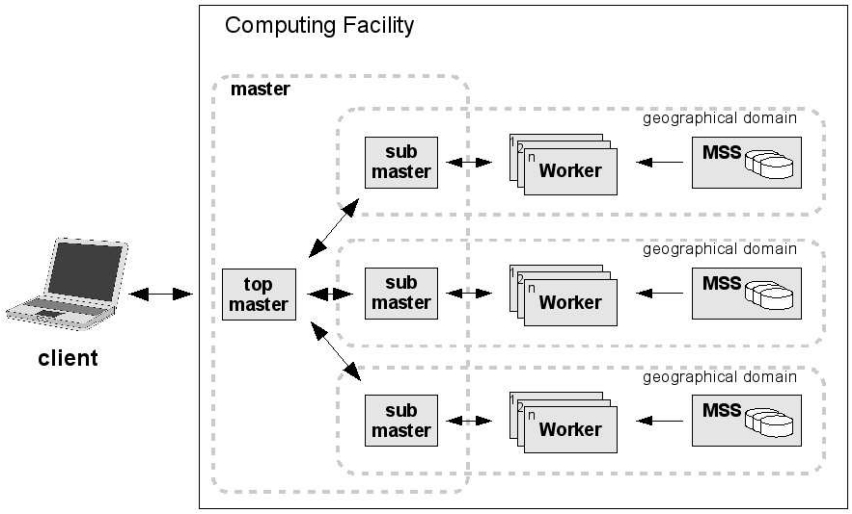


Figure 7 - PROOF Multi-tier master-worker architecture
(Image source: CERN)

The master tier can be also multi-layered as shown in Figure 7, following thereby the hierarchical master-worker paradigm (Chapter 2.3). The multi-layer concept was originally introduced to PROOF because of necessity to serve geographically separated domains (federated clusters). However, as the PROOF clusters are getting larger and processed data bigger, it is worth it to use the multi-level configuration also for homogeneous clusters. The main reason is the single master, which may become bottle-neck in the case of too many workers or too long merging of sub-results in the end. Both situations are described in detail in Chapter 6.

4.4. Packetizer - load-balancing engine of PROOF

The PROOF master does not just divide the task to pieces equal to the number of its workers. Instead, the accepted task is being gradually cut into pieces called *packets*, which are continuously sent to workers. The *packet* is only a description of a sub-task, and it does not contain any data itself. Typically, it carries a full name of a file with HEP events (located anywhere), and then a range of events, which should be processed. Since the events are uncorrelated, they can be processed independently, which means on any node and in any order. This is where PROOF exploits the inherent parallelism in HEP data.

For simplicity, we will refer to the *size of a packet* as to the number of events the packet describes, even if the size of the packet object itself is naturally always the same. The size of an assigned packet may vary according to the worker's recent performance and estimated time until the end of the processing. In principle, the packet size can be as small as the basic unit being processed – one collision event.

The process of packets calculating and assigning is managed by the load-balancing engine called the *Packetizer*. There is a separate instance of the Packetizer on the master node for each job. If a multi-master configuration is used, then there is one Packetizer on each of the sub-masters.

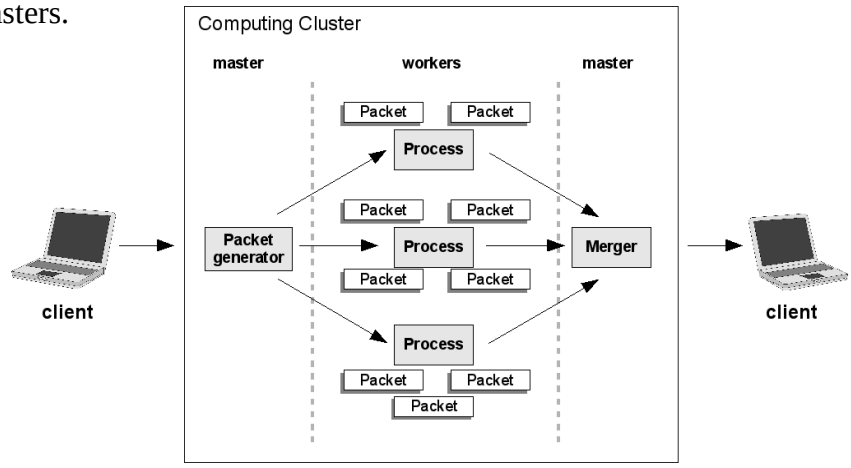


Figure 8 - Processing of PROOF packets
(Image source: CERN)

The *pull approach* is used for work distribution, i.e., workers ask for the next package when they have finished the previous one. The main goal of the pull approach is to let all the nodes finished at approximately the same time. Once they have finished their work, i.e., they do not get any new packet when they ask the master; they send their sub-results to the master. In other words, there is no gradual merging of sub-results on the master during the computation phase when workers are still receiving new sub-tasks.

The PROOF Packetizer not only distributes the work among nodes, but also accepts confirmation that this work was successfully processed. If some worker fails, the Packetizer is responsible for reassigning all its work to other nodes.

The optimal Packetizer strategy depends on the task type being processed. Some strategies available in PROOF are described in [16]. In a data-driven task type *data locality* is the main optimization criteria. Some data sets needed for a given job may be located on the worker nodes assigned to that job, while some other data sets may be located elsewhere. Naturally, a worker is given the local data sets to process first, if it has any, and then remote ones in order to minimize data transfers among cluster nodes.

4.5. Merging outputs

The sub-result of each worker can be found in the form of an *output list*. We do not have to care about the order of merging of individual output lists, as the merging is commutative. In other words the final result is the same no matter if we merge together outputs from workers w_a and w_b first; and then we merge the result with the output from worker w_c. Or if we merge outputs from w_b and w_c first, and then we merge w_a. Commutativity of merging simply comes from the independence of HEP events.

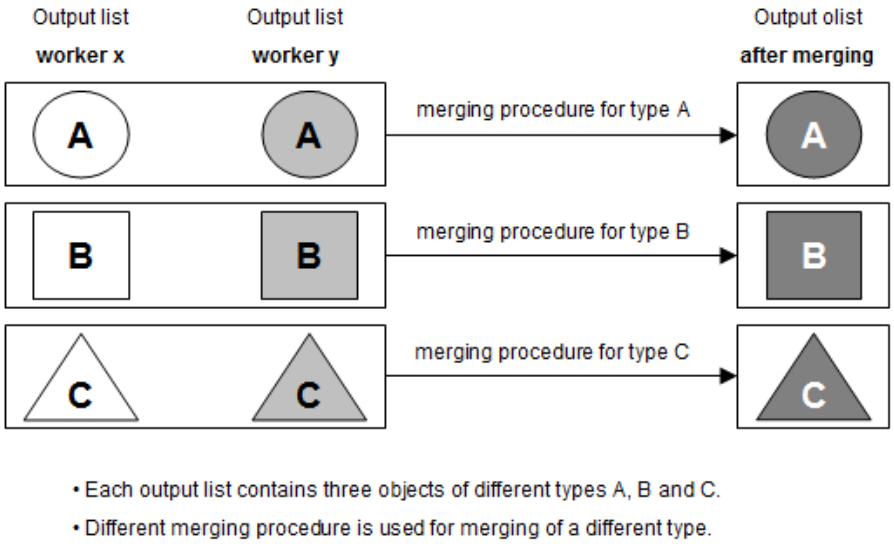


Figure 9 – Example of output lists merging

An output list can contain output objects of various types according to the definition in the analysis code. Each output list contains the same number of objects of the given type. In other words, all output lists are equal from the type point of view as shown in Figure 9.

Merging of two objects of the same type always needs the same amount of resources and therefore, under equal conditions, takes the same time, because exactly the same code runs. However, merging time of two objects of type A and merging time of two objects of type B can be significantly different even if measured in equal conditions. The length of the merging strongly depends on the merging function itself, which can range from simple and quick addition to some more complicated procedure.

One should note that characteristics of the output depend completely on the analysis code and there is no forward relation between the input and the output. Therefore we cannot make any assumptions in advance, either about the number of objects in the output list, or about their type determining the size (without e.g., performing some parsing and grammatical analysis of the input code). However, both characteristics determine, together with the merging procedure and cluster configuration, the length of the merging.

In one case, the output object can be, for instance, an integer number; and the merging procedure can be the choice of a minimum of two such numbers. In another case, the output object can be a full multidimensional histogram. Merging of two histograms then, naturally, involves going through all their dimensions and bins, and adding appropriate values together.

4.6. Scheduling in PROOF

The goal of the scheduling is to efficiently use computing resources in order to minimize execution times of processed queries. The PROOF Scheduler assigns workers to each query submitted to PROOF cluster in accordance with its scheduling policy. If the policy allows, a query can also be rejected (in the case of system overloading) or put in the waiting queue.

The scheduling policy is managed by a cluster administrator via configuration files, which simply determines the scheduling algorithm. The number of assigned workers for a query or a user can be determined either by the current system load or by user's or group priority. These priorities can be set in a static way, or they can be calculated dynamically in cooperation with some monitoring systems, e.g., MonALISA [17].

5. HEP data analysis with PROOF

In this chapter, we present PROOF mainly from the view of an end-user physicist. PROOF accepts both task description and input data in a special format, which ensures that it can be parallelized automatically without user having to take care of parallel resources. We describe the analysis code structure and how it is parallelized when being executed on the PROOF cluster. We also provide an example PROOF session, in order to give a reader authentic feel of how the work with PROOF can really look. However, we start with one important observation.

5.1. Typical use-case

Typically, once a physicist has developed some analysis, it is very probable that such analysis code will run many times, just each time on different data sets. Naturally, the greater the amount of events has been processed within the analysis, the more precise the conclusion can be. One must always find a good balance between processing as much data as possible *at once* and seeing some output in reasonable time, typically in hours at maximal. The gradual analysis of huge data, performed in multiple runs can, therefore, be a good compromise. Moreover, the same analysis can sometimes be intentionally performed on data sets coming from various stages of experiment or measured under different conditions. The stand-alone cases are then various data quality analyses which are run regularly on most of the gathered data.

5.2. TSelector query interface

The task to be run in parallel is called *query* in PROOF terminology since usually the analysis itself can be considered as a more complicated query on the HEP data. The PROOF query must be implemented as a class derived from *TSelector* abstract class. A simplified version of TSelector interface is provided below. The complete description is available as a part of the ROOT Reference Guide [18].

```
class TSelector {
public:
            virtual void        Begin(TTree *);
            virtual void        SlaveBegin(TTree *);
            virtual Bool_t      Notify();
            virtual Bool_t      Process(Long64_t /*entry*/);
            virtual void        SlaveTerminate();
            virtual void        Terminate();
};
```

The user writes his/her own analysis code in the predefined methods, having in mind the general analysis workflow, i.e., when and where each piece of code is executed. The data structure used in TSelector is the *TTree*. For simplicity we can perceive TTree as an optimized container for HEP events, which is being typically stored within a so-called *ROOT file*.

- *Begin* is called on the client side before starting the data processing. It prepares the global environment for the analysis like histograms to be filled with result values.
- *BeginSlave* is called on every worker before starting the data processing. It can prepare the local environment.
- *Notify* is called on the worker when a new file has been opened.
- *Process* is a piece of code that is executed on every event of the input *TTree.*
- *TerminateSlave* is called on every worker after the data processing on this node has finished.
- *Terminate* is called on the client side after all workers have finished their jobs. It is where result are available and can be presented in the required way.

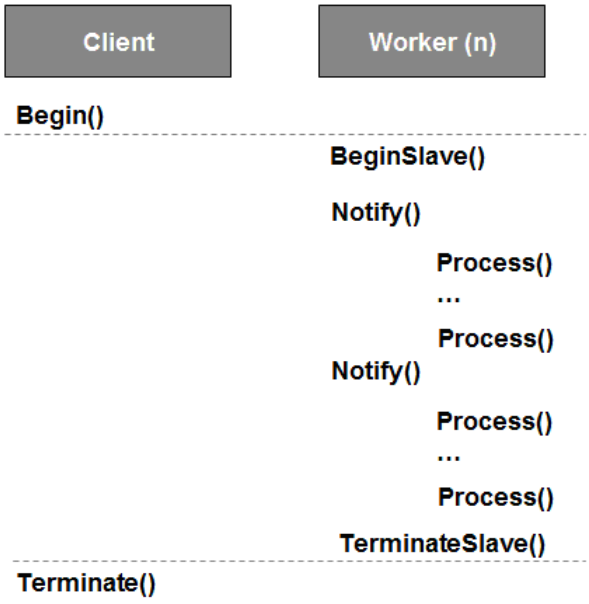


Figure 10 - TSelector calls flow in PROOF

The described selector approach is ROOT-transparent. It means that the user can run exactly the same code locally, also within the standard ROOT session. In such a case, of course, they would miss the advantage of the speed-up due to the parallelization, but they would get the same results. However, non-parallel approach is almost impossible for analysis of large data sets. In the following text, we will refer to the *selector* as to the class derived from *TSelector* and containing the analysis code.

5.3. Simple sample PROOF session

The user connects to some PROOF cluster via TProof API by typing the following code in the standard ROOT prompt:

```
root[0] TProof *p = TProof::Open("user@master:port")
```

User is the user name for accessing the cluster, and *master* is the name of the machine (the master node) to connect to. The standard PROOF port number (on which appropriate daemon listens and accepts connections) assigned by IANA [19] is 1093, and this is used by default if omitted.

We overlook the client authentication part since it is beyond the limits of this work, as well as possibilities of uploading of specialized user packages. The sufficient coverage on these topics can be found within the standard PROOF documentation.

After the connection to the specified PROOF cluster user gets a message[1] which informs them on the assigned computing power.

```
Starting master: opening connection ...
Starting master: OK
Opening connections to workers: OK (26 workers)
Setting up worker servers: OK (26 workers)
PROOF set to parallel mode (26 workers)
```

Before running the analysis itself the data must be prepared. Here, logical set *h1set* is created and filled with the data from four specified ROOT files.

```
root [1] TDSet * h1set = new TDSet("h1");
root [2] h1set->Add("dstarmb.root");
root [3] h1set->Add("dstarp1a.root");
root [4] h1set->Add("dstarp1b.root");
root [5] h1set->Add("dstarp2.root");
```

All of the above used ROOT files come from the H1 collaboration at Deutsches Elektronen-Synchrotron (DESY), Hamburg and can be downloaded freely from the ROOT homepage [2]. Each of the ROOT files contains a part of TTree named *h1*.

From the same location, user can download also the sample H1 analysis files (*h1analysis.C* and *h1analysis.H*) containing the selector for the four data files.

[1] The example was obtained from *alicecaf.cern.ch*, access to which was kindly provided by the ALICE Collaboration.

Data set name	File size (MB)	No. events
dstarmb.root	20.3	21 920
dstarp1a.root	68.2	73 243
dstarp1b.root	79.9	85 597
dstarp2.root	96.0	103 053

Table 4 – Basic properties of H1 data sets

There are several ways to analyze data via selectors on a PROOF cluster. The easiest way is to call the *Process* method right on the *TProof* object and pass both the data set and the selector as its arguments. In fact, by this call, we issue processing command to the master.

```
root [6] p->Process(h1set, "h1analysis.C");
```

Within the *Process* method, we can further specify e.g., the processing mode, which can be either synchronous (interactive) or asynchronous (batch). By default, a query is processed in a synchronous way. Other useful optional arguments are for example the number of events, i.e., TTree entries to process (all by default) or the starting entry (the first one by default).

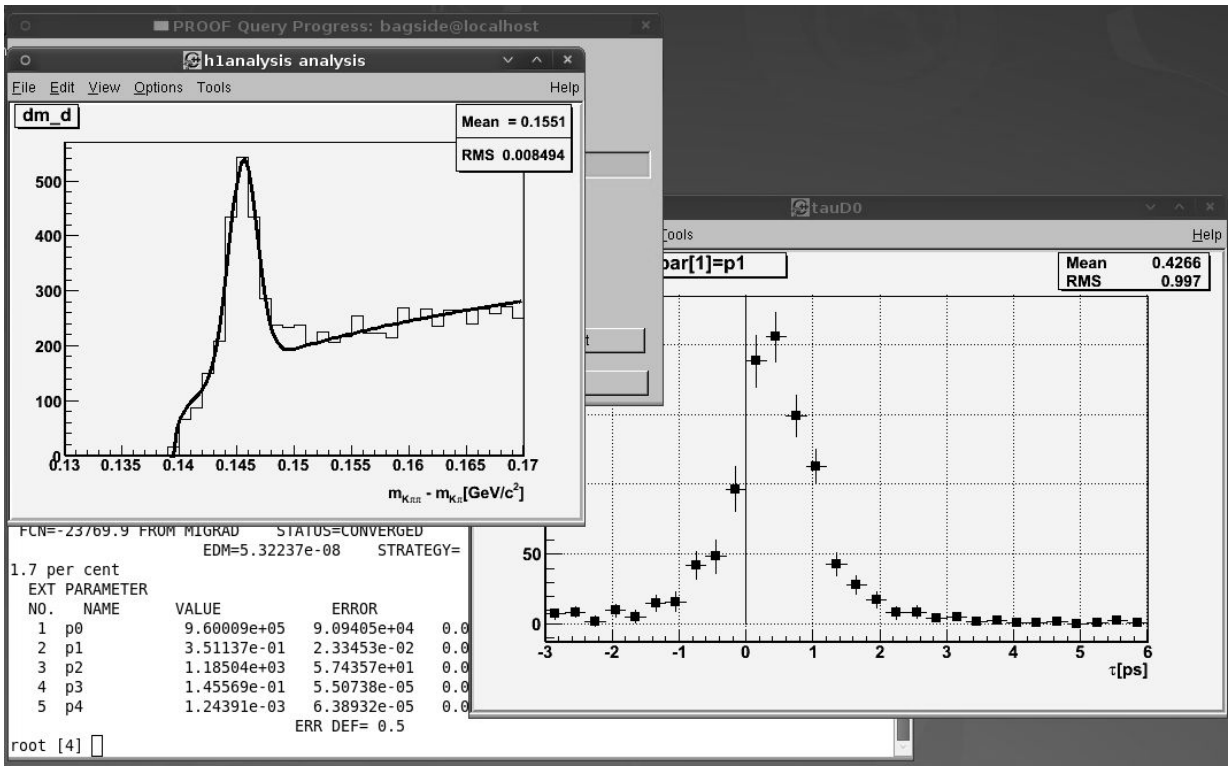
Since we issued the Process command in the synchronous mode, we would wait for the query to be processed before the command line is enabled again. We are regularly informed about the state of the processing via the progress bar in the *PROOF Query Progress* window.

After a while, we get the following text results:

```
FCN=-23769.9 FROM MIGRAD  STATUS=CONVERGED 225 CALLS 226 TOTAL
EDM=5.32237e-08 STRATEGY= 1
ERROR MATRIX UNCERTAINTY 1.7 per cent

EXT PARAMETER                                   STEP         FIRST
NO. NAME      VALUE            ERROR          SIZE         DERIVATIVE
 1  p0     9.60009e+05  9.09405e+04  0.00000e+00  -1.03857e-08
 2  p1     3.51137e-01  2.33453e-02  0.00000e+00   2.83166e-02
 3  p2     1.18504e+03  5.74357e+01  0.00000e+00   2.75548e-06
 4  p3     1.45569e-01  5.50738e-05  0.00000e+00  -5.42216e-01
 5  p4     1.24391e-03  6.38932e-05  0.00000e+00  -1.56613e+00
 ERR DEF= 0.5
```

Followed by the graphical output:



Picture 1 - Example H1 analysis running on PROOF (results)

6. PROOF query processing on single master configuration

In this chapter, we focus on the detailed analysis of the PROOF query processing when using cluster setup with the single master and *N* workers. The main goal is to find weak points of this processing, such ones that they could be eliminated by more sophisticated master role's assignment (Chapter 2.3). Besides the description of the query processing work-flow, we also present a set of important observations based on the analyses of real PROOF logs.

6.1. Typical PROOF cluster

We expect to have a typical PROOF *homogeneous cluster*, meaning that all its nodes are of the same hardware equipment and, therefore, offer the same computing power. In practice, when more queries are processed at once, the real exploitable power of different nodes can vary depending on their actual external load. For simplicity, we neglect this difference as it is mostly unpredictable. An option would be the regular measuring of the current external load on each node and its employment into the model. However, this can be quite time consuming and, therefore, impractical, because we want to devote as much as possible of the computing power to the computing itself. Moreover, different external load on nodes can be successfully balanced by the PROOF Scheduler.

All the cluster nodes are also expected to be linked via a *fast local network*, allowing us to consider communication channels equal between any two nodes. We also do not expect the *network latency* between individual nodes to be the most limiting factor, as it usually is on computational grids.

Observation 1

The most limiting factor of PROOF query processing is the access rate to the processed data sets, not the network latency.

6.2. Task execution phases for single master

We define the *task execution time* as the time elapsed between starting the computation and getting the final result. In the case of PROOF, it starts by the master accepting the task from the client and finishes by the master sending the final result to the client.

For simplicity, we do not include the time spent on sending the task from the client to the master and on sending the final result from the master back to the client. Obviously, these parts cannot be parallelized and are considered to have a fixed duration for the given task and environment. Therefore, they are not interesting for us.

In the following text, we will use *t* to denote a PROOF task (i.e., selector – Chapter 5.2). We will call *task size* as a number of ROOT events to process, which we denote *e.* Please note that *e* can differ each time when *t* runs on the PROOF cluster. We also denote as l_q the output list containing *q* objects, which is left on every worker after the processing.

The execution time of task *t* with *e* events on the single master and *N* worker nodes is then composed of three main sequential phases: *initialization*, *computation* and *finalization.*

6.2.1. Initialization

Initialization covers the period from the beginning of the processing to the point when all workers have received their first packet to process.

- **Start:** The master has received task *t* from the client.
- **End:** All *N* allocated workers have received initial packet from the master.
- **Length function:** $init\_master_t(N)$

Simply said, the more workers to initialize we have, the longer the initialization phase takes. Number of workers *N* determines the length of the initialization because they are all informed *sequentially* by the master about the fact that they act as workers for this computation, and they are all given the analysis code and the initial packet. The initialization phase is independent on the current task size *e,* as each packet object has always the same size regardless how many events it describes (Chapter 4.4)

Note that some packets can be completely processed even during the initialization phase.

Observation 2

As observed from PROOF logs, the initialization phase length is practically negligible in comparison to the computation and finalization. The reason is that *N* is usually in range of tenths (hundreds as maximal) and the appropriate part of the analysis code to transfer is usually not bigger than a few kilobytes.

6.2.2. Computation

We call *computation* the phase when all *N* workers are processing some packets.

- **Start:** All *N* allocated workers have received initial packet from the master.
- **End:** The first worker has sent its output to the master.
- **Length function:** $computation_t(e, N, 1)$

The length of the *computation* phase is determined by the task size *e*, the number of workers in operation *N*, and the number of masters managing these workers, which is simply one in

the single-master case. Naturally, the bigger *e,* i.e., the more events to process while *N* is constant, the longer the computation phase takes. Similarly, the bigger *N,* the shorter the computation phase if *e* is constant. However, this works only until some point when the single master is unable to manage all its workers at once. We will focus on this situation in Chapter 6.4.

Observation 3

As observed from PROOF logs, the number of workers manageable by one master is constant for the given cluster, and independent on processed task. The role of the master during the computation phase is always the same for all types of queries – the work distributor. Workers can treat the data in a completely different way depending on the selector; but the master, or more precisely the PROOF Packetizer on the master node, always does the same thing under the same schema: it distributes the work among its workers.

6.2.3. Finalization

- **Start:** The first worker has sent its output to the master.
- **End:** The master has sent the final result to the client.
- **Function:** $final\_master_t(l_q, N)$

The finalization phase length for task *t* can be expressed as a function of $l_{q,}$ and the number of workers *N,* as it involves merging of *N* output lists, each of the length *q*.

We can distinguish two parts of the finalization. In the first part, the master is receiving output lists one after another as workers are finishing. The goal of minimizing this part (of finishing all workers at once) is the task for the PROOF Packetizer. In the first part of the finalization, the master does both the merging of already received outputs and managing of the workers, which are still processing.

When all workers have finished, the master focuses only on the merging itself. This second phase called *full merging* is a critical one because it is purely sequential and involves only the master node. Both finalization phases are depicted in Figure 11.

Observation 4

Full merging on the single master may create a significant bottleneck in the case of too large outputs or too complicated of merging procedures.

6.3. Resource utilization diagram

We can clearly see all the above mentioned phases in the *resource utilization diagram* (Figure 11), where the utilization for each node in time is depicted. We consider two states of a node: *busy* or *idle*. Periods in white color are periods in which a node was busy, i.e., doing some useful work. This is either communicating with the master or processing some packet if the node is a worker node. If color is grey it means that a node was idle in that time.

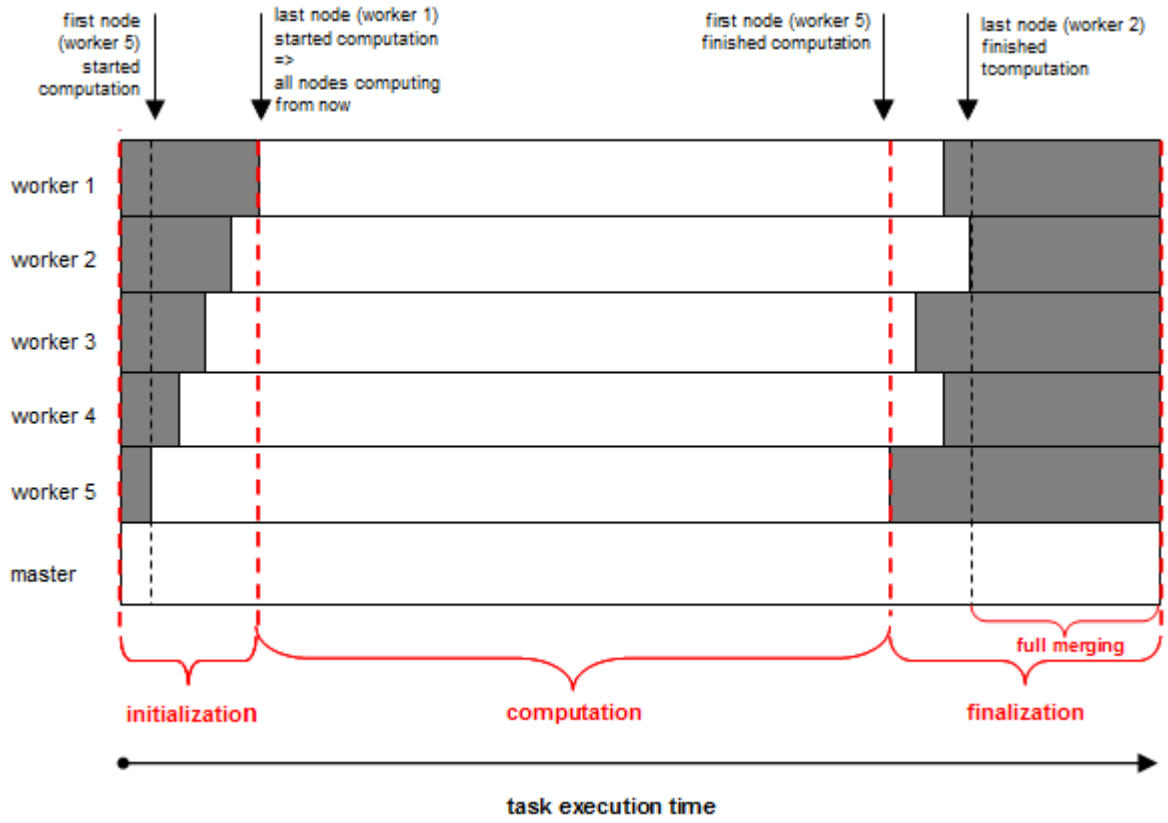


Figure 11 - A sample resource utilization diagram (individual phases not proportional)

6.4. Idle periods during computation

In the real environment, we can also recognize the idle periods (grey areas in resource utilization diagram) during the computation, not only in the beginning and in the end of the task execution (Figure 12). We call them *computation idles* and they may significantly prolong the total execution time. In fact, they directly express how the connection is fast and the master responsive. Naturally, it always takes some time to deliver a request for the next packet to the master, and in return, deliver the next packet back to the worker.

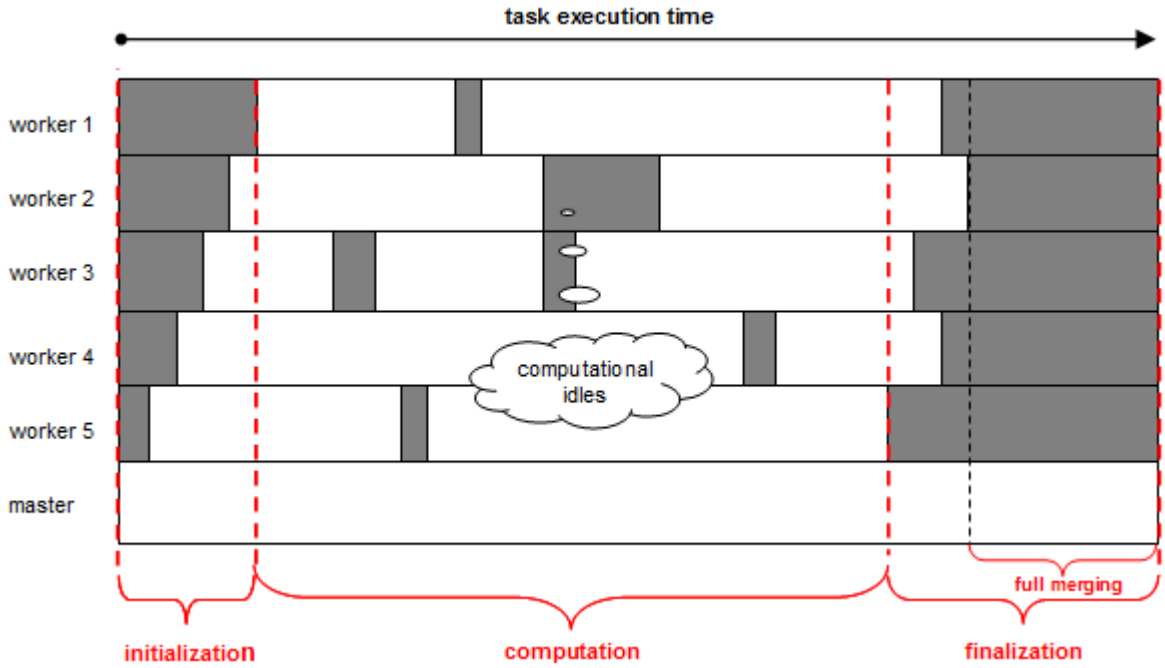


Figure 12 – Computational idles in resource utilization diagram (individual phases not proportional)

In the case of no congestion on the master (meaning that it can send a new packet immediately when it receives an appropriate request), the computation idles are expected to be as short as the network allows. Moreover, they tend to be regular if a worker processes tasks at a constant rate, and then sends also the requests at a constant rate. In such a case, computation idles can be observed and easily incorporated into worker's real processing rate.

If the master cannot respond immediately and thus the workers do not receive their sub-tasks within a granted time period, computation idles get longer (workers are starving for work).

Note that some idle times can occur also on the master during the computation (preferably in the beginning when all workers have already received their first data to process, but they have not asked for a new packet yet). However, we do not consider this phenomenon to be harmful, as the idle master does not slow down the computing in this case.

6.5. Summary

We can define the total task *t* execution time when processing *e* events on the single master and *N* workers as function $execution\_single_t\,(e,\ N)$, consisting of the lengths of the above mentioned sequential phases:

$$execution\_single_t\,(e, N) = init\_master_t\,(N) + computation_t\,(e, N, 1) + final\_master_t\,(l_q, N)$$

Formula 1 – Total execution time for task *t* (*e* events) on the single master and *N* workers

Naturally, using the *hierarchical master-worker* is relevant only in the case when it clearly speeds up the total task execution time. Such a speed-up can be expected in a situation when the single master strongly limits the overall performance and there are enough workers

from which another master(s) can be selected. In this chapter, we have addressed two main situations, which limit the total PROOF performance:

1) The master is too overloaded and then slow in serving its workers during the computation

2) Finalization on the single master (full merging) creates a bottleneck of the execution

Both issues can be possibly overcome by the use of the hierarchical master-worker if meeting certain requirements.

The important fact to note is that by deploying the sub-master, we always lose some real computing power which is then devoted on the sub-mastering. However, this loss can be greatly compensated by the speed-up of the merging or even of the computation (due to better resource utilization). In the next chapter, we will study both observed cases in more detail.

7. PROOF query processing using more masters

In this chapter, we describe the PROOF query processing when using the multi-master configuration; and we compare it to the single master configuration which was analyzed in the previous chapter. We focus on the changes in the computation and finalization caused by deploying of more masters, and we explain why these phases have almost antagonistic requirements on the number of masters in operation. Further, we show what the optimal number of masters for the computation and finalization is and what information we need in order to determine it. Again, we consider having a typical PROOF cluster as described in Chapter 6.1.

7.1. One level multi-master configuration

In the following text, we always consider only one level of sub-masters. It means that we have the top-master on the first level, all sub-masters on the second level, and all workers on the third level as shown in Figure 13.

- *To use one master* means to use the single master configuration.
- *To use more* or *s masters* (s ≥ 2) means to use *s* sub-masters on the second level, i.e., to have the top-master and *s* sub-masters under it.

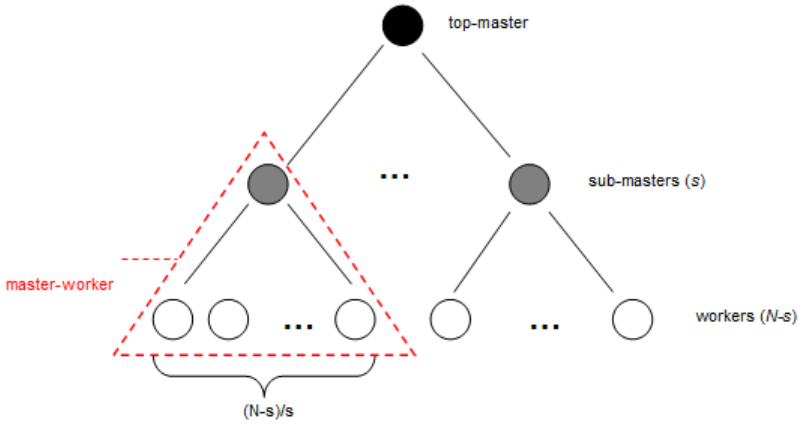


Figure 13 – Hierarchical master-worker system with s sub-masters

More levels of sub-masters can be useful either on a grid or in the case that the top-master itself becomes too overloaded. On a grid, adding some levels may be useful as it could help to better utilize the communication channels. Nevertheless, we consider a homogenous cluster to be our primal environment. Therefore, having more levels of masters is usually pointless until the single top-master becomes the bottle-neck of the whole system. Then basically the same problem must be solved as with the single master being overloaded, in this case just one level up. However, such a situation is not expected to occur on clusters running PROOF.

7.2. Task execution phases for more masters

We denote *S* the set of sub-masters (the top-master is not included). We denote *s* the number of these sub-masters, i.e., $s = |S|$. To summarize it, we have one top-master node, *s* sub-masters and *N-s* workers. In other words, *s* nodes from the original number of *N* workers became sub-masters and, therefore, they do not serve as workers anymore.

Obviously:

- $s \geq 2$ Two sub-masters create the simplest reasonable[1] multi-master configuration.
- $s \leq \lfloor (N-s) / 2 \rfloor$ Each sub-master must manage at least two workers.

Every sub-master node s_j $(s_j \in S)$ should manage a group of workers of approximately the same computing power. These groups can then process basically equal parts of the original task size *e,* i.e., *e/s* in the period of approximately the same length. Considering our assumption of the homogeneous cluster supported by the PROOF Scheduler forcing the uniform load distribution, we assign to each sub-master s_j $(s_j \in S)$ either $\lceil (N-s)/s \rceil$ or $\lfloor (N-s)/s \rfloor$ workers. For simplicity, we will expect that each sub-master manages exactly $(N-s)/s$ workers. Possible ± 1 difference should be balanced out by the PROOF Packetizer.

In the case of more masters, we can recognize even more task execution phases. The initialization/finalization phase runs now in parallel on each sub-master, as they are now the nodes, which communicate directly with workers. Therefore, we can simply imagine the whole master-worker schema as cloned and being put one level lower. The code, which previously ran only on the single master, now runs on each of *s* sub-masters (Figure 13). On top of that, we have the new initialization and finalization phases on the top-master node.

7.2.1. Top-master initialization

- **Start:** The top-master has received task *t* from the client.
- **End:** The first sub-master has received the initial sub-task from the top-master.
- **Length function:** $init\_topmaster_t$

In order to describe the task execution in the form of non-overlapping phases, we define the top-master initialization as the phase which ends by the receiving of the first sub-task on the first sub-master. Obviously, it is completely independent on the number of sub-masters *s* and, hence, of the constant length for task *t*.

[1] If *s = 1,* it means that the top-master manages only one master, which then manages all workers. From the practical point of view, such a configuration is possible, but it would have all the negatives of the single master configuration and on top of that, an additional overhead due to that one sub-master.

7.2.2. Initialization of one sub-master

The initialization of one sub-master s_i (managing *(N-s)/s* workers) is defined in accordance with the initialization of the single master managing *N* workers (Chapter 6.2.1)

- **Start:** Sub-master s_i has received the initial sub-task from the top-master.
- **End:** All *(N-s)/s* workers of sub-master s_i have received their initial packet.
- **Length function:** $init\_master_{si,t}((N-s)/s)$

7.2.3. Initialization of s sub-masters

Let us now focus on the joint initialization of *s* sub-masters from set *S*. They are all expected to have the same initialization length, as they all manage basically the same amount of workers. Their initializations should run, theoretically, in parallel; but they are started with a slight delay. Therefore, we define the initialization phase of *s sub-masters* as the period, during which at least one of these sub-masters is still in the initialization phase. In other words, the initialization of *s* sub-masters starts with the initialization of the first sub-master and ends with the end of the initialization of the last (slowest) sub-master (Figure 14). We will denote this period $span_{sj \in S}\{ init\_master_{sj,t}((N-s)/s) \}$

- **Start:** The first sub-master has received the initial sub-task from the top-master.
- **End:** All workers of all sub-masters have received their initial packet.
- **Length function:** $span_{sj \in S}\{ init\_master_{sj,t}((N-s)/s) \}$

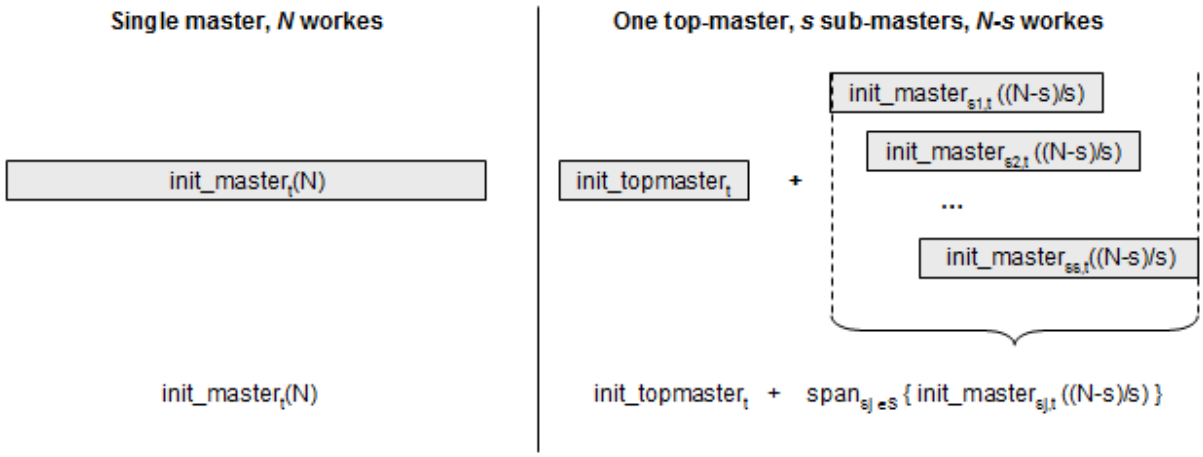


Figure 14 – Initialization on the single master vs. initialization on *s* sub-masters

7.2.4. Computation

Again, the computation is the phase when all workers (of all sub-masters) are processing their packets. Some packets can be processed also during the initialization or the finalization phase.

- **Start:** All workers (of all sub-masters) have started to process their first packets.
- **End:** The first worker (of any sub-master) has finished its last packet.
- **Length function:** $computation_t(e, N\text{-}s, s)$

7.2.5. Finalization of one sub-master

Finalization of one sub-master s_i (managing $(N\text{-}s)/s$ workers) is defined in accordance with the finalization of the single master managing N workers (Chapter 6.2.3)

- **Start:** The first worker of sub-master s_i has sent its output to s_i.
- **End:** Sub-master s_i has sent its output (sub-result) to the top-master.
- **Length function:** $final\_master_{si,t}(l_q, (N\text{-}s)/s)$

7.2.6. Finalization of s sub-masters

Finalization of s sub-masters starts with the finalization of the first sub-master and ends with the end of the finalization of the last (slowest) sub-master. We denote this period as $span_{sj \in S}\{ final\_master_{sj,t}(l_q, (N\text{-}s)/s) \}$.

- **Start:** The first worker of some sub-master has sent its output to its sub-master.
- **End:** All sub-masters have sent their outputs (sub-results) to the top-master.
- **Length function:** $span_{sj \in S}\{ final\_master_{sj,t}(l_q, (N\text{-}s)/s) \}$

7.2.7. Top-master finalization

- **Start:** All sub-masters have sent their outputs (sub-results) to the top-master.
- **End:** The top-master has sent the final result to the client.
- **Length function:** $final\_topmaster_t(l_q, s)$

7.3. Execution time summary

The initialization phase in the case of s sub-masters s_j $(s_j \in S)$, $s \geq 2$, each managing $(N\text{-}s)/s$ workers, is comprised of the initialization of the top-master and initialization of all s sub-masters: $init\_topmaster_t + span_{sj \in S}\{ init\_master_{sj,t}((N\text{-}s)/s) \}$.

Similarly, the finalization consists in the parallel finalization of s sub-masters followed by the finalization of the top-master: $span_{sj \in S}\{ final\_master_{sj,t}(l_q, (N\text{-}s)/s) \} + final\_topmaster_t(l_q, s)$.

The total execution time $execution_t\,(e, s, N\text{-}s)$ for task t (e events) on a setup with s sub-masters $s_j\,(s_j \in S)$, $s \geq 2$, which manage together $N\text{-}s$ equal workers can be expressed:

$$execution_t\,(e, s, N\text{-}s) = init\_topmaster_t + span_{sj \in S}\,\{\, init\_master_{sj,t}\,((N\text{-}s)/s)\,\} + computation_t\,(e, N\text{-}s, s) + span_{sj \in S}\,\{\, final\_master_{sj,t}\,(l_q, (N\text{-}s)/s)\,\} + final\_topmaster_t\,(l_q, s)$$

Formula 2 – Total execution time of task t (e events) on s masters and N-s workers

We can define $execution_t$ also for $s = 0$. It means that there is just the top-master and no sub-masters under it, i.e. it describes the single master configuration, which is, obviously, just a special case of the multi-master configuration with $S = 0$. Naturally, $execution_t(e, 0, N)$ must correspond to Formula 1 introduced in Chapter 6.5:

$$execution_t\,(e, 0, N) = execution\_single_t\,(e, N) = init\_master_t\,(N) + computation_t\,(e, N, 1) + final\_master_t\,(l_q, N)$$

Formula 3 - Total execution time of task t (e events) on single master and N workers

Generally, the more workers we have for the computation phase, the faster this phase can be. Masters do not process any data and, therefore, cannot speed up the computation in a direct way. Therefore, deploying of more masters usually does not help the computation phase to get faster. Per contra, it even slows down this phase as there are fewer resources devoted to direct data processing.

In general, we can say that usually $computation(e, N, 1) < computation(e, N\text{-}s, s)$ for $s \geq 2$. However, a special situation can be found when this does not hold. We focus on it more in Chapter 7.4

In contrast to the computation, for the finalization, we generally prefer to have more masters to which to distribute the merging load and thereby make the finalization shorter. The optimal number of masters for the finalization of N workers is discussed in Chapter 7.5

7.4. How computation can be speeded up by more masters

In this section, we concentrate only on the computation part of the task execution, not on the corresponding changes in the initialization and finalization. We focus on the situation described in Chapter 6.4 when the single master is too overloaded to respond to its workers on time. Consequently, the workers can become occasionally idle even during the computation.

Let us have general task *t*, which had the computation phase of length *k* (units of time) when executed on the configuration of the single master and *N* workers.

We denote *p* as the processing rate (events/unit of time) of a worker node in the case of no idle periods. All workers are considered equal (Chapter 6.1); therefore, the total theoretical processing rate of the whole PROOF cluster (*N* workers) in the case of no idle periods is *p*N*.

Let d_i be the total time when worker n_i was idle during the task computation phase. Then $p*d_i$ represents the lost in the task computation for n_i. In other words, it says how many more of the events could have been processed on node n_i if there were no idle periods. We denote $d = \sum_{i=1..N} d_i$ the total time lost on workers during the computation.

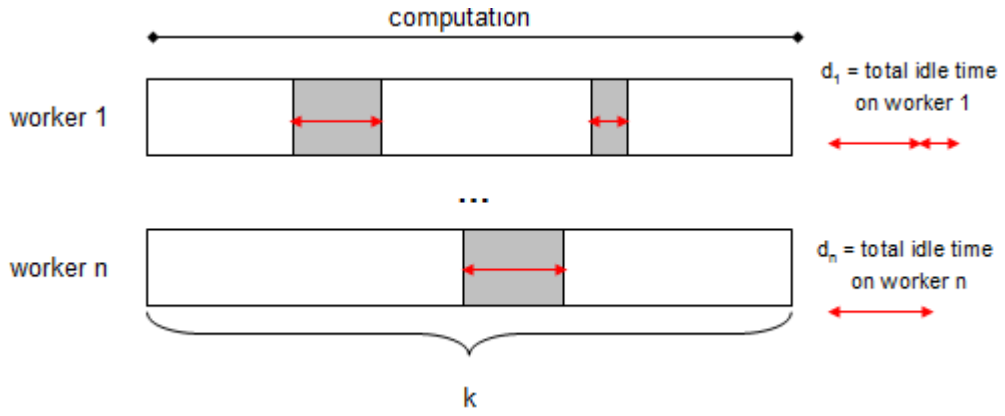


Figure 15 – Visualization of the time lost due to idle periods

The computation phase of length *k* is obviously comprised from the time when workers were really processing and from the time when they were idle, which is on average *d/N* per worker. We know that these *N* workers have together processed *e* events in parallel. It means that each worker was processing for $\frac{e}{p*N}$ units of time on average.

The total computation time *k* can be then described in the following way:

$$k = \frac{e}{p*N} + \frac{d}{N}$$

From that we can express *e* as: $e = (k * p*N) - d*p$

Let *s* be the minimal number of the sub-masters, for which all *N-s* workers are 100% utilized (when used for the computation of task *t*). In other words, *s (s ≥ 2)* is the smallest number of sub-masters so that all the idle periods on workers (previously caused by the master overloading) are eliminated. Hence the new total computation lost *d'* is equal to 0.

After deploying *s* masters, there are *N-s* workers left; so their cumulative processing rate is now a little lower, only *p*(N-s)* events per unit of time.

The new task computation length *k'* is then:

$$k' = \frac{e}{p*(N-s)} + d' = \frac{e}{p*(N-s)} + 0 = \frac{e}{p*(N-s)} = \frac{N*k-d}{N-s}$$

In order to decide which computation time is shorter, we compare *k'* and *k* i.e., *(N*k-d)/(N-s)* and *k*, i.e., *(N*k-d)* and *k*(N-s)* , i.e., *d* and *k*s*.

From that we get:

if $d \leq (k*s)$ *then* $k \leq k'$

if $d \geq (k*s)$ *then* $k \geq k'$

The second case expresses that the computation can be faster even on a smaller amount of workers if these workers are managed in a better way. Considering that the initialization and finalization are generally getting shorter when using more masters, we can conclude that the multi-master configuration in such a situation would probably lead to the speed-up of the overall task execution.

On the other hand, $k' \geq k$ does not necessarily mean that the overall task execution when using more masters must be longer. Changes, preferably in the finalization phase, must be taken into account before the final conclusion is made.

7.5. Optimal number of masters for finalization

Now we focus on the determination of the optimal number of masters for the finalization phase involving *N* workers (outputs).

Again, we consider having *N* workers, each with output list $l_q = \{obj_1, obj_2 \ldots obj_q\}$ containing *q* objects. These objects do not have to be equal (Chapter 4.5).

In addition, we have other *s* nodes serving as sub-masters, i.e., we have *N+s* nodes in total.

We denote mrg_i *(i=1..q)* as the time needed for merging obj_i *(i=1..q)* to the final result. Generally, for merging of *N* objects of the same type, we need to run an appropriate merging procedure at least *N-1* times. In the simplest case, it always takes two objects and returns a merged one, which is then used as the input for the next round.

Therefore, the finalization (merging) of *N* output lists l_q on the single master takes:

$$final\_master_t(l_q, N) = (N-1) * \sum_{i=1..q} mrg_i$$

Formula 4 – Merging objects on the single master

7.5.1. Another simplified view on parallel finalization of more masters

Now, imagine that we have *s* more masters to be used for the finalization, i.e., the same amount of outputs from *N* workers can now be processed in parallel on *s* masters; and the outputs of these *s* masters are then merged on the top-master. The total merging time is then comprised from the (span of the) parallel merging time on $s=|S|$ sub-masters s_j $(s_j \in S)$ and the merging time on the top-master (Chapter 7.3), i.e.:

$$span_{s_j \in S}\{final\_master_{s_j,t}(l_q, \frac{N}{s})\} + final\_topmaster_t(l_q, s)$$

In reality, it is not possible to precisely compute the span of finalizations on *s* sub-masters. The span can differ each run from the other depending on the current conditions on the cluster. In general, we can only conclude that the higher *s* we use, the more probable it is that the span gets bigger. Moreover, if the total number of workers *N* is not divisible by number of sub-masters *s*, some sub-masters must merge one more extra worker. This may also lead to the prolongation of their finalization phases and, therefore, to enlarging of the entire span.

We can also describe the length of the finalization on more masters in a different way. The top-master finalization starts when the top-master has received outputs from all sub-masters (Chapter 7.2.7). However, the top-master naturally starts to merge the first outputs as soon as it gets them even if it does not have all the outputs at the moment. This corresponds to the behavior of the single master.

Therefore, we can also describe the length of the finalization on *s* masters in the form of two consequent *single master-like finalizations* (Formula 5). The first proceeds on the fastest of all sub-masters. The second proceeds on the top-master node, starting at the point when the top-master receives the first output from the fastest of its sub-masters. As we do not know which sub-master will start the finalization as the first one, we omit the sub-master's index.

$$final\_master_t(l_q, \frac{N}{S}) + final\_master_t(l_q, s)$$

Formula 5 – Another simplified description of the finalization length in case of s sub-masters

The visual interpretation of the above mentioned formula is provided in Figure 16. In both cases, we have the same situation. On the left, we describe the length of the finalization in a similar way as we did in Chapter 7.2.6 and 7.2.7. On the right, we describe it by Formula 5.

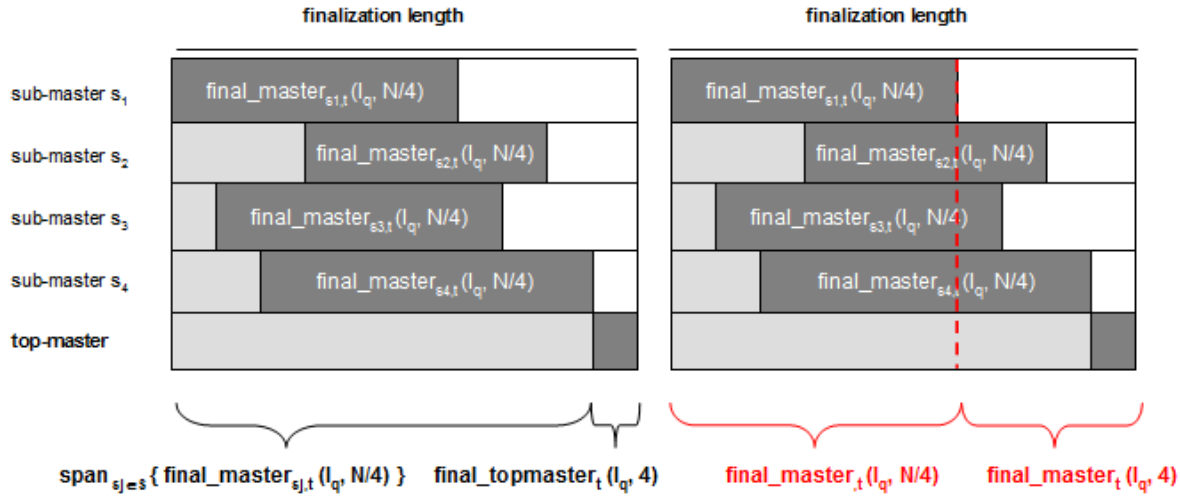


Figure 16 – Two possible ways of the finalization length description in case of more masters

By substitution of Formula 4 to Formula 5 we get:

$$final\_master_t(l_q, \frac{N}{s}) + final\_master_t(l_q, s)$$

$$= (\frac{N}{s} - 1) * \sum_{i=1..q} mrg_i + (s-1) \sum_{i=1..q} mrg_i = [\frac{N}{s} + s - 2] * \sum_{i=1..q} mrg_i$$

7.5.2. Speed-up of parallel finalization

In order to describe the gain we get by the deploying of s sub-masters to merge N output lists l_q, we need to compare Formula 4 with Formula 5:

$$finalization\_speedup(s, N, l_q) = \frac{final\_master_t(l_q, N)}{final\_master_t(l_q, \frac{N}{s}) + final\_master_t(l_q, s)}$$

$$= \frac{(N-1) * \sum_{i=1..q} mrg_i}{(\frac{N}{s} + s - 2) * \sum_{i=1..q} mrg_i} = \frac{N-1}{\frac{N}{s} + s - 2} = \frac{s*(N-1)}{s^2 - 2s + N}$$

As *finalization_speedup* is independent on l_q, we can define it also as a function of only two variables:

$$finalization\_speedup(s, N) = \frac{s*(N-1)}{s^2 - 2s + N}$$

Formula 6 – Finalization speed-up for N outputs in case of s masters

We say that s is the optimal number of masters for N workers if the best finalization speed-up for N workers is reached on the configuration with s masters. In other words, it is that number s which makes the value of function *finalization_speedup(s, N)* maximal when N is constant.

For example, the optimal number of masters for 80 workers is a maximum of function

$$finalization\_speedup(s, 80) = = \frac{s * 79}{s^2 - 2s + 80}$$

The graph of this function is displayed below:

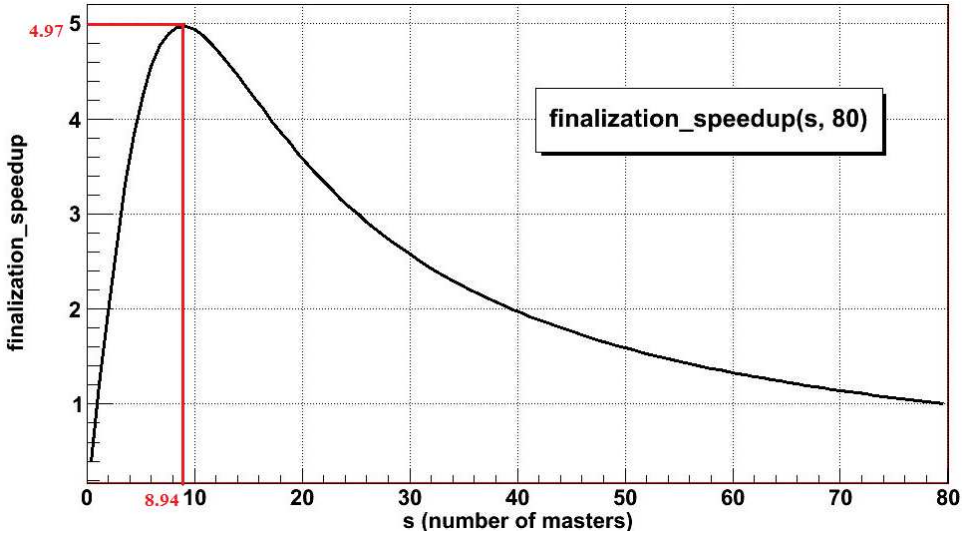


Figure 17 – Finalization speed-up for N=80

The optimal number of masters for 80 workers is 9 (the closest integer number to the function's real maximum at 8.94). Each of these masters then merges outputs from 9 workers, except one master, which merges only from 8 workers (8*9 + 1*8 = 80). The top-master then merges the output lists of these 9 masters. This corresponds to our intuition that the work load should be distributed uniformly not only among the sub-masters, but also between sub-masters and the top-master. The awaited speed-up in the finalization, when neglecting additional overhead, is almost 5.

Now, we find the maximum of *finalization_speedup* in a general case by computing its @*s* derivative and by setting it to zero.

$$\underline{\partial finalization\_speedup}\ (\partial s, N) = \frac{(N-1)*(N-s^2)}{(s^2 - 2s + N)^2}$$

As we always have N > 1, s ≥ 2, we get the optimal number of sub-masters for *N* workers as:

$$s_{optimal} = \sqrt{N}$$

Formula 7 – Optimal number of masters for merging of N outputs

Note: Formula 7 expresses the optimal number of *s* masters for *N* workers in the case when these masters are created on additional nodes. In order to find the optimal number of *s* masters in the case when these masters are started on some of *N* assigned nodes (i.e., number

of workers is then only *N*-*s*), we would have to compute *s* from the following equation: $s=\sqrt{N-s}$ **.** However, in Chapter 9, we will return to our primal equation Formula 7.

By substitution $s = s_{optimal}$ from Formula 7 to *finalization_speedup(s, N)*, we get the function of the best finalization speed-up for *N* workers:

$$best\_finalization\_speedup(N) = \frac{\sqrt{N} * (N-1)}{2(N-\sqrt{N})}$$

Formula 8 – Best finalization speed-up for N worker

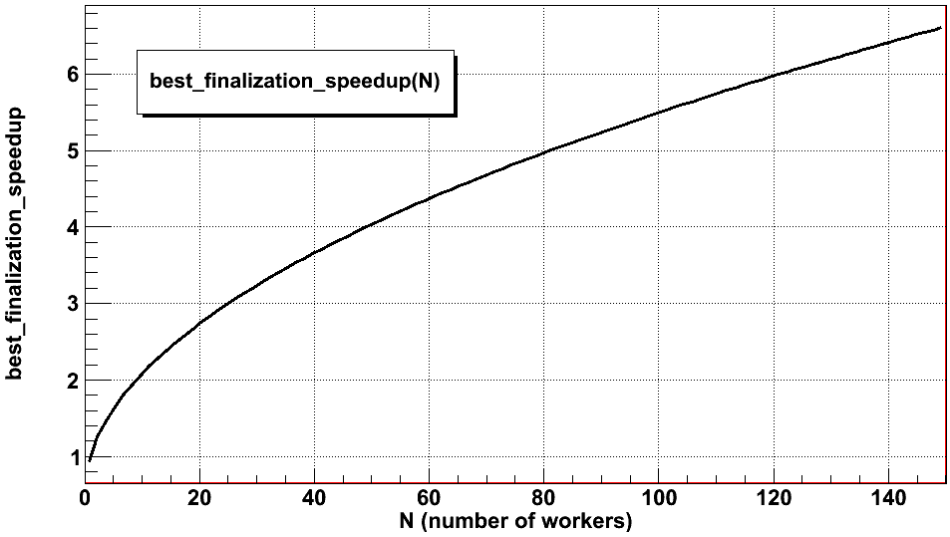


Figure 18 – Best possible speed-up in finalization due to deploying of more masters

It is clear that the bigger cluster we have, i.e., bigger *N*, the higher the speed-up in the finalization can be. While the expected finalization speed-up for clusters having around 20 nodes is not higher than 3, large clusters around 100 nodes are expected to get the finalization speed-up over 5. Obviously, the increase is not linear.

One should note that *best_finalization_speedup* function neglects completely any additional overheads that can occur in the real environment. Therefore we can consider it rather an upper boundary of our expectations.

8. In search for multi-master setup algorithm

In this chapter, we describe our first steps when searching for a suitable multi-master setup algorithm for PROOF. We explain how the problem of multi-master setup for PROOF differs to the general multi-master problem, and why we could not readapt any of the recent known algorithms. We also provide a description of the first algorithm which was later not used due to some new findings not directly connected to the algorithm's setup strategy itself. We explain this issue and its background. The main result of this thesis, another algorithm which was later successfully implemented, is presented in Chapter 9.

8.1. Using state-of-art knowledge

When first thinking of some suitable algorithm solving the multi-master setup problem for PROOF, we naturally tend to reuse or adapt some of the already developed ones. In Chapter 3, we presented a polynomial algorithm for determination of the optimal node for the single master. We also referred to a heuristic approach for determining locations of more masters, which itself is an NP-hard problem. Both of these algorithms focus on the maximizing of platform's throughput, i.e., on maximizing the total number of processed *application tasks* in a time unit. We are aiming at the minimizing of the execution time of the single PROOF query. Nevertheless, both these approaches can be considered equal thanks to the fine granularity of the PROOF query. We can simply consider processing of one or more collision events as the executing of one application task. The higher the throughput of the platform is, i.e., the higher the number of events processed in time unit, the shorter the time of a PROOF query.

However, the problem of multi-master setup for PROOF is significantly different to the general multi-master problem. Even if neglecting some alterations in the general work-flow which can be eventually overcome[1], the two problems are basically focusing on two different worlds.

Both works presented in Chapter 3 build on heterogeneous environment comprising nodes of various power and communication links of various transfer rates. However, a typical PROOF cluster offers a homogeneous environment as described in Chapter 6.1. As a consequence, all nodes simply would be considered equal by the algorithms presented in Chapter 3. Moreover, all recent works focus on the problem of which node(s) to choose as master(s) in order to maximize the platform throughput. Such a configuration is then

[1] E.g., in PROOF: the single master is always determined by connection, masters do not perform any computations; work distribution is not performed in parallel with results collecting etc.

platform-dependent, meaning that it is used for all incoming tasks. Our problem is how many of the masters we need to have in order to make processing of a single PROOF query as short as possible. In other words, we are looking for the *specific configuration for each task*. Once this number of masters is determined, we can choose the nodes for them almost freely, regarding just the local presence of the processed data, etc.

To summarize it, in general, the main factor is the *environment* and the main question is *which nodes* to choose as masters. In the case of PROOF, the main factor is the *task* itself; and the main question is *how many nodes* to use as masters in order to find a balanced configuration for both its computation and finalization phase. As both the problems are fundamentally distant, we had to develop a new algorithm for PROOF from scratch.

8.2. Record-based MMS algorithm

In this chapter, we present the first designed MMS algorithm for PROOF – the record-based algorithm. It takes an advantage from the typical PROOF use-case presented in Chapter 5.1: The same analysis code usually runs several times on PROOF cluster, each time, just on a different data set. The main idea of this algorithm is to use the information from one run to adjust the cluster setup for another run.

The finalization is the main phase, which utilizes masters; therefore, its length and total execution share is important when deciding how many masters to use. However, there is no relation between the input and output (Chapter 4.5) and, therefore, no reasonable way to estimate in advance, either its duration or the proportion to the execution phase.

The record-based MMS algorithm is a *heuristic algorithm* designed for the use in the dynamic environment of PROOF cluster. It is expected that the more runs of the same task we have done, the more precise the estimation of the optimal configuration for this task can be. However, as with many heuristic algorithms, the real quality is hard to proof formally. The version presented here is the first draft based on the basic estimations. The original plan was to create a simple pilot implementation and then to adjust the algorithm according to the observations, preferably focusing on the optimistic estimation Formula 9 (Chapter 8.2.3). However, due to some other complications, we eventually decided to abandon the record-based approach completely. Related facts are discussed in Chapter 8.2.4.

8.2.1. Algorithm's quick overview

When task *t* is submitted on the PROOF cluster with *N* nodes for the first time, it always runs on the single master configuration. Then, the proper values in *Record table(t, N)* (Table 5) are filled.

When *t* is submitted for the second time (and if, again, it is assigned *N* workers), the previous results from *Record table(t, N)* are checked. Based on that, it is decided if more masters should be considered, i.e., if the finalization share in the total execution is significant enough.

- If so, we try to add one more master so we will use two masters[1] for the second run.
- If not, we run task *t* again on the single master configuration. When it is done, we adjust the stored information in *Record table(t, N).*

Every time we run *t* on *N* workers again, we try to add one more master to *h* masters, which were used the last time. However, we do it only if the following two criteria are met:

1) The previous configuration on *h* masters was more efficient than the one before on *h-1* masters (details on the *configuration's efficiency* in Chapter 8.2.2). In other words, we add one more master only if we see that this approach has been helpful so far.
   - If it has not been, we stay at the best configuration so far, i.e., on *h-1* masters.

2) The estimation of execution for configuration with *h+1* masters is promising in terms of its efficiency. Simply said, we believe that adding one master can help.
   - If we do not believe it can help, we stay at the best configuration so far, i.e., on *h* masters.

After every run we also update *Record table(t, N).*

Record table(task *t*, number of workers *N*)	
Variable	**Description**
$time_{last}$	The total execution time of the last run.
$final_{last}$	The length of the finalization phase from the last run.
$masters_{last}$	The number of masters used for the last run.
$events_{last}$	The number of events processed in the last run.
$time_{last\ -1}$	The total execution time of the run before the last run.
$masters_{last\ -1}$	The number of masters used for the run before the last run.
$events_{last-1}$	The number of events processed in the run before the last run.
p_s	The average processing rate of a node serving both as a worker and as a master
p_d	The average processing rate of worker *w*, which contains processed data i.e., $w \in D$
p_a	The average processing rate of worker *w*, which does not contain processed data i.e., $w \notin D$

Table 5 – RecordTable(t,N) contains all necessary information about runs of task t on N workers

[1] Meaning the top-master and two sub-masters as defined in Chapter 7.1

8.2.2. Changing conditions

We expect the PROOF Scheduler to assign to task *t* of user *u* the same number of nodes *N* each time. However, even if this is the default strategy, we cannot guarantee it. Therefore, we must remember one separate *RecordTable(t, N_k)* for each number of workers N_k ever assigned to task *t*. The optimal configuration naturally depends on the number of assigned nodes and cannot be shared. If we run *t* on several different N_k, we can simply imagine it as it runs on several different clusters.

However, we do not have to start with the single master configuration for each different assigned N_k. If the current number of assigned workers *N* is greater than some previous N_k, we can simply use records of (the highest found) N_k also for *N*. The explanation is that if the configuration with *s* masters was helpful for N_k workers, it is expected to be helpful also for *N* workers where $N > N_k$. On the other hand, if *N* is smaller than any N_k assigned so far, we have to start the algorithm from the single master configuration.

The number of events *e* to process can also differ (and usually does so) for each run. Therefore when comparing which configuration is faster (more efficient), we cannot compare only absolute lengths of the executions, but we have to normalize them by appropriate numbers of events (see the appropriate formula in Action 5 of the algorithm). This is the way we compute the *configuration's efficiency*, which can be used for comparing of configurations involving processing of different numbers of events.

In this algorithm, we also neglect the external load on the nodes for the same reasons as described in Chapter 6.1.

8.2.3. Detailed description of record-based MMS algorithm

All used variables are task *t* related, so they should have another index *t*, determining the task. However, we omit that index for brevity.

INPUT:

- Code input: Selector *t*
- Data input: Addresses of *d* different places containing data sets with *e* events to process in total
- *N* assigned worker nodes n_i $(i=1..N, N \geq 6)$ + master node M
- *FINAL_SHARE* constant defined by cluster administrator (e.g., 10 %)
    - The maximal finalization share which is still considered insignificant

START - INITIALIZATION:

Denote by *D* the set of assigned nodes n_i, which contain some data sets for task *t* locally. In general *D* can range from the empty set (if all the data are remote) to $|D| = N$, meaning that each of the assigned nodes contains some data sets for task *t* locally. We cannot influence the choice of the nodes as this is the task for the PROOF Scheduler. However, we assume that if some nodes contain data sets for task *t*, they are assigned to *t* in preference.

Go to ACTION 1

ACTION 1 - Checking if task *t* has ever run before

```
if (task t has run before on N nodes ) {
        Load Record table(t, N)
        Go to ACTION 3
}
else (task t has run before on M nodes,  M < N)  {
        Create new Record table(t, N)
        Copy all values from Record table(t, M) to Record table(t, N)
        Go to ACTION 3
}
else {  // Task t has never run or task t has run only on M workers, where M >N
        Create new Record table(t, N)
        Set all values in Record table(t, N) to 0
        Go to ACTION 2
}
```

ACTION 2 - Running task *t* on the single master configuration with *N* workers

Run task *t* on the single master configuration with *N* workers. From the task execution, obtain the following information:

- Number of processed events e_i on each worker node n_i.
- Computation phase length *k* (including the initialization phase).
- Finalization phase length *f* .

Shift the last configuration results so far (if no previous results exist, all values are 0):

$time_{last-1} = time_{last}$

$masters_{last-1} = masters_{last}$

$events_{last-1} = events_{last}$

Remember current results as the last ones:

$time_{last}$ = (k+f) *// Total execution time*

$masters_{last}$ = 1 *// Number of masters equaled to 1 in this run*

$final_{last}$ = f *// Length of the finalization phase*

$events_{last}$ = e *// Number of events processed in this run*

Compute p'_d and p'_a from the previously obtained information.

$$p'_d = \frac{\sum_{i;n_i \in D} e_i}{k * |D|} \qquad p'_a = \frac{\sum_{i;n_i \notin D} e_i}{k * (N - |D|)}$$

- p'_d is an average processing rate (events/time unit) of a worker node that belongs to set *D*, i.e., of a node, which contains some of the processed data sets locally.
- p'_a is an average processing rate of a worker node, which does not belong to set *D*.

if (p_d == 0) { $p_d = p'_d$ } *// First run on single master configuration*

else { p_d = average (p_d, p'_d) } *// Multiple run on single master configuration*

if (p_a == 0) { $p_a = p'_a$ } *// First run on single master configuration*

else { p_a = average(p_a,p'_a) } *// Multiple run on single master configuration*

Go to FINALIZATION

ACTION 3 - Evaluating of multi-master suitability

last_fshare = $final_{last}$ / $time_{last}$ *100 *// Finalization share in last execution time (%)*

if ($masters_{last}$ == 1 AND last_fshare < MERGING_SHARE)
{ *// Insignificant share of finalization ⇒ no need for more masters*
Go to ACTION 2 *// Run single master configuration again*
}
else {
if ($masters_{last}$ == 1 AND ($masters_{last-1}$ ==0 OR $masters_{last-1}$ ==1))
Go to ACTION 5 *// Make estimation for $masters_{last}$ +1, i.e., for 2 masters*
else
Go to ACTION 4 *// Check results from last run*
}

ACTION 4 - Checking results from the last run

if $\left(\frac{time_{last}}{events_{last}} < \frac{time_{last-1}}{events_{last-1}}\right)$ *// Efficiency of last configuration higher*

{

if ($masters_{last}$ > $masters_{last-1}$) *// Last number of masters higher*

Go to ACTION 5 *// Make estimation for $masters_{last}$+1 masters*

else

Go to ACTION 6 (run on $masters_{last}$ masters)

}

else *// Efficiency of last configuration not higher*

// => go back to configuration before last one

Go to ACTION 6 (run on $masters_{last-1}$ masters)

ACTION 5 - Estimation of the computation length for *$masters_{last}$ + 1* masters

Note: *For resources utilization reasons, we allow a node to serve both as a worker and as a master at once.*

By default, put all N assigned nodes to set $W = \{n_i\}_{i=1..N}$.

Construct $S = \{n_j\}$, set of $masters_{last} + 1$ masters, i.e., choose some $masters_{last} + 1$ nodes from W, preferably those not belonging to D[1].

Load previously measured values p_a, p_d and p_s. Note that p_s can still be equal to zero, which means that no multi-master configuration has run so far. Based on the previous results, estimate processing rate p_i for each node n_i :

for each $n_i \in W$ (i = 1 .. N) {

if ($n_i \in S$) { *// Node running both worker and sub-master*

if (p_s == 0) { *// No multi-master configuration has run so far*

if ($n_i \in D$) { $p_i = p_d$ } *// Processing local data sets*

else { $p_i = p_a$ } *// Processing remote data sets*

}

else { $p_i = p_s$ } *// Previously measured processing rate of sub- master*

}

[1] We do this because we want to keep the processing rate of nodes $n_i \in D$ on the highest possible level in order to allow them to process as big as possible part of their local data right on them.

else // $n_i \notin S$ = *node running only worker*

{

if ($n_i \in D$) { $p_i = p_d$ } // *Processing local data sets*

else { $p_i = p_a$ } // *Processing remote data sets*

}

}

Estimate aggregated processing rate *P* for all *N* worker nodes: $P = \sum_{n_i \in W} p_i$

Estimate the execution time on $masters_{last} + 1$ masters for processing *e* events[1] according to Formula 9.

$$estimation(masters_{last} + 1) = \frac{e}{P} + \frac{final_{last} * masters_{last}}{masters_{last} + 1}$$

Formula 9 – Optimistic estimation formula for s masters

Check if at least the estimation for $masters_{last} + 1$ masters (normalized by *e*) is better than the real result obtained for $masters_{last}$ masters (normalized by previous $events_{last}$). If so, we try to run task *t* on $masters_{last} + 1$ masters. If not, we go back to the last configuration $masters_{last}$, which is also the best so far. Naturally, we prefer the configuration with higher efficiency.

if $\left(\frac{estimation(masters_{last} + 1)}{e} < \frac{time_{last}}{events_{last}}\right)$

Go to ACTION 6 (run on $masters_{last} + 1$ masters) // *Try to add one more aster*

else

Go to ACTION 6 (run on $masters_{last}$ masters) // *Stay at previous configuration*

ACTION 6 (run on *s* masters) - Running *t* on configuration of *s* masters

if (s == 1)

Go to ACTION 2 // *Run t on single master configuration*

Determine set *S* of *s* sub-masters the same way as described in Action 5.

Divide all *N* worker nodes into *s* sub-groups W_j *(j=1..s)* so that the aggregated processing rate of nodes in each sub-group is approximately equal. In fact, we can just divide *N* workers into *s* sub-groups randomly, only taking care that:

- Worker processes running on sub-master nodes are assigned to these sub-masters.
- Each worker group contains approximately the same number of workers from set D.

[1] See Chapter 8.2.4 for more details on the estimation formula

Once the worker groups for all sub-masters are determined, run t on these s sub-masters. From this run obtain and store the following information:

- Number of processed events e_i on each worker node $n_i \in W$.
- Computation phase length k (including the initialization phase).
- Finalization phase length f .

Compute p'_a, p'_d (for pure worker nodes only, i.e., do not include those nodes which serve as sub-masters, too) and p'_s from the previously obtained information:

$$p'_d = \frac{\sum_{i;n_i \in D, n_i \notin S} e_i}{k * (|D| - |D \cap S|)} \qquad p'_a = \frac{\sum_{i;n_i \notin D, n_i \notin S} e_i}{k * (N - |D \cup S|)} \qquad p'_s = \frac{\sum_{i;n_i \in S} e_i}{k * (|S|)}$$

Refine old p_a, p_d and p_s values using new values p'_a, p'_d and p'_s :

p_d = average(p'_d, p_d)

p_a = average(p'_a, p_a)

if (p_s == 0) { $p_s = p'_s$ }

else { p_s = average(p'_s, p_s) }

If we run the same configuration as the last time, we only replace $time_{last}$, $final_{last}$ and $events_{last}$ by the most actual value. This means that the information on configuration $masters_{last-1}$ is preserved. If we tried a new configuration, we also shift the results.

if (s != $masters_{last}$) *// Shift so far the last configuration results*

{

$masters_{last-1}$ = $masters_{last}$

$time_{last-1}$ = $time_{last}$

$events_{last-1}$ = $events_{last}$

$masters_{last}$ = s

}

$events_{last}$ = e

$time_{last}$ = k + f

$final_{last}$ = f

Go to FINALIZATION

FINALIZATION

Save RecordTable(t, N)

End of the algorithm

8.2.4. Optimistic estimation formula

For the estimation of the execution time on *s* masters, we use optimistic estimation Formula 9. The optimism in this formula is twofold. First, in the finalization phase, we simply neglect any additional overhead related to running of more masters. This overhead is likely to occur, but hard to express in the form of a mathematical function.

Second, when the multi-master configuration is being tried for the first time (on two masters), the sub-mastering overhead in the computation (expressed through p_s working rate) is not yet known and therefore ignored. This also lowers the estimated execution time and, therefore, encourages using of more masters. The estimation is intentionally stimulative. We consider better to try more masters, get worse result and thereby reach the upper limit, than not try and possibly miss a better configuration.

Optionally, also a version of the algorithm completely without the estimation formula can be considered. In such a case, we would always try to add one more master until the point when the efficiency of the configuration starts to decrease.

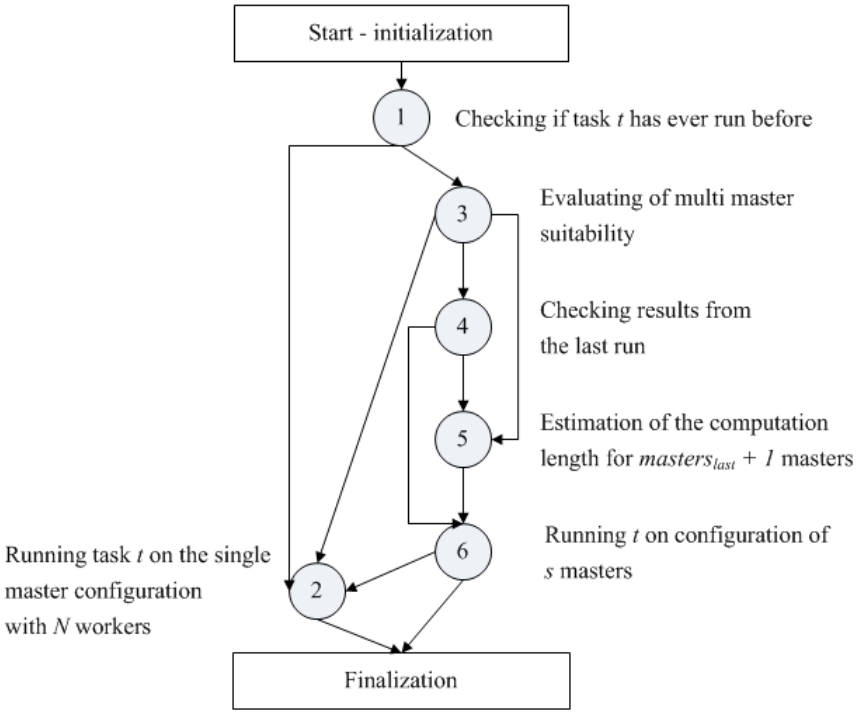


Figure 19 –Work-flow of the record-based algorithm

8.2.5. Pilot implementation learning

Pilot implementation of the record-based MMS algorithm revealed some interesting facts, which unfortunately made this algorithm hard to use in the real environment. The same use-case, which the algorithm tries to benefit from (i.e., multiple runs of the same analysis code), suddenly also became the biggest obstacle.

It turned out that *natural behavior of PROOF users* makes the recognition of the same analysis code a non trivial task. The users tend to use their analyses several times, but they usually perform slight changes, like adding or removing dumps, variable renaming, code formatting, etc. Logically, the analysis could still be the same, but the source code would look slightly different each time.

The basic idea was to recognize a selector, which has already run, by the file name and author. If the name and author are recognized, then the hash codes of both the incoming and the remembered source code are compared. If they are equal, then the upcoming run can be considered as a multiple run of the same analysis.

Even if the users would not rename their files and would not change anything inside, there is still an additional problem of storing and searching for hash codes. It became obvious that some more sophisticated way of storing and searching would have to be developed in order not to overload the PROOF master by this task.

Another option was to involve the users, themselves, in the analysis code recognition. For example, they would get a special number/code after their task's execution. When they run the same task again, they can use this number/code to benefit from the record-based algorithm. However, this would require education of the users in what *the same analysis code* means. PROOF clusters like CAF are also meant for analysis development. In such cases, the first version of the analysis code and the final one may differ significantly in measured characteristics. However, from the user's point of view, it is still the same analysis.

As we had overall low confidence in the above mentioned approach, we abandoned the record-based approach completely. Instead, we decided to design an algorithm, which would require neither obtaining nor storing of any information. Such one is introduced in the next chapter.

9. Merger-based multi-master setup algorithm

In this chapter, we present the main outcome of this work, the merger-based multi-master setup algorithm for PROOF.

As already explained, the optimal number of masters for the computation phase can greatly differentiate to the optimal number of masters for the finalization phase. Previously introduced *record-based MMS algorithm* (Chapter 8.2) simply tried to find a balance between these different needs. As a consequence, neither the computation nor the finalization was performed on the optimal number of masters.

Overcoming this undesirable feature became the primal motivation for the concept of two different configurations - one optimal for the computation phase and another optimal for merging. The main question is when and how to switch between these two setups, and not to affect or slow down the whole processing. Therefore we introduce a new type of the node - the *merger*. The merger acts as a worker during the computation phase and as a master during the finalization phase.

Merger-based multi-master setup algorithm changes the work-flow of the finalization part. Before diving deeper into these changes, we present a short comment regarding the optimal configuration for the computation phase.

9.1. Optimal configuration for computation

In general, we want to devote as much as possible of the computing resources to the processing itself; i.e., we want to have as many as possible workers for the computation. As mentioned in Observation 3 (Chapter 6.2.2), the amount of workers manageable by one master is task independent and can be determined by a simple load test. We expect to know this limit, and we will call it *max_manageable_limit*.

We can see the *max_manageable_limit* as the number of workers, for which the computation time is the shortest possible. In other words, adding just one more worker on top of *max_manageable_limit* would make the total computation time longer. The reason is the idle periods on workers, which are the result of the non responsive overloaded master (Chapter 6.4 and Chapter 7.4).

The optimal configuration for the computation is then the configuration with as little masters as possible, but no one of them serving more than *max_manageable_limit* of workers.

Let N be the number of workers assigned for processing of query t by the PROOF Scheduler. Then the optimal number of masters for the computation is simply given as:

$$masters_{opt} = \lfloor N / MAX\_MANAGEABLE\_LIMIT \rfloor$$

Formula 10 – Optimal number of masters for the computation

This is also how we find the initial configuration, i.e., the initial number of masters. If $masters_{opt} \geq 2$, then equal part of the workers is assigned to each of these masters. The same approach as in the record-based algorithm can be used for that. We determine the workers, which contain some data sets to be processed locally; and we assign these workers uniformly to all $masters_{opt}$ masters.

9.2. Optimal configuration for finalization

In order to incorporate mergers smoothly to the task execution, we made the following change into the PROOF processing work-flow:

When a worker has processed its last packet, it does not send its output to the master as it would usually do when using the standard approach. Instead, it just informs the master about the fact that it has finished. In turn, the master reacts in one of the following ways[1]:

- Establishes the finished worker as a merger for given number of workers *("Be merger m_i for p workers").*
- Tells the finished worker to which merger to send its output *("Send output to m_i").*

The total number of mergers established by the master follows the computations introduced and explained in Chapter 7.5. For N workers (N output lists), there are $\sqrt{N}$ mergers established. To be more precise, the first finished $\sqrt{N}$ nodes become mergers as these nodes are currently the most efficient ones and also free to work as masters.

The remaining nodes are just redirected to these specified mergers. Once a merger has received the outputs from all its workers, it sends the merged output to the master. The master then merges the outputs from all mergers to the final result.

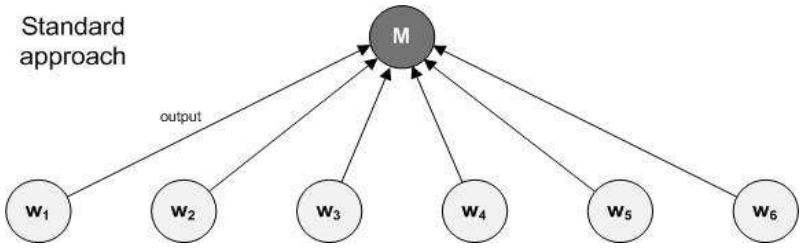


Figure 20 - Standard approach: Finished workers send their outputs automatically to the master

[1] In the case of less than 6 workers or unexpected problems (Chapter 9.5), the master can also ask the worker for sending the output directly to it *("Send output back").*

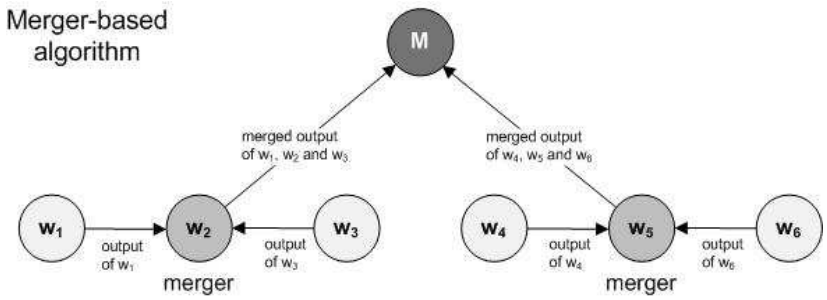


Figure 21 - Merger-based algorithm:
Workers send their outputs to mergers, which then send the merged outputs to the master

This means that each time query *t* is executed on the different number of nodes, the number of mergers is different, too. If *t* is executed on the same number of workers, the number of mergers is the same. However, different nodes can be chosen as mergers each time depending on their current performance.

9.3. Detailed description of merger-based algorithm

In this chapter, we describe the merger-based MMS algorithm more formally in the form of a simple communication protocol. As the algorithm interferes only with the finalization part, we start at the point when the first worker has finished. Until this point, there is no intervention to the current approach except the introduction of Formula 10 for setting of the initial number of masters (Chapter 9.1).

For clarity, we neglect here any potential problems such as a node failure or a message lost. These issues are later discussed in Chapter 9.5.

All used variables in the following text are query *t* related, so they should have another index *t*, determining the query. However, we omit that index for brevity.

VARIABLES AND MARKING:

- Master node M
- N workers nodes $w_1 \ldots w_N$

In the beginning we expected to have initialized the following variables:

On the master node:	Integer	mergers	= -1
	Integer	workers_on_merger	= -1
	Integer	created_mergers	= 0
	Integer	finished_mergers	= 0
	Integer	current_merger_index	= 1
	Array	merger_list[]	

On the worker node:	Integer	workers_to_merge	= 0
	Integer	merged_workers	= 0

We use auxiliary function *Message (from, to, content)* to describe that the node *from* is sending the message with the *content* to the node on address *to*. Names of nodes determine their addresses.

EVENT #1: Worker w_i *(i=1..N)* has finished its computation.

Action:

```
// Worker w_i informs master M that it has finished the computation
Message(w_i, M, "Finished");
```

Comment: Event #1 occurs on every worker node exactly once; i.e., it occurs *N* times in total during the execution of the algorithm.

EVENT #2: Master *M* is informed by worker w_i *(i=1..N)* that w_i has finished its computation.

Action:

```
if ( N < 6 )    // Too little workers for the merger-based algorithm
{
        // Master M asks worker w_i to send its output directly to it
        Message(M, w_i, "Send output back");
}
else    // Enough workers for mergers
{
        if (mergers == -1)   // First worker has finished - number of mergers not set yet
        {
                mergers = √N;        // Total number of mergers to be created [1]
                initialize array merger_list[] from index 1 to mergers;
                                                        // Array for addresses of mergers
                // Number of workers for one merger [2]
                workers_on_merger  = (N – mergers) / mergers;
        }
```

[1] More precisely, *mergers = round(√N)* as we want mergers to be an integer number. However, we skip *round* function for brevity.

[2] For simplicity, we expect that all mergers serve exactly the same number of workers. In practise, there is usually ± 1 worker difference; however, this does not change anything in the algorithm. Just the master has to then remember the number of workers for each merger individually.

```
        if (created_mergers <  mergers)   // Some mergers still to be created.
        {
                created_mergers++;
                // Master M informs worker w_i that it will serve as merger m_created_mergers
                // for workers_on_merger workers
                Message (M, w_i, "Be merger m_created_mergers  for worker_on_merger
                                workers");
                merger_list[created_mergers] = w_i;  // Save merger's address
         }
        else   // All mergers have been created – we redirect remaining workers to
              // these mergers in the round-robin fashion
        {
                // Master M tells worker w_i to send its output to merger
                // m_current_merger_index on address  merger_list[current_merger_index]
                Message (M, w_i, "Send output to m current_merger_index on
                                merger_list[current_merger_index]");

                current_merger_index++;
                if (current_merger_index > mergers)
                        current_merger_index = 1;
        }
}
```

Comment: Event #2 occurs on the master node exactly *N* times in total (once for each finished worker).

EVENT #3: Worker w_i $(i=1..N)$ has been informed by master *M* that it will serve as merger m_j $(j=1..\sqrt{N})$ for *p* workers.

Action:

```
workers_to_merge = p;
merged_workers = 0;
// As a merger, it starts to wait for workers_to_merge connections
// to come from other workers.
```

Comment: Event #3 occurs on every node, which is selected by master *M* as a merger. As master M creates $\sqrt{N}$ mergers, event #3 occurs $\sqrt{N}$ times in total.

EVENT #4: Worker w_i was told to send its output directly to master *M*.

Action:

// Worker w_i sends its output directly to master M

Message (wi, M, "Output" + output);

Comment: Event #4 occurs only if $N < 6$, or in the case of some unexpected error (see Chapter 9.5)

EVENT #5: Worker w_i was told to send its output to merger m_j on address w_x.

Action:

// Worker w_i sends its output to merger m_j on address w_x

Message (w_i, w_x, "Output" + output);

Comment: Event #5 occurs on every worker node which was not selected as a merger. As we have N worker nodes in total and $\sqrt{N}$ of them were selected as mergers; event #5 occurs exactly $N-\sqrt{N}$ times in total.

EVENT #6: Merger m_j has received output from worker w_i.

Action:

```
// Merger m_j accepts output of w_i and merges it with its current output
merged_workers++;
if (merged_workers == workers_to_merge) // Merger already merged all its workers
{
        // Merger m sends its output (including outputs from all workers_to_merge
        // merged workers) to master M.
        Message (m_j, M, "Output" + output);
}
```

Comment: Event #6 occurs on every merger node *workers_to_merge* times. In total, it occurs $N - \sqrt{N}$ times, i.e., once for each worker which is not a merger.

EVENT #7: Master *M* has received output list from merger m_j.

Action:

```
// Master M merges the output list from m_j with its current output list.
finished_mergers++;
```

```
if (finished_mergers == mergers)      // All mergers already finished
{       // Master send its output – which is also the final result now – to the client.
        // It is the end of the algorithm
        Message(M, client, "Output" + output);
}
```

Comment: Event #7 occurs on the master node exactly once for each merger, i.e., $\sqrt{N}$ times in total.

Messages between nodes			
Worker → Master	**Master → Worker**	**Worker → Merger**	**Merger → Master**
Finished	Be merger m_j for p workers	Output	Output
Output	Send output to m_j		
	Send output back		

Table 6 - Five types of messages can occur between nodes in the merger-based algorithm

9.4. Correctness and finiteness

We described the merger-based MMS algorithm in the form of several events. Because of its parallel character, we cannot usually determine which event will occur on which node, or which the order of these events on the individual nodes is. In the following figure, there are depicted the most important moments in the algorithm's work-flow:

- Establishing of a merger (by the master)
- Redirection of a worker to its merger (by the master)
- Sending of the output from a worker to its merger
- Sending of the output from a merger to the master

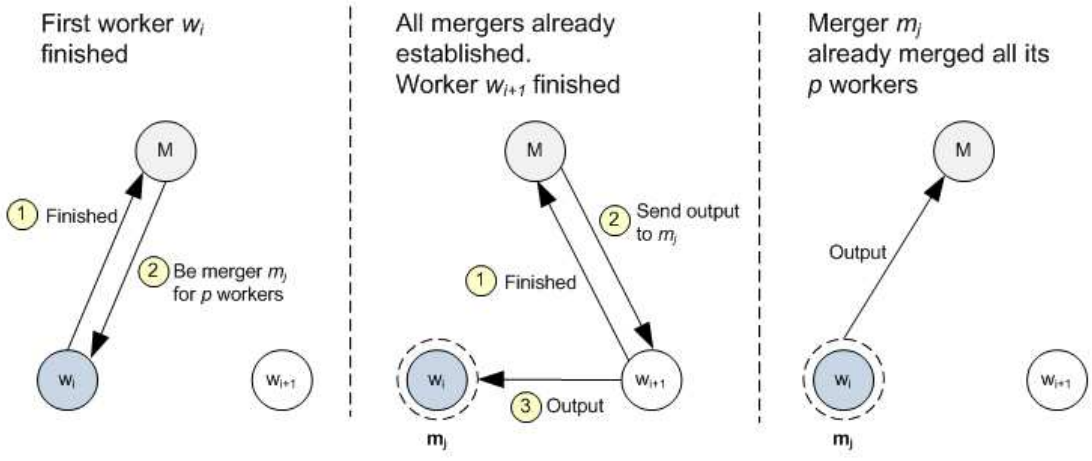


Figure 22 – Merger-based algorithm work-flow (focus on one merger)

9.4.1. Algorithm is finite

By finiteness of the merger-based MMS algorithm, we mean that the modified finalization part can reach its end; i.e., that in the end, master *M* always sends its output to the client. To demonstrate that we help ourselves with the state diagram of the master node depicted in Figure 23. There are 5 states of the master. In fact, the merger-based algorithm starts with the transition from state 1 to state 2, i.e., by finishing of the first worker. In state 2, master *M* establishes finished workers as mergers. In state 3, master *M* redirects each finished worker to some merger. In state 4, master *M* accepts the outputs from finished mergers. When the last $\sqrt{N}$-th merger has finished, master *M* mergers its output with the current one and sends this final result to the client.

We can clearly see that all the loops in the state diagram are finite, restricted by the number of finished workers or mergers. Master *M*, hence, stays in each of the states only for a finite time and therefore always reaches final state 5, which is the sending of the output to the client.

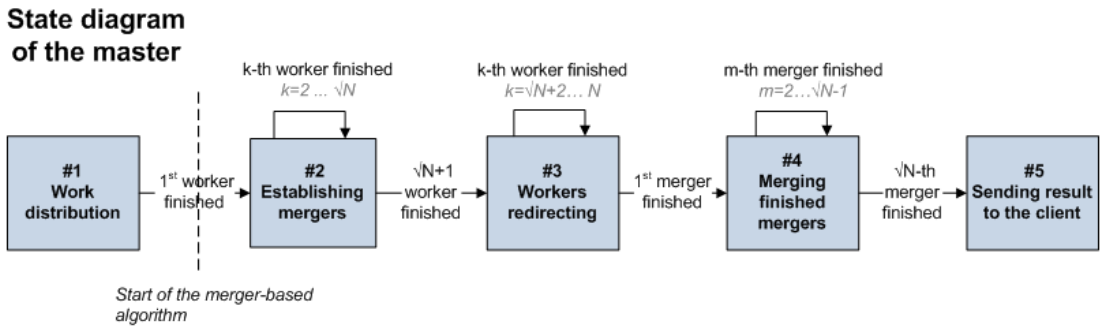


Figure 23 – State diagram of the master in the merger-based algorithm

To make the picture complete, we also provide the state diagram of the worker node (Figure 24). The transition from state 2 to either state 3, or to state 5, is determined by when the worker finishes. If it becomes a merger, it then waits for *p* workers to finish; i.e., the loop in state 3 is also finite, restricted by *p*.

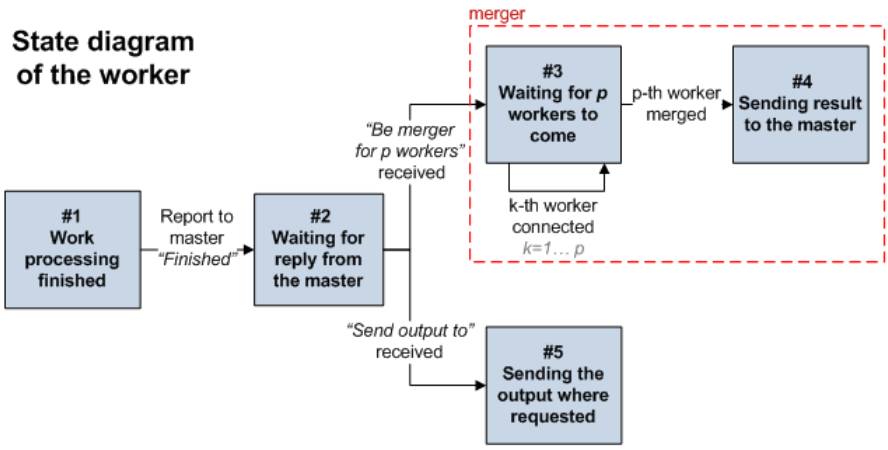


Figure 24 – State diagram of the worker/merger in the merger-based algorithm

9.4.2. Algorithm is correct

By correctness of the merger-based MMS algorithm, we mean that the output list, which is sent from the master to the client, contains merged outputs from all *N* workers; and each output is merged exactly once. In other words, the final output list is the same as if all of the merging would have been done on the master node only. We know that the merging of the output lists is commutative (Chapter 4.5).

Observations:

- **The output of each worker goes to one merger.**
  - Every worker is either established as a merger itself or redirected to exactly one merger (Figure 24 – transition from state #2 to either state #3 or #5).
- **The output of each merger goes to the master.**
  - Each merger sends its output list only once (Figure 24 – state #4).
  - The master does not send its output list to the client until it gets the output lists from all mergers (Figure 23 - transition from state #4 to state #5).

The output list from each worker is merged exactly once on some merger. The output list from each merger is merged exactly once on the master. It means that each output list goes "through" exactly one merger, and all the mergers are merged on the master ⇒ all output lists are merged into the final result.

9.5. Supporting algorithm's robustness

In the previous sub-chapter, we described the work-flow in the ideal case when there are no unexpected events, such as a non-master node failure or a message lost. In order to make the merger-based algorithm more resistant to such accidents; and hence, really usable in the real environment, we introduce the following confirmation messages:

- Confirmation of merger's start-up success *("Merger started")*
- Confirmation of successful merging from a merger to its worker *("Successfully merged")*
- Confirmation of successful merging from a worker to the *master ("Successfully merged on m_j")*

The usage of these three confirmation messages is depicted in the following figure, which is an extension of Figure 22. Further details of these confirmation messages are discussed in the following sub-chapters.

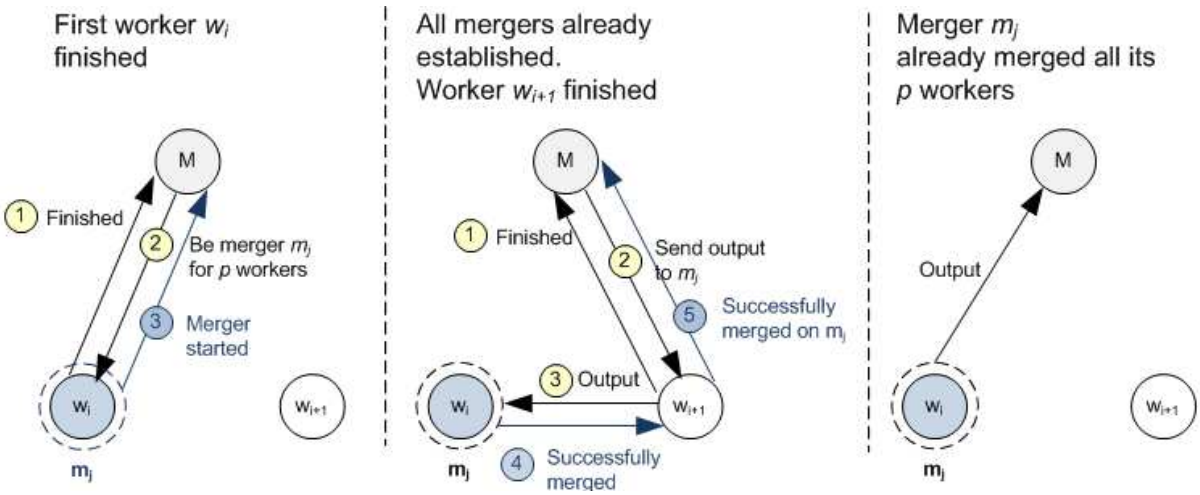


Figure 25 – Use of confirmation messages

The main idea behind the support of the algorithm's robustness is the independence of merging. If a merger fails, its workers can still resend their outputs directly to the master. Here, they are simply merged with the output lists coming from successful mergers.

If a worker fails before it sends out its output, the appropriate part of the input must be processed again. In fact, it is just a special case of a worker failing during the computation, which is handled by the PROOF Packetizer (Chapter 4.4). If the master node fails, then the computation is completely lost as in many similar master-worker based applications.

In the ideal case when all confirmations come as expected, the state diagrams of the master and worker look like the following:

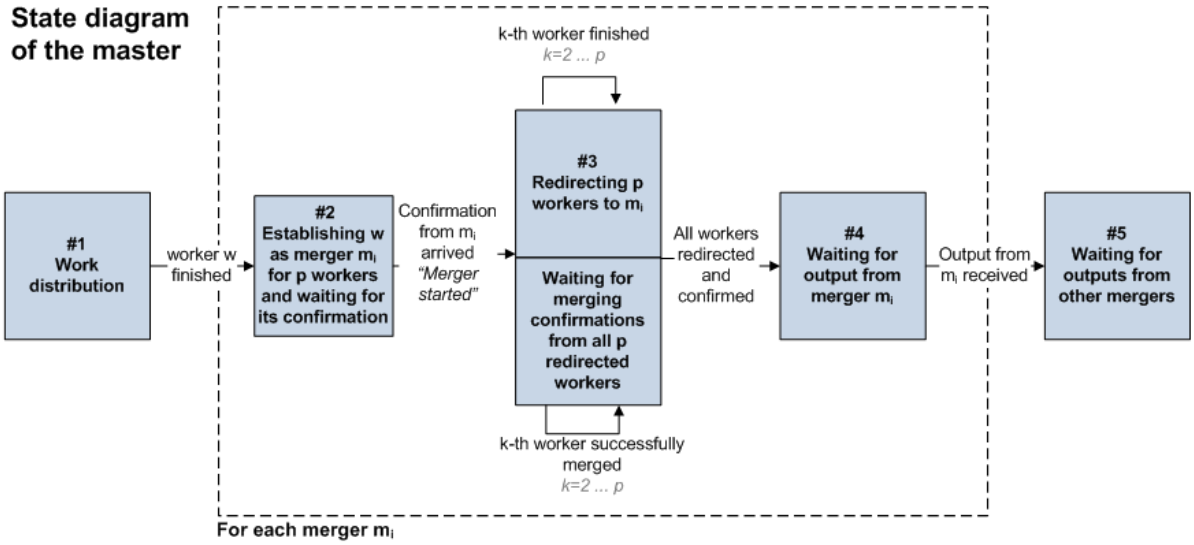


Figure 26 – State diagram of the master including sending/waiting for confirmations

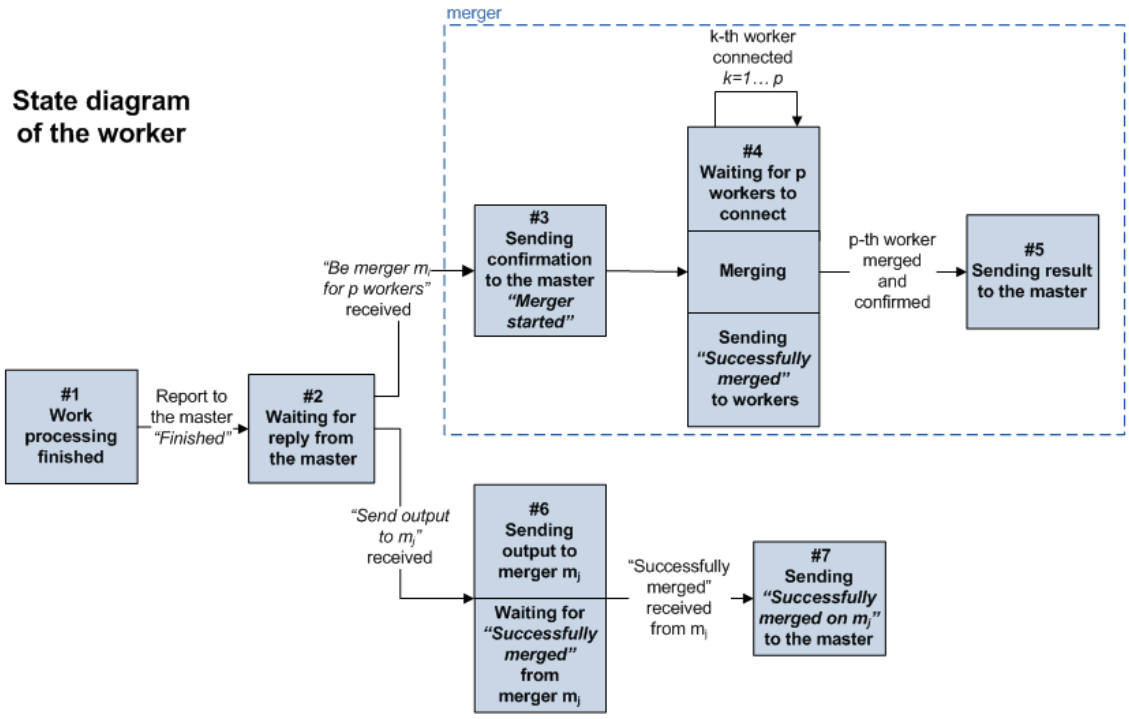


Figure 27 - State diagram of the worker including sending/waiting for confirmations

In the following sub-chapters, we focus on the situations when confirmations do not come within the given time period; or even negative confirmations come. When we say that some confirmation was not received *within the given time period*, it also covers the case when more request attempts were made. We also expect that all the entities can handle possible duplicates of all messages; e.g., more mergers are never started on the same worker node even if multiple requests are received. Both the timeout length and the maximal number of attempts can be set freely according to the properties of the PROOF cluster.

9.5.1. Confirmation of merger's start-up success

More precisely, the title refers to confirmation *"Merger started"* coming from worker w_i to master *M* and confirming that w_i became merger m_j as requested (Figure 25 – the first case).

No workers are redirected to merger m_j by master *M* until the confirmation from this merger arrives on *M*, i.e., until it is sure that m_j was started successfully (Figure 26 – state #2). Finished workers can simply wait until they are redirected by master M (Figure 27 – state #2).

- **Problem 1:** Master *M* has not received confirmation from worker w_i *("Merger started")* within the given time period.
- **Solution:** Worker w_i is asked to send its output directly to master *M ("Send output back").*

- **Problem 2:** Negative confirmation of master start-up arrives *("Cannot be merger"*).
- **Solution:** Worker w_i is asked to send its output directly to master *M ("Send output back").*

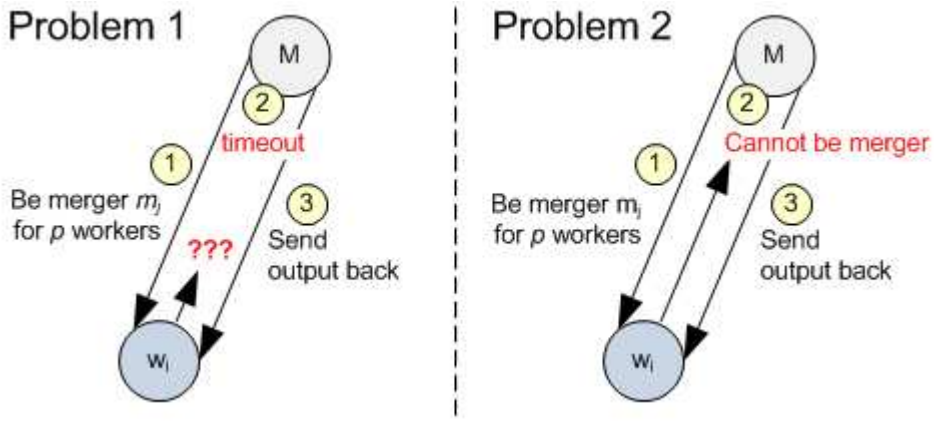


Figure 28 – Possible problems related to confirmation of merger' start-up success

If worker w_i is asked to send its output directly to master *M*, it means that w_i will certainly not serve as a merger anymore. Its merger role is then assigned by master *M* to some other node. This node is simply chosen from the group of waiting workers (which were originally waiting for the unsuccessful node w_i), and the whole cycle, including the waiting for the confirmation repeats. The only change is that the number of workers for the new merger is lowered by one.

In a better case, the unsuccessful worker w_i sends its output to master *M;* and here it is later merged with the outputs from successful mergers.

- **Problem 3:** Worker w_i was asked for output (*"Send output back"*), but no response has come within the given time period.
- **Solution:** Output of worker w_i is considered unreachable.
    - The node, on which w_i was running, is deleted from the list of available nodes; and the cluster administrator is informed about the problem (manual restart can be necessary).
    - The PROOF Packetizer is asked by master *M* to re-assign the work originally processed on w_i, to other nodes. This is performed as a completely standalone query with the same selector, but only with the sub-set of the original data ("emergency" query). This allows its processing as fast as possible; as again, all the nodes may be involved, although used in the different and thus independent sessions.
    - When the "emergency" query is finished, its output does not go to any client, but directly to master *M*, which started the session. Here, this emergency query output is merged with the standard output of the original query. Master *M* cannot finish before this emergency output is received.

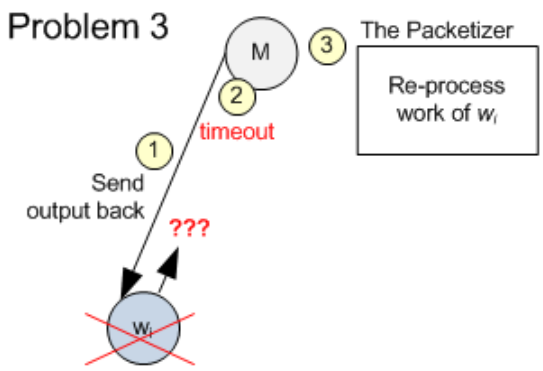


Figure 29 – Worker w_i not responding to request "Send output back"

9.5.2. Confirmation of successful merging from merger to worker

More precisely, the title refers to confirmation *"Successfully merged"* coming from merger m_j to worker w_k and confirming that m_j merged output of w_k successfully (Figure 25 - second case, message 4).

After worker w_k sends its output to merger m_j, it waits for the confirmation. If the confirmation does not come within the given time period, worker w_k sends to merger m_j a special message requesting the confirmation of the previously sent output *("Confirm merging").* This covers the case when the confirmation (*"Successfully merged"*) was sent by m_j, but got lost on the way. The advantage is that the output itself does not have to be transferred again from w_k to m_j.

- **Problem 4:** Confirmation of merging (*"Merged successfully"*) has not come within the given time period.
- **Solution:** The same as for Problem 5.

- **Problem 5:** Negative confirmation of merging came *("Merging unsuccessful").*
- **Solution:** Worker w_k informs master *M* that merger m_j is broken *("Merger m_j down")* and sends its output directly to master *M* (*"Output"*).

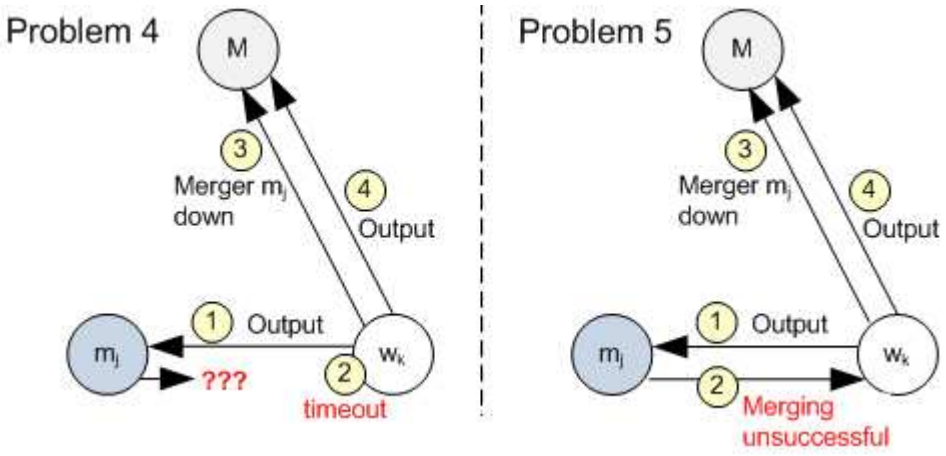


Figure 30 - Possible problems related to the confirmation of successful merging from a merger to a worker

9.5.3. Confirmation of successful merging from worker to master

More precisely, the title refers to confirmation (*"Successfully merged on m_j"*), which comes from worker w_k to master *M* and confirms that output of w_k was successfully merged by merger m_j. As depicted in Figure 25 (second case), ideally this confirmation immediately follows the previous confirmation from merger m_j to worker w_k *("Merged successfully").* Master *M* is awaiting the confirmation on successful merging from each of the redirected workers.

- **Problem 6:** Worker w_k reports its merger m_j as broken *("Merger m_j down")* to master *M.*
- **Solution:** Master *M* immediately stops using merger m_j.
    - The workers, expected to be redirected on m_j, are asked to send their outputs directly to master *M ("Send output back")* when they finish.
    - Master *M* asks for the current output of merger m_j *("Send output back").* If received, it contains the outputs of all the workers, which were successfully merged on m_j before, as well as the output of m_j itself.

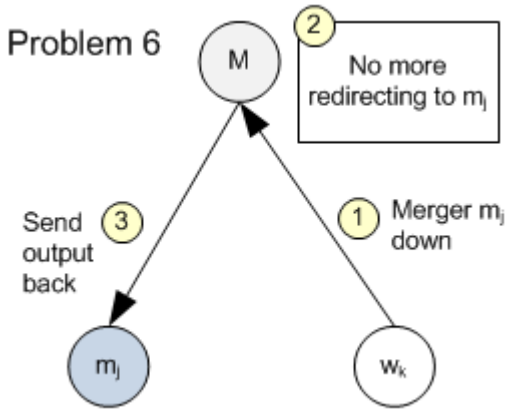


Figure 31 – Worker w_k reporting failed merger m_j

- **Problem 7:** Broken merger m_j was asked for output *("Send output back"),* but no response has come within the given time period.
- **Solution:**
    - Master *M* asks all workers, which have been merged successfully on m_j before, to send their outputs directly to it *("Send output back").*
    - The workers, which are not responding, are considered unreachable; and their part of the work must be processed again, as well as the work of m_j as a worker.
    - The same approach is used as in the solution of Problem 3. The Proof Packetizer is asked to again process the work, originally processed on m_j and all the unreachable workers. Master *M* waits until this result is received, and then it merges it with the results obtained from the successful mergers.

Note that the negative confirmation *("Merger mj down")* comes from worker w_k when there is a problem on merger m_j, but w_k is all right. If no confirmation comes, there is a problem on w_k.

- **Problem 8:** No merging success confirmation *("Merged successfully on m_j")* has come from worker w_k to master *M* within the given time period after w_k's redirecting to m_j.
- **Solution:** We need to discover whether worker w_k has ever sent its output to merger m_j or not. Therefore Master *M* asks merger m_j if it has received output from w_k *("w_k merged?").*

  Merger m_j replies:
  - *"w_k merged"*:
    - As we have the output of w_k merged, we do not have to care about its crash.
  - *"w_k not merged"*:
    - Together with the send-out of this message, merger m_j automatically lowers the total number of workers to merge from *p* to *p-1*.
    - Worker w_k is asked for sending its output directly to master *M ("Send output back").* Possible problems are handled the same way as in the case of Problem 3.
  - If merger m_j does not respond at all, then master *M* behaves as if merger m_j was reported as broken (Problem 7).

When master *M* receives the merging confirmations *("Successfully merged on m_j")* from all workers redirected to merger m_j, it then starts to wait for the output from m_j.

- **Problem 9:** No output has come from merger m_j within the given time period, although all its workers already reported successful merging on m_j to master *M*.
- **Solution:** Master *M* asks merger m_j for the final output explicitly *("Send output back").* If merger m_j does not respond, the same approach as for Problem 7 is used; i.e., all merged workers are asked to re-send their outputs directly to master *M*.

9.5.4. Summary

There are many other possibilities of how to solve unexpected problems of worker's or merger's failure. We decided to choose the most straightforward approach in order to make the work-flow transparent and loggable even in complicated situations. For instance, according to the above mentioned rules, if a worker was once redirected to some merger, it will never be redirected to any other merger again (even if the original merger fails). In such situations, the worker is always asked to send its output directly to the master. Generally, redirecting to some other merger could be faster. However, we decided on this approach

for several reasons. First, such failures are rather rare so the potential slow-down is negligible. Second, redirecting right to the master is easier to control (another merger would have to be informed about another worker on the top of the original limit; it would have to confirm it, etc.). Moreover, once some merger has failed, it is wise to be more careful when relying on other mergers as mergers' failures often happen together due to some shared problems on the cluster.

In the following table, we list all types of messages, which can occur between any pair of nodes including all confirmations and negative confirmations.

Messages between nodes					
Worker → Master	**Master → Worker**	**Worker → Merger**	**Merger → Worker**	**Merger → Master**	**Master → Merger**
Finished	Be merger	Output	Successfully merged	Output	Send output back
Output	Send output to m_j	Confirm merging	Merging unsuccessful	Merger started	w_k merged?
Merger m_j down	Send output back			w_k merged	
Successfully merged on m_j				w_k not merged	
Cannot be merger					

Table 7 - Messages between nodes including confirmations

10. Benchmarks of merger-based algorithm

In this chapter, we present some benchmarks of the previously introduced merger-based MMS algorithm. We do not only provide the comparison of the *standard approach* with the *merger-based algorithm*, but we also measure the finalization speed-up when different than the optimal number of $\sqrt{N}$ mergers is used[1]. Therefore, we run each query on 4 different configurations covering the standard approach, $\sqrt{N}$ mergers, $\sqrt{N-1}$ mergers and $\sqrt{N+1}$ mergers.

The merger-based algorithm is intended for tasks featuring a significant finalization phase, a common characteristic of many HEP analyses. However, it is necessary to ensure that queries with the short finalization can also be processed as fast as possible. In such cases, naturally, we cannot expect any significant speed-up; but we need to make sure that the overhead of mergers does not cause any noticeable prolongation. However, as our test queries also feature the finalization phase shorter than 1 minute, which is also speeded up; we can conclude that the mergers' overhead does not degrade performance of the queries with the short finalization.

10.1. Measurement methodology

10.1.1. Test environment

All the results presented in this chapter were obtained from the Alice CAF cluster [6] at CERN offering the following environment:

- Number of physical cluster nodes: 15 (1x master node, 1x test node, 13x worker node)
- HW configuration of a node: 8x Intel Xeon CPU 2.33GHz, 16 GB RAM
- Network configuration: 2 x Gigabit Ethernet Controller

We will present results obtained on CAF when using the following numbers of workers:

- 26 workers (i.e., 2 worker processes per physical worker node)
- 52 workers (i.e., 4 worker processes per physical worker node)

The worker processes running on the same physical node share the memory (16 GB) of this node.

10.1.2. Test approach

Alice CAF cluster is recently used for the reconstruction of the first LHC data; and therefore, it is not possible to ensure exactly the same conditions for all the runs of our test queries.

[1] N is the number of workers assigned

However, we can take an advantage of the fact that none of the configurations differ in the computation phase, i.e., that the same code runs during the computation regardless of the configuration.

We say that the computation phase lengths in a set of queries are *mutually comparable* if their coefficient of variation is lower than 5%. Coefficient of variation is defined as the ratio of the standard deviation to the mean.

Each test query was run 10 times on each of four tested configurations; i.e., it was run 40 times in total. All 40 runs were performed in a single row, one after another. If all these 40 runs were *mutually comparable* in their computation phase lengths, it means that the external load on the cluster was without significant changes during the time that these 40 queries were running. If all the computation phases are mutually comparable, we can also consider the finalization phases mutually comparable.

In order to be able to get sets of runs, which are mutually comparable in computation, we tested preferably shorter queries, i.e., queries lasting around a few minutes each (see Appendix A for details). In the case of longer queries, it would be impossible to ensure the stability of the external load during the whole time of their execution.

10.1.3. Test data

We tested six different queries, either developed by us or by the ROOT developers for measurement of the PROOF performance. For simplicity, we do not describe their background here. Naturally, we are more interested in their outside characteristics, e.g., the length, finalization speed-up, etc., than in their functionality or physical meaning.

Query name	Input size (# events)	Output size (# output objects)	Output object type
Query A	5,000,000	10,000	General custom object B
Query B	100,000	100	ROOT List of objects
Query C	10,000,000	10,000	General custom object A
Query D	20,000,000	15,000	General custom object B
Query E	1,000,000	10,000	1D histogram
Query F	250,000	25,000	2D histogram

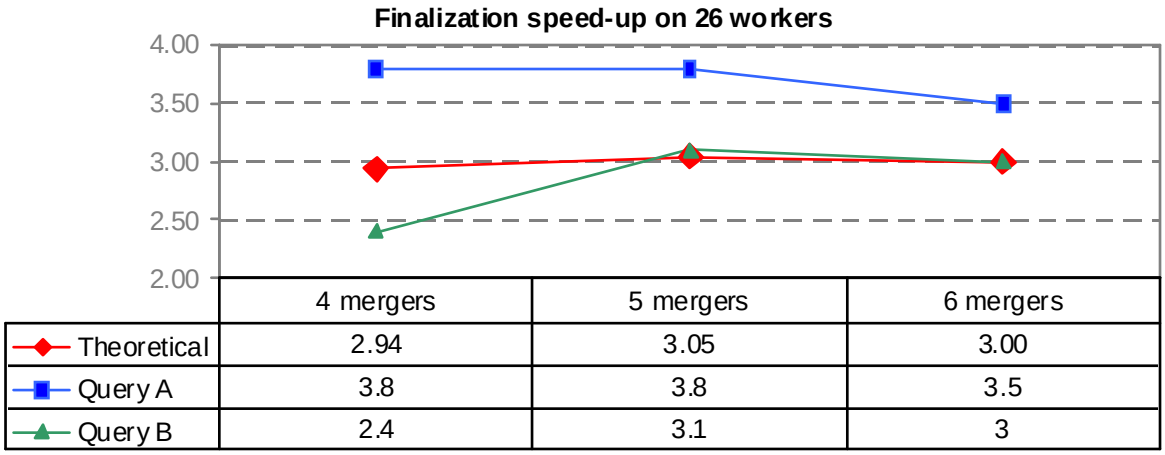
Table 8 – List of tested queries with basic characteristics

10.2. Benchmarks of queries using standard objects

First, we present the finalization speedup for Query A – E, which involve merging of standard objects, i.e., either general ROOT non-optimized objects or custom objects. More details on these measurements are presented in Appendix A.

The theoretical speed-up in the finalization phase when using merger-based algorithm depends on the size of the cluster. The bigger cluster we have, the bigger the finalization speed-up can be. For cluster with *N* nodes, the theoretical speed-up in case of *s* mergers is described by Formula 6 in Chapter 7.6.2.

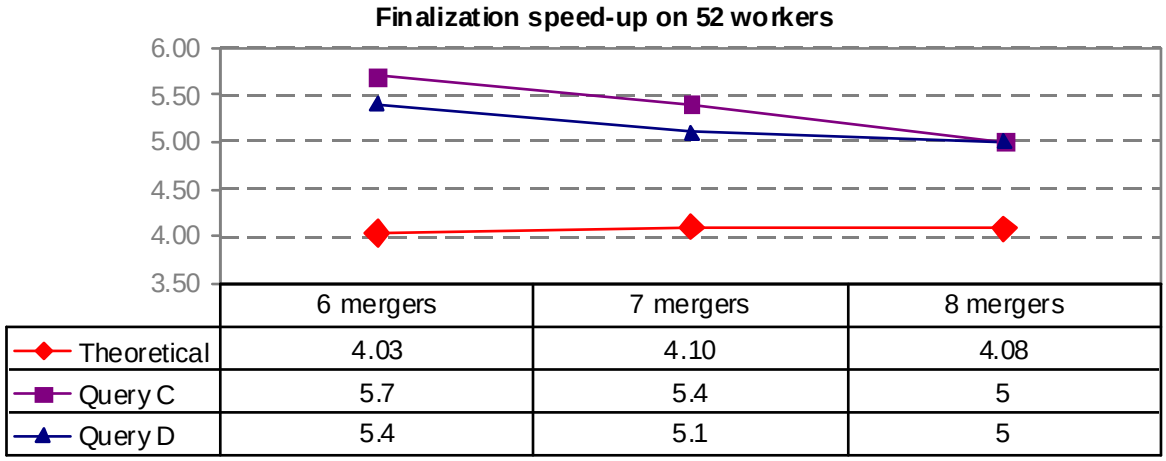
In Figure 32, we compare the theoretical finalization speed-up with the real finalization speed-up reached in Query A and B. Both queries were tested on 26 nodes, having either 4, 5 (optimal) or 6 mergers.



	4 mergers	5 mergers	6 mergers
Theoretical	2.94	3.05	3.00
Query A	3.8	3.8	3.5
Query B	2.4	3.1	3

Figure 32 –Finalization speed-up in PROOF queries using standard objects (26 workers)

In Figure 33, we present the finalization speed-up in Query C and D, which run on 56 workers, having either 6, 7 (optimal) or 8 mergers. The theoretical speed-up, awaited for such configurations, is also displayed.



	6 mergers	7 mergers	8 mergers
Theoretical	4.03	4.10	4.08
Query C	5.7	5.4	5
Query D	5.4	5.1	5

Figure 33 - Finalization speed-up in PROOF queries using standard objects (52 workers)

An interesting observation is that the real speed-up in the finalization phase is usually above the theoretical values. This is especially remarkable considering the fact that the theoretical speed-up does not include any overheads related to the establishing or running of mergers. The most probable explanation of this phenomenon is related to the memory. When merging in parallel, each merger not only merges fewer objects than the single master; but it can also use its own memory for that. Therefore, the total amount of memory available for merging is simply bigger. This supports the hypothesis of less page faults during the merging, which also brings an additional speed-up.

We can also recognize a tendency of reaching the best speed-up on $\sqrt{N-1}$ mergers, instead of on $\sqrt{N}$ mergers. However, even if the difference in the speed-up may seem significant, i.e., 5.4 to 5.7, in real values the difference is usually only in a range of a few seconds (see Appendix A). Moreover, even the theoretical speed-ups for $\sqrt{N}$, $\sqrt{N-1}$ and $\sqrt{N+1}$ mergers are very close to each other. Considering our measurement precision due to the dynamic environment, all the presented theoretical speed-ups are simply equal for constant N (i.e., 3 for 26 workers, 4 for 52 workers). It is also the reason why we use two decimal places precision for theoretical values, but only one decimal place precision for real values. Nevertheless, the most important fact is that there is no significantly better number of mergers than $\sqrt{N}$, which basically confirms our theory of $\sqrt{N}$ as an optimum. However, $\sqrt{N-1}$ or $\sqrt{N+1}$ mergers can also be considered as good options if, from any reason, a little less or more mergers are needed.

10.3. Benchmarks of queries using optimized objects

Recently, some optimizations aimed at the merging of selected objects were introduced to the ROOT system. In general, merging is performed by 2; i.e., for N objects, the merging procedure runs $N-1$ times. As a consequence, merging of $2N$ objects takes basically twice as much time as merging of N objects. This fact also served as one of our initial preconditions when determining the optimal number of mergers in Chapter 7.5.

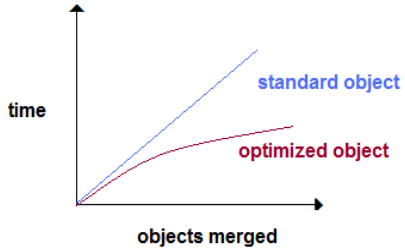


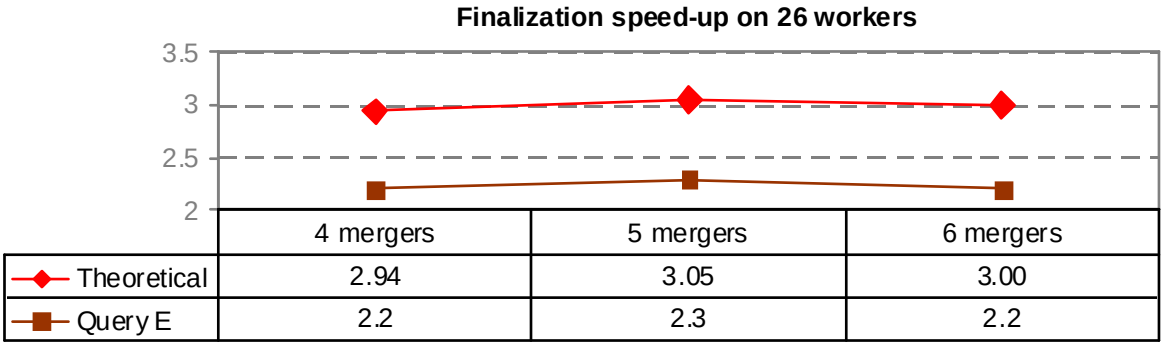
Figure 34 – Time needed for merging of objects increases with the amount of these objects

However, for optimized merging, there is no linear relation as objects are merged in larger groups and in a more sophisticated way. In fact, its authors focused on the same problem as

we are solving in this work, but they decided for a different approach. Instead of trying to parallelize the critical part of the merging as we do, they improved some of the merging procedures themselves. As a consequence, the real curve of the time needed for the merging of *N* objects may looks like the one in Figure 34. The exact curve's shape naturally depends on the concrete type of the optimized object.

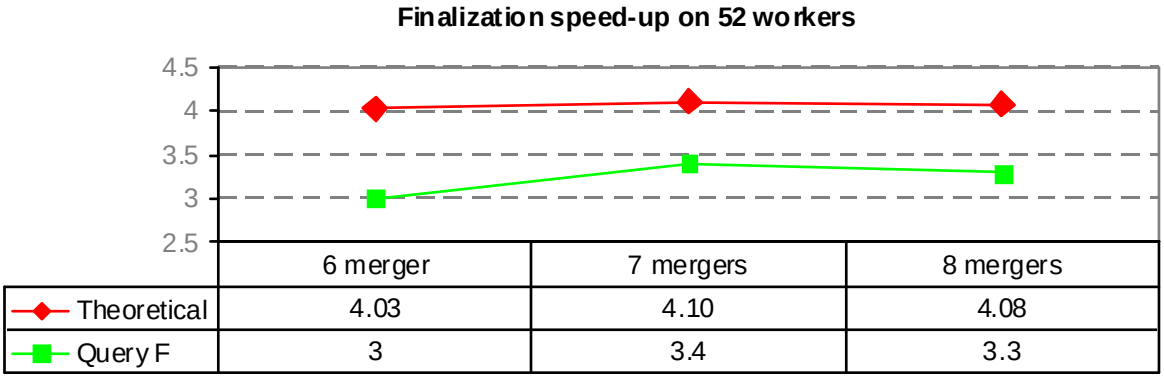
As both these approaches do not go against each other, and thus can be used together, we decided to test the merger-based algorithm for the optimized objects, too.

In the following figures, we can see the results gained for Queries E and F which are build on optimized objects, representing a 1D and s 2D histogram. Again, we present results obtained on 26 and 52 nodes and we confront them with our theory.



	4 mergers	5 mergers	6 mergers
Theoretical	2.94	3.05	3.00
Query E	2.2	2.3	2.2

Figure 35 –Finalization speed-up in PROOF queries using optimized objects (26 workers)



	6 merger	7 mergers	8 mergers
Theoretical	4.03	4.10	4.08
Query F	3	3.4	3.3

Figure 36 –Finalization speed-up in PROOF queries using optimized objects (52 workers)

As expected, the finalization speed-up in these queries was lower than in the queries featuring standard objects. However, the speed-up is still significant enough; and thus, the merger-based algorithm can be also successfully used for the queries built on the optimized objects.

The finalization speed-up for these queries is, nevertheless, hard to describe or evaluate in general as it is strongly object-dependent. In order to adjust the merger-based algorithm also for these specific objects, we would have to examine them one by one and analyze their optimized merging procedures. Since the optimization was introduced only a few weeks before the deadline of this project, we consider it to be beyond the limits of our work.

11. Conclusion

In this thesis, we dealt with the problem of multi-master setup, which was defined in Chapter 1. Our goal was to design and implement a multi-master setup algorithm for PROOF, a specialized framework for parallel data analysis at CERN. The main motivation of our effort was basically to speed-up the data analysis in physical research.

We described the PROOF system as well as the master-worker paradigm, on which PROOF is built. After the initial discussion on the paradigm's advantages and limitations, we presented recent knowledge on the topic. Further, we thoroughly analyzed PROOF query processing in the case of the single and multi-master configuration; and we found possible areas for its work-flow improvement. Based on that, we designed the *record-based algorithm*, and we explained reasons why we later decided to abandon it.

Finally, we presented the *merger-based algorithm*, which is considered the main outcome of this work. We provided a detailed description of it in both a simple and a more robust form. We also created its pilot implementation, which was deployed in the real environment of one of CERN's computing clusters. In order to evaluate the algorithm's qualities, we conducted several tests and analyzed and clarified their results.

After benchmarks' evaluation, we can conclude that the merger-based algorithm did meet our expectations, based on the previously invented theory. Considering the standard objects, the real finalization speed-up usually even exceeds the theoretical values (due to better memory utilization). For specific optimized objects, the finalization speed-up is, according to our expectations, lower, but still distinctive enough to make the long analyses perceptibly shorter. Therefore, we can state that the goal of this project was fulfilled. We designed and implemented a multi-master setup algorithm, which really speeds up the data analysis in defined, but common cases.

The merger-based algorithm is already incorporated in the official ROOT repository and will be part of release 5.26, which is scheduled for the end of December 2009.

11.1. Future work

Even if delivering awaited speed-up, there are still possible ways the algorithm can be improved in the future. The most natural next step would be to analyze merging in the case of individual optimized objects (Chapter 10.3), preferably those to be most likely used by PROOF users, e.g., 1D or 2D histograms. Based on that, a new estimation for the speed-up can be set. Possibly, some changes would be made to the algorithm itself. The recognition of the output object type can be included right into the beginning of the algorithm. Instead just informing on its finishing, a worker would also tell the master which type(s) of object(s) it has

in its output list. Some of the next steps of the algorithm can then be influenced by that information.

Another field of future research and possible improvement is the choice of merger nodes. Now, the first finished, i.e., the fastest, nodes are established as mergers. In small and middle-sized clusters (< 100 workers), all workers finish almost at the same time, usually within a few seconds. However, in larger clusters, the span of the finished times can get longer; and therefore, we can consider also alternate approach when establishing mergers. In other words, we would not establish the fastest nodes as mergers because they would all wait too long for their first redirected workers. Instead, we could establish a merger and redirect a worker alternately as workers are finishing. The question, then, would be the proportion of mergers and redirected workers (e.g., the first and each 5th finished node becomes merger). Could this approach make a difference? Could it help or would it give worse results? It is an interesting topic and definitely worth future research.

References

[1] Worldwide LHC Computing Grid, http://lcg.web.cern.ch.

[2] Brun, R., Rademakers, F.: ROOT - An Object Oriented Data Analysis Framework, Proc. of AIHENP: New Computing Techniques in Physics Research, Lausanne, Switzerland, 1996, http://root.cern.ch.

[3] Ballintijn, M., Roland, G., Brun, R. and Rademakers, F.: The PROOF Distributed Parallel Analysis Framework Based on ROOT, Proc. of the Conference for Computing in High-Energy and Nuclear Physics, La Jolla, California, 2003.

[4] Ganis, G., Iwaszkiewicz, J., Rademakers, F.: Data Analysis with PROOF, Proc. of the XII Advanced Computing and Analysis Techniques in Physics Research, Erice, Italy, 2008.

[5] ALICE, A Large Ion Collider Experiment, http://aliceinfo.cern.ch.

[6] Grosse-Oetringhaus, J.: The CERN Analysis Facility - A PROOF Cluster For Day-One Physics Analysis, Proc. of International Conference on Computing in High Energy and Nuclear Physics, Victoria, British Columbia,
http://aliceinfo.cern.ch/Offline/Activities/Analysis/CAF

[7] Basney, J., Raman, B. and Livny, M.: High throughput Monte Carlo, Proc. of the Ninth SIAM Conference on Parallel Processing for Scientific Computing, San Antonio Texas, 1999.

[8] Cantu-Paz, E.: Designing efficient master-slave parallel genetic algorithms, Genetic Programming: Proc. of the Third Annual Conference, San Francisco, Morgan Kaufmann, 1998.

[9] Govindan, V. and Franclin, M.: Application Load Imbalance on Parallel Processors, Proc. of the 10th International Parallel Processing Symposium, Honolulu,. Hawaii, 1996.

[10] Aida, K., Natsume, W., Futakata, Y.: Distributed Computing with Hierarchical Master-worker Paradigm for Parallel Branch and Bound Algorithm, Proc. of the 3rd International Symposium on Cluster Computing and the Grid, Tokyo, Japan, 2003.

[11] Sbal, G., Berman, F., Wolski, R.: Master/slave Computing on the Grid, Proc. of the 9th Heterogeneous Computing Workshop, IEEE Computer Society, 2000.

[12] Banino, C.: Optimizing Locationing of Multiple Masters for Master-Worker Grid Applications, Proc. of the Workshop on Applied Parallel Computing, Lyngby, Denmark. 2004.

[13] Shao, G., Berman, F., and Wolski, R.: Using effective network views to promote distributed application performance, Proc. of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, USA 1999.

[14] Ausiello, G., Protasi, M., Marchetti-Spaccamela, A., Gambosi, G., Crescenzi, P., Kann, V.: Complexity and Approximation: Combinatorial Optimization Problems and Thein Approximability Properties. Springer-Verlag New York, 1999.

[15] Goto M.: CINT, C++ Interpreter, CQ publishing, ISBN4-789-3085-3 (Japanese), http://root.cern.ch/root/Cint.html.

[16] Ganis, G., Iwaszkiewicz, J., Rademakers, F.: Scheduling and Load Balancing in the Parallel ROOT Facility (PROOF), Proc. of the XI International Workshop on Advanced Computing and Analysis Techniques in Physics Research, Amsterdam, the Netherlands, 2007.

[17] MonALISA, MONitoring Agents using a Large Integrated Services Architecture, http://monalisa.cern.ch/monalisa.html.

[18] ROOT Reference Guide, http://root.cern.ch/root/html.

[19] IANA, Internet Assigned Numbers Authority, http://www.iana.org.

Appendix A – Benchmarks of merger-based algorithm

All tested configurations are listed in the first column called *Configuration*. The following two columns show mean values and standard deviations (sd) of the computation (including initialization) and finalization phases. In the third column, there is the speed-up in the finalization phase compared to the standard setup. The total execution time and the total execution speed-up are displayed in the 4th and 5th column. Naturally, the total execution speed-up strongly depends on the original finalization share in the total execution, which is individual for each tested query.

Please note that all differences in the computation average within one query are only due to the variability of the real environment. Ideally, the computation phase within one query should have the same duration, regardless on the configuration (Chapter 10.1.2).

All time data are stated in minutes.

Query A (run on 26 workers)

Configuration	**Computation average*** **(sd)**	**Finalization average** **(sd)**	**Average speed-up in finalization**	**Total execution time** **(sd)**	**Average speed-up in execution**
Standard setup	1:51 (0:01)	1:09 (0:04)	-	3:00 (0:04)	-
4 mergers	1:50 (0:01)	0:16 (0:00)	4.3	2:06 (0:01)	1,4
Merger-based algorithm (5 mergers)	1:49 (0:00)	0:18 (0:01)	**3.8**	2:08 (0:01)	1,4
6 mergers	1:50 (0:01)	0:20 (0:01)	3.5	2:10 (0:02)	1,4

Coefficient of variation in computation phase: 0,9%

Query B (run on 26 workers)

Configuration	**Computation average*** **(sd)**	**Finalization average** **(sd)**	**Average speed-up in finalization**	**Total execution time** **(sd)**	**Average speed-up in execution**
Standard setup	2:48 (0:03)	0:51 (0:03)	-	3:39 (0:05)	-
4 mergers	2:52 (0:03)	0:21 (0:02)	2.4	3:13 (0:01)	1,1
Merger-based algorithm (5 mergers)	2:53 (0:03)	0:16 (0:00)	**3.1**	3:09 (0:03)	1,2
6 mergers	2:50 (0:09)	0:17 (0:01)	3	3:07 (0:10)	1,2

Coefficient of variation in computation phase: 3,3%

Query C (run on 52 workers)

Configuration	**Computation average* (sd)**	**Finalization average (sd)**	**Average speed-up in finalization**	**Total execution time (sd)**	**Average speed-up in execution**
Standard setup	1:53 (0:00)	2:05 (0:08)	-	3:58 (0:08)	-
6 mergers	1:49 (0:00)	0:22 (0:01)	5.7	2:11 (0:01)	1,8
Merger-based algorithm (7 mergers)	1:51 (0:01)	0:23 (0:02)	**5.4**	2:14 (0:01)	1,8
8 mergers	1:50 (0:01)	0:25 (0:02)	5	2:15 (0:02)	1,8

Coefficient of variation in computation phase: 4%

Query D (run on 52 workers)

Configuration	**Computation average* (sd)**	**Finalization average (sd)**	**Average speed-up in finalization**	**Total execution time (sd)**	**Average speed-up in execution**
Standard setup	5:28 (0:04)	6:09 (0:18)	-	11:37 (0:17)	-
6 mergers	5:26 (0:01)	1:08 (0:04)	5.4	6:34 (0:01)	1,8
Merger-based algorithm (7 mergers)	5:29 (0:11)	1:12 (0:04)	**5.1**	6:41 (0:13)	1,7
8 mergers	5:27 (0:01)	1:20 (0:05)	5	6:48 (0:06)	1,7

Coefficient of variation in computation phase: 2,3%

Query E (run on 26 workers)

Configuration	**Computation average* (sd)**	**Finalization average (sd)**	**Average speed-up in finalization**	**Total execution time (sd)**	**Average speed-up in execution**
Standard setup	2:55 (0:03)	1:40 (0:03)	-	4:35 (0:05)	-
4 mergers	2:56 (0:03)	0:43 (0:01)	2.3	3:39 (0:03)	1,3
Merger-based algorithm (5 mergers)	2:55 (0:03)	0:45 (0:01)	**2.2**	3:40 (0:04)	1,3
6 mergers	2:55 (0:03)	0:48 (0:02)	2.2	3:43 (0:02)	1,3

Coefficient of variation of computation phase: 1,7%

Query F (run on 52 workers)

Configuration	**Computation average* (sd)**	**Finalization average (sd)**	**Average speed-up in finalization**	**Total execution time (sd)**	**Average speed-up in execution**
Standard setup	2:48 (0:03)	17:25 (2:01)	-	20:13 (1:59)	-
6 mergers	2:51 (0:08)	5:48 (0:11)	3	8:33 (0:16)	2,3
Merger-based algorithm (7 mergers)	2:50 (0:05)	5:09 (0:07)	**3.4**	7:59 (0:07)	2,5
8 mergers	2:51 (0:12)	5:14 (0:09)	3.3	8:05 (0:07)	2,5

Coefficient of variation of computation phase: 4.2%

Appendix B – Content of enclosed DVD

/root	A complete copy of the ROOT Subversion repository (trunk), revision 31416. The current trunk can be obtained from *https://root.cern.ch/svn/root/trunk*
mergers101209.diff	The patch with the merger-based algorithm. It contains all necessary changes for its integration in the ROOT/PROOF system. Directly applicable to the enclosed ROOT version.
opocenska_thesis.pdf	Electronic version of this text.